November 2020

Unifying Security Policy Enforcement: Theory and Practice

Shamaria Engram
University of South Florida

Unifying Security Policy Enforcement: Theory and Practice

by

Shamaria Engram

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

Major Professor: Jay Ligatti, Ph.D.
Yao Liu, Ph.D.
Lawrence Hall, Ph.D,
Sanjukta Bhanja, Ph.D.
Theodore Molla, Ph.D.

Date of Approval:
October 30, 2020

Keywords: Security mechanisms, Runtime enforcement, Code granularity, Safety, Liveness

Acknowledgments

I'd like to thank my major professor Dr. Jay Ligatti for his guidance, collaborative efforts, and help developing the research ideas presented in this dissertation. Thank you to my committee members Dr. Yao Liu, Dr. Lawrence Hall, Dr. Sanjukta Bhanja, and Dr. Theodore Molla for helpful feedback and thought provoking questions on the work presented in this dissertation.

I'd like to extend a heartfelt thanks to Mr. Bernard Batson, Dr. Sanjukta Bhanja, and Dr. Sylvia Thomas for all of their mentorship and support during my time at the University of South Florida.

I'd like to thank the National Science Foundation's Graduate Research Fellowship Program, the National GEM Consortium, the McKnight Florida Education Fund, the Florida Georgia Louis Stokes Alliance for Minority Participation, and Sloan Scholars program for their funding support.

Thank you to my friends and family for their encouraging words and support during this journey.

Most importantly, I'd like to thank my Lord and Savior for sustaining me through this challenging process by His unmerited grace and favor.

Table of Contents

List of Tables

List of Figures

Abstract

Security policies stipulate restrictions on the behaviors of systems to prevent them from behaving in harmful ways. One way to ensure that systems satisfy the constraints of a security policy is through the use of security enforcement mechanisms. To understand the fundamental limitations of such mechanisms, formal methods are employed to prove properties and reason about their behaviors. The particular formalism employed, however, typically depends on the time at which a mechanism operates.

Mechanisms operating before a program's execution are static mechanisms, and mechanisms operating during a program's execution are dynamic mechanisms. Static mechanisms are fundamentally limited in the types of policies that they can enforce, due to the lack of runtime information. However, the class of policies enforceable by particular types of dynamic mechanisms typically depends on the capabilities of the mechanism.

An open, foundational question in computer security is whether additional sorts of security mechanisms exist. This dissertation takes a step towards answering this question by presenting a unifying theory of security mechanisms that casts existing mechanisms into a single framework based on the granularity of program code that they monitor. Classifying mechanisms in this way provides a unified view of security mechanisms and shows that all security mechanisms can be encoded as dynamic mechanisms that operate at one or more levels of program code granularity. This unified view has allowed us to identify new types of security mechanisms capable of enforcing security policies at various levels of code granularity. This dissertation also demonstrates the practicality of the theory through a prototype implementation that enables security policies to be enforced on Java bytecode

applications at various levels of code granularity. The precision and effectiveness of the implementation hinges on an extensible Java library that we have developed, called JaBRO, that enables runtime code analysis on optimized Java bytecode at runtime. It is shown that JaBRO allows some security policies to be enforced more precisely at runtime than statically operating mechanisms.

Chapter 1: Introduction

Formal methods for security have become increasingly popular within the last few decades. For some systems, incorrect behavior can be detrimental for users of the system; for example, the malfunction of an avionics system may be life threatening to passengers of an aircraft [41]. Traditional testing methods (e.g., unit tests) cannot provide sound guarantees that a system will always behave correctly. This limitation is a result of the complexity of computer systems, which are typically capable of operating in an infinite number of ways. Testing the functionality of a system can only result in the guarantee that a system behaves correctly with respect to a finite set of test cases; therefore, it is nearly impossible for programmers to consider all possible test cases if a system can take infinitely many possible inputs.

Formal methods provide mathematical means to rigorously prove whether a system behaves in accordance with a formal specification [31]. With respect to security, these specifications are called security policies and typically place constraints on untrusted programs. In practice, systems should be able to interact with, or run, untrusted programs. For example, email users may desire the ability to receive attachments through email. However, email attachments provide a way to spread viruses to computer systems [78]. To aid in the detection of malicious attachments, some email users may use a security mechanism, such as antivirus software or a spam filter, to prevent downloading such attachments. However, due to fundamental limits of computation, one should not blindly trust security mechanisms that claim to enforce particular security policies, but should be able to verify such claims.

Therefore, it is imperative to understand which policies are enforceable by particular types of security mechanisms.

To provide guarantees about which policies security mechanisms can enforce, formal frameworks are employed to rigorously model and reason about their behaviors. However, the type of formalism employed generally depends on whether the mechanism enforces policies statically or dynamically. Some policies can be enforced either way. One such example is type safety, which is a policy that partitions well-typed programs from ill-typed programs. Type safety is an important policy because it guarantees that all well-typed programs will behave in well-defined ways. Type checkers that enforce type safety statically only permit well-typed programs to execute. On the other hand, dynamic type checkers enforce type safety during a program's execution and only permit well-typed program statements, or expressions, to execute. Therefore, dynamic type checkers do not explicitly guarantee that whole programs are type safe before their execution, but rather that program sub-expressions, observed up-to the currently executing sub-expression, are type safe. Although some policies, such as type safety, can be enforced either statically or dynamically, there are important trade-offs that exist between static and dynamic enforcement.

Static mechanisms (e.g., static type checkers or virus scanners) analyze all of a program's source [28], intermediate (e.g., [42, 26]), or binary code (e.g., [19, 56]) before its execution. Such mechanisms allow policy violations to be caught before code is put into production without inhibiting a program's runtime performance. However, it is undecidable to determine statically whether an arbitrary program satisfies a nontrivial policy [80, 89, p. 219]. Consequently, static code analyzers typically enforce policies conservatively and sometimes report false positives [28]. This problem prevents many developers and security practitioners from using static code analyzers due to the onerous task of figuring out which reports are actually a policy violation [52]. Other static code analyzers aim to reduce the burden on the developer of having to manually verify which reports are false positives by

being more precise (e.g., [99, 87]); however, this approach often comes with introducing false negatives, which provides no guarantees that code is free from vulnerabilities [28].

Dynamic, or runtime, mechanisms monitor program events during a program's execution and intervene as necessary. For example, web browsers may prevent unsafe JavaScript functions, such as `eval`, from executing [67], mobile-based enforcement mechanisms may prevent unauthorized use of location-based services [69], and operating systems may limit, or prevent entirely, the use of particular system resources based on user privilege levels. These mechanisms can enforce some policies more precisely than static mechanisms because of available runtime information, but generally sacrifice runtime performance for precision.

Hybrid mechanisms combine a static and dynamic approach to policy enforcement and can enforce policies that are difficult to enforce by just a static or dynamic mechanism alone. For example, many information flow control mechanisms combine static and dynamic enforcement (e.g., [82, 83, 20]), such as a static type checker and a runtime monitor to prevent public outputs from revealing information about secret inputs. Some program rewriters—mechanisms that modify programs prior to their execution to ensure policy satisfaction [46]—can be viewed as a particular type of hybrid that is able to perform static code analysis and inline checks to enforce policy-specific constraints at runtime for statically undecidable policies.

The static-dynamic view of security mechanisms is important for providing a foundation for characterizing the class of policies enforceable by both static and dynamic mechanisms. However, this perspective presents a problem in addressing two open, foundational research questions in computer security:

1. Which policies are precisely enforceable by hybrid mechanisms? To precisely show the class of policies enforceable by hybrid mechanisms, one must be able to present a rigorous proof of the types of policies enforceable by the mechanism as a whole. However, a precise characterization of the class of policies enforceable by hybrid mechanisms has

been difficult to derive because different proof techniques are used for the static and dynamic components [82].

2. Do additional mechanisms exist, beyond: static code analyzers, runtime mechanisms, and hybrids? Research has shown that, given the necessary capabilities, some security mechanisms can enforce policies that other mechanisms cannot (e.g., [61, 62, 33]), which raises the question of whether additional mechanisms exist that can enforce even larger classes of security policies.

A first step to answering these questions might be to reason about security mechanisms more uniformly by casting existing mechanisms into a single framework. A holistic view of security mechanisms and policies might help us to better understand the capabilities of various security enforcement mechanisms and provide a better understanding of how arbitrary mechanisms, including hybrids, may enforce policies in practice.

This dissertation thus introduces an alternative perspective for classifying security mechanisms based on the granularity of code that they monitor.

1.1 Contributions

The main contributions of this dissertation are twofold: (1) we present a unifying theory of security enforcement mechanisms by encoding existing mechanisms as runtime mechanisms that operate at one or more levels of program code granularity, and (2) we demonstrate the practicality of the theory with an implementation of a framework for enforcing security policies at various levels of code granularity on Java bytecode applications.

The unifying theory has allowed us to identify new kinds of runtime security mechanisms capable of operating at various levels of code granularity, and develop a new taxonomy of security policies that can aid in determining how particular security policies might be enforced on untrusted programs. The implementation hinges on an extensible Java library,

called JaBRO, that we have developed to enable runtime code analysis at several levels of code granularity on Java bytecode applications. JaBRO enables certain security policies conservatively enforced by static mechanisms to be enforced more precisely at runtime.

1.2 Organization

This dissertation is organized as follows:

1. Chapter 2 discusses background and related work on formal frameworks and security mechanisms. Section 2.1 discusses how formal frameworks have been used to reason about security policies over systems and presents formal definitions for security policies. Section 2.2 discusses existing classes of security mechanisms and shows how there does not exist a unified framework capable of capturing all security mechanisms.

2. Chapter 3 introduces the first contribution of this dissertation, a unified model of security mechanisms parameterized by the granularity of program code that mechanisms monitor. This chapter shows how this model unifies existing classes of security mechanisms and presents a new taxonomy of security policies. Additionally, this chapter gives several example security policies and shows how they can be encoded at various levels of code granularity.

3. Chapter 4 presents a practical example of how forward secrecy, a desirable property of authentication methods, can be enforced at a fine-grained level on a variant protocol of an authentication method called Coauthentication [63].

4. Chapter 5 presents the second contribution of this dissertation, an implementation of the unified framework that enables security policies to be enforced on Java bytecode applications at various levels of code granularity. The chapter shows how the precision

and effectiveness of the framework depends on an extensible Java library, called JaBRO, that we have developed to enable runtime code analysis on Java bytecode applications.

5. Chapter 6 presents details about the evaluation of the implementation and shows how we evaluated the effectiveness and efficiency of the implementation by enforcing security policies at various levels of code granualrity on two popular, open-source Java bytecode applications.

6. Chapter 7 concludes with a summary of the contributions of this dissertation and provides directions for future work.

Parts of the text presented in this dissertation have appeared in two published conference papers [63, 36]. Permission to use material from the aforementioned papers are presented in Appendix A.

Chapter 2: Background and Related Works

This chapter discusses prior work on formal frameworks for reasoning about security policies and mechanisms. Section 2.1 describes two categories of formal frameworks that have been used in computer security to reason about systems, their executions, and policies expressed over such systems. This section also presents formal definitions of policies, properties, safety, and liveness that will be used throughout this dissertation. Section 2.2 discusses existing classes of security mechanisms and shows how there does not exist a unified framework to reason about all such mechanisms.

2.1 Formal Frameworks

Formal frameworks are used to specify and reason about security policies over systems. A system can be modeled as a set of executions, and policies can be expressed as sets of valid systems.

Formal frameworks for security can be categorized as either system centric or mechanism centric. System-centric frameworks reason about systems as monolithic units that execute events (e.g., [77, 86, 40]), whereas mechanism-centric frameworks reason about systems as a composition of distinct components, such as an untrusted program, a security mechanism, and an underlying executing system, each of which can exhibit events [61].

This section focuses on general-purpose, automata-theoretic frameworks (e.g., [86, 61, 62, 79, 33]) that encode system executions as sets of traces, where each trace represents a single run of a system.

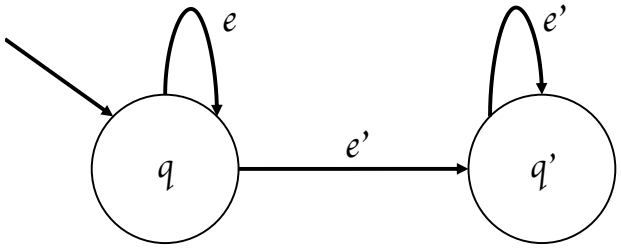

Figure 2.1. State machine model of a system.

2.1.1 System-Centric Frameworks

System-centric frameworks model systems as state machines that transition between states by executing events, as shown in Figure 2.1. There are two ways to encode the executions of systems under a system-centric view: 1) as sequences of states, or 2) as sequences of observable events (i.e., events executed by the system). Policies over such systems are sets of valid system executions.

States encode the condition of the system at each step of its computation (e.g., variable values) [57, 5]. This formalism began with research in program verification, where the objective was to ensure that systems remained in acceptable states at each step of their computation. Given a policy, encoded as a set of valid executions, it is possible to prove that the policy holds for a particular system if it can be shown that the set of executions describing the system is contained in the policy.

System-centric, event-based traces are encoded as sequences of events executed by systems during their computation (e.g., [62]). This model is sometimes used in security research because it is not only important to ensure that systems remain in acceptable states at each step of their computation, but also to ensure that systems only execute valid events to reach acceptable states. Event-based traces can be more expressive than state-based traces because a system may execute several, possibly different, events before transitioning between states. Due to this added expressiveness, this dissertation will only present formal notation for system-centric, event-based traces.

Let E, ranged over by metavariable e, denote the set of all possible system events. A system execution, or trace, x is a finite or infinite sequence of such events (e.g., $e_1, e_2, e_3...$). Let E^* denote the set of all possible finite-length traces, E^ω denote the set of all possible infinite-length traces, and E^∞ denote the set of all possible traces (i.e., $E^* \cup E^\omega$). A trace $x \in E^*$ that is a prefix of trace $y \in E^\infty$ is denoted by $x \preceq y$; conversely, a trace y that extends trace x is denoted by $y \succeq x$. The empty trace (i.e., a trace containing 0 events) is notated as ε and $\mid x \mid$ denotes the size of a trace x (i.e., the number of events appearing in x).

System-centric policies only encode, what are called, observable events [17]. Observable events are events that are actually executed by the system and cause the system to take a computational step. Events attempted by an untrusted program that are prevented from executing by a security mechanism are considered unobservable events and do not appear on system traces. For example, consider a policy that states "classified data must not be sent over unsecured networks". If a program attempts to send classified data over an unsecured network but is prevented from doing so by a security mechanism, the interaction between the security mechanism and the program is not described by the system-centric policy because this interaction is considered to be unobservable. Therefore, this model makes it difficult to reason about the effects of security mechanisms on untrusted programs when mechanisms must remedy invalid program events.

2.1.2 Mechanism-Centric Frameworks

Mechanism-centric frameworks encode system executions as sequences of pairs of events, where each pair is called an exchange [33, 79]. The first event in each pair is an event attempted by a component of the system, and the second event in each pair is an event actually executed by the system or an observable event that may be returned by the system. Systems in this framework are modeled as state transducers that transition between states

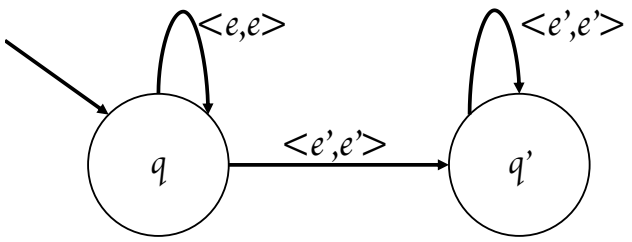

Figure 2.2. State transducer model of a system.

based on exchanges, as shown in Figure 2.2. The ability to distinguish between attempted and executed events is important because policies expressed over such systems can describe the effects of security mechanisms and how they might remedy potentially invalid program events.

Formally, as shown in [79], E denotes the set of possible system events and determines $\mathcal{E}$ the set of possible exchanges, where an exchange can be any pair of events from E. A trace x is a finite or infinite sequence of exchanges (e.g.,$\langle e_0, e_0' \rangle \langle e_1, e_1' \rangle \ldots$). The events in an exchange may not be equal. For example, when a system attempts to execute an event e that violates a policy, a security mechanism may output an alternative event e' that satisfies the policy instead; the resulting exchange is $\langle e, e' \rangle$. The set of all finite-length traces is denoted by $\mathcal{E}^*$, the set of all infinite-length traces is denoted by $\mathcal{E}^\omega$, and the set of all possible traces is denoted by $\mathcal{E}^\infty$ (i.e., $\mathcal{E}^* \cup \mathcal{E}^\omega$). Similar to system-centric trace models, $x \preceq y$ denotes that trace $x \in \mathcal{E}^*$ is a prefix of trace $y \in \mathcal{E}^\infty$; the extension operator $\succeq$ is defined symmetrically. The empty trace is commonly notated as ε and the size of a trace x is notated as $\mid x \mid$ (i.e., the number of exchanges in x).

2.1.3 System-Centric vs. Mechanism-Centric Frameworks

System-centric frameworks are more suitable for program verification because system-centric policies specify only valid system events. System-centric policies therefore specify what systems should do, or the system events that are allowed to execute, but not how partic-

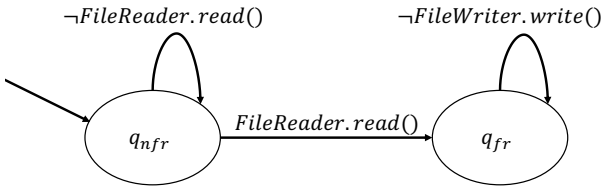

Figure 2.3. System-centric policy: no FileWriter.write() after FileReader.read().

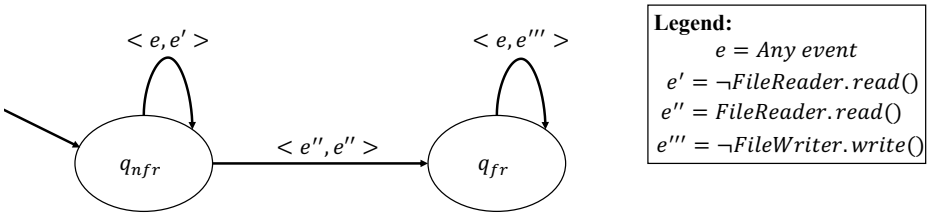

Figure 2.4. Mechanism-centric policy: no FileWriter.write() after FileReader.read().

ular systems can be made to satisfy such policies when they deviate from a particular policy's behaviors. Consequently, system-centric frameworks do not inherently provide the capability to reason about the effects of security mechanisms on untrusted programs. For example, consider the policy described in Figure 2.3; this policy stipulates that `FileWriter.write()` cannot occur after `FileReader.read()` has occured. Suppose an untrusted program attempts to execute `FileWriter.write()` while in state q_{fr}—meaning that a `FileReader.read()` has already occurred—a mechanism enforcing the policy by simulating execution according to the state machine in Figure 2.3 cannot make a transition on `FileWriter.write()` while in state q_{fr}, thereby making it difficult to determine a suitable response to the attempted policy violation.

Mechanism-centric frameworks are more suitable for reasoning about the enforcement of policies on untrusted programs. Mechanism-centric policies encode not only valid events but how invalid events may be transformed into valid events. In other words, mechanism-centric policies describe not only the events allowed by the policy but also the effects of security mechanisms when programs deviate from the behaviors allowed by the policy.

For example, consider the mechanism-centric version of the "no `FileWriter.write()` after `FileReader.read()`" policy described in Figure 2.4. While in state q_{fr} an untrusted program can attempt any event, indicated by e; however, only an event that is not a `FileWriter.write()`, indicated by e''' can actually be executed. Therefore, even if a program attempts a `FileWriter.write()` while in state q_{fr}, it is clear that a mechanism can respond to this attempted policy violation by outputting any event that is not a `FileWriter.write()` event. Next, we will present formal definitions for system-centric and mechanism-centric policies, properties, safety, and liveness.

2.1.4 Policies

Security policies specify constraints on system behaviors and can be defined qualitatively or quantitatively. Qualitative security policies, also known as "black-and-white" policies [79], are predicates over sets of system traces. A qualitative policy distinguishes between secure and insecure systems. Quantitative policies, also known as "gray" policies [79], are functions over sets of system traces. Rather than specifying what makes a system secure or insecure, qualitative policies determine how secure systems are. For example, consider the policy that states "all resources acquired must eventually be released", if a system does not release one particular resource but releases all other acquired resources, a qualitative version of such a policy will consider this system to be insecure. However, a quantitative version of such policy might consider the system to be, say, 90% secure, assuming the policy decreases the security of the system by 10% for each unreleased resource. This dissertation will only present formal definitions for qualitative security policies.

Formally, a system-centric policy P is a predicate $P : 2^{E^\infty} \rightarrow \{true, false\}$ over systems, where a system is encoded as a set of executions, or traces, $X \subseteq E^\infty$. A policy can also be encoded as a set of valid systems: $P \subseteq 2^{\mathcal{E}^\infty}$.

A mechanism-centric policy P is also a predicate $P : 2^{\mathcal{E}^\infty} \rightarrow \{true, false\}$ over systems [86]. However, as shown in Section 2.1.2, a mechanism-centric trace is a sequence of exchanges rather than a sequence of events. Therefore, a system's set of traces, under the mechanism-centric view encodes the original, possibly invalid, events of the target system, encoded as the first event of every exchange, and the valid events of the system, encoded as the second event in every exchange. A mechanism-centric system is therefore modeled as a set of traces $X \subseteq \mathcal{E}^\infty$, so a policy can also be encoded as a set of valid systems: $P \subseteq 2^{\mathcal{E}^\infty}$.

Another term for policies is hyperproperties [32]. The term "hyperproperties" is used to distinguish between policies that may require relationships to hold between different executions of a system and policies that hold for individual executions of a system, termed "properties". An example hyperproperty that is not a property is a policy stipulating a bound on a system's average response time to requests [32]. To verify whether this hyperproperty holds for a particular system, all of a system's executions must be examined to ensure that the system's average response time is less than or equal to the specified bound. The term "property" is formally defined next in Section 2.1.5.

2.1.5 Properties

Properties are policies that stipulate constraints on individual executions of a system [32]. These type of policies can only atomically reason about a single trace and cannot specify relationships between different traces of a system, like the average response time hyperproperty described previously in Section 2.1.4. Therefore, properties are predicates over individual executions. A property is said to hold for a system if the property holds for each individual execution of the system.

Formally, a system-centric policy P is a system-centric property iff

$$\exists \hat{p} : E^\infty \rightarrow \{true, false\} : \forall X \subseteq E^\infty : P(X) \iff (\forall x \in X : \hat{p}(x)).$$

Because properties are boolean predicates over executions, they can also be formally defined as a set of valid executions. Formally, a system-centric policy P is a property iff

$$\exists \hat{p} \subseteq E^{\infty} : \forall X \subseteq E^{\infty} : (X \in P \iff (\forall x \in X : x \in \hat{p})).$$

Similarly, a mechanism-centric policy P is a mechanism-centric property iff

$$\exists \hat{p} : \mathcal{E}^{\infty} \to \{false, true\} : \forall X \subseteq \mathcal{E}^{\infty} : (P(X) \iff (\forall x \in X : \hat{p}(x))),$$

and a mechanism-centric policy P is a property, described as a set of executions, iff

$$\exists \hat{p} \subseteq \mathcal{E}^{\infty} : \forall X \subseteq \mathcal{E}^{\infty} : (X \in P \iff (\forall x \in X : x \in \hat{p})).$$

It is important to distinguish between policies that are properties and policies that are not properties because it determines what kind of security mechanism can be used to enforce such policy. Dynamic mechanisms that only have access to the current execution can only enforce properties whereas static mechanisms can approximate all possible executions of a program by examining its code and therefore can enforce some policies, or hyperproperties, that are not properties.

Every property can be encoded as an equivalent hyperproperty [32]. Because a property can be encoded as a set of traces $\hat{p} \subseteq E^{\infty}$, or $\hat{p} \subseteq \mathcal{E}^{\infty}$ for mechanism-centric properties, the equivalent hyperproperty can be encoded as the powerset of $\hat{p}$. That is, the equivalent system-centric hyperproperty of $\hat{p}$ is $P = 2^{\hat{p}}$; the mechanism-centric hyperproperty can be defined analogously.

2.1.6 Safety

Policies can be further categorized into safety and liveness policies. A safety policy that is not a property is called a safety hyperproperty, or hypersafety, and a safety policy that is a property is called a safety property [32].

Safety policies are specifications that disallow invalid program behaviors from occurring during a system's execution, meaning that every event or exchange appearing in an execution must be deemed valid. Once an invalid event or exchange occurs, the system is deemed irremediably insecure, meaning that there is no system event or exchange that can appear after the invalid event or exchange that will return the system to a secure state.

In order to formally define hypersafety, we first define two new operators, as they are formally defined in [79]: 1) $\sqsubseteq$ and 2) $\sqsupseteq$. The $\sqsubseteq$ and $\sqsupseteq$ operators are defined similarly to the $\preceq$ and $\succeq$ operators, except where $\preceq$ and $\succeq$ are the prefix and extension operators, respectively, for a single trace, the $\sqsubseteq$ and $\sqsupseteq$ operators are prefix and extension operators, respectively, for a set of traces. Formally, given $X, X' \subseteq E^\infty$ or $X, X' \subseteq \mathcal{E}^\infty$, $X' \sqsubseteq X \implies \forall x' \in X' : \exists x \in X : x' \preceq x$. Therefore, a system-centric policy P is hypersafety iff

$$\forall X \subseteq E^\infty : (X \in P \iff (\forall X' \sqsubseteq X : X' \in P)),$$

and a system-centric policy P is a system-centric safety property $\hat{p}$ iff

$$\forall x \in E^\infty : (x \in \hat{p} \iff (\forall x' \preceq x : x' \in \hat{p})).$$

Similarly, a mechanism-centric policy P is hypersafety iff

$$\forall X \subseteq \mathcal{E}^\infty : (X \in P \iff (\forall X' \sqsubseteq X : X' \in P)),$$

and a mechanism-centric policy P is a mechanism-centric safety property $\hat{p}$ iff

$$\forall x \in \mathcal{E}^\infty : (x \in \hat{p} \iff (\forall x' \preceq x : x' \in \hat{p})).$$

Note that system-centric safety policies require every event in a trace to be valid, whereas mechanism-centric safety policies require every exchange to be valid. Therefore, although the target system may attempt an invalid event, which would be the first event in an exchange, if a security mechanism transforms the invalid event to a valid event, which would be the

second event in the exchange, the overall exchange is considered to be valid, thus illustrating how mechanism-centric policies allow security practitioners to reason about how policies may be enforced in practice.

2.1.7 Liveness

Liveness policies differ from safety policies in that they allow every finite, insecure trace to become secure by extending the trace with a valid extension. A liveness policy that is not a property is called a liveness hyperproperty, or hyperliveness, and a liveness policy that is a property is called a liveness property [32]. Formally, a system-centric policy P is hyperliveness iff

$$\forall X \subseteq E^* : \exists Y \subseteq E^\infty : (Y \sqsupseteq X \wedge Y \in P),$$

and a system-centric policy P is a system-centric liveness property $\hat{p}$ iff

$$\forall x \in E^* : \exists y \in E^\infty : (y \succeq x \wedge y \in \hat{p}).$$

Similarly, a mechanism-centric policy P is hyperliveness iff

$$\forall X \subseteq \mathcal{E}^* : \exists Y \subseteq \mathcal{E}^\infty : (Y \sqsupseteq X \wedge Y \in P),$$

and a mechanism-centric policy P is a mechanism-centric liveness property $\hat{p}$ iff

$$\forall x \in \mathcal{E}^* : \exists y \in \mathcal{E}^\infty : (y \succeq x \wedge y \in \hat{p}).$$

From these definitions, we can see that liveness policies can be satisfied in infinite time. For example, consider the nontermination policy stating that systems should not terminate. Every terminating system that violates this policy can be extended to satisfy the policy by extending every finite-length trace with an infinite-length trace.

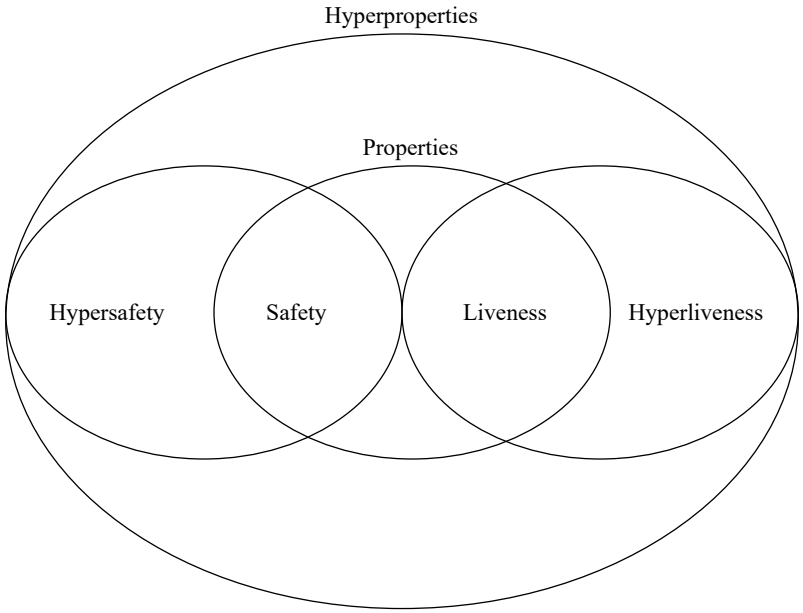

Figure 2.5. Taxonomy of security policies in prior work.

2.1.8 Summary

Figure 2.5 illustrates the taxonomy of security policies discussed in this section. The figure shows that hyperproperties encapsulate all possible policies, while properties are a subclass of the hyperproperties. Therefore, every property is a hyperproperty but every hyperproperty is not a property. Furthermore, as discussed, properties and hyperproperties can be further categorized as safety or liveness properties, or hypersafety or hyperliveness, respectively. Notice that there are policies that are neither safety or liveness, these policies are called nonsafety, nonliveness properties or nonsafety, nonliveness hyperproperties. Prior work has shown that every nonsafety, nonliveness policy is the intersection of a safety and a liveness policy [5, 32].

2.2 Mechanisms

This section reviews existing classes of security mechanisms and shows how formal frameworks have been used to reason about the class of policies enforceable by such mecha-

nisms. This section also reviews work on practical implementations of security mechanisms. This review will show that there does not exist a single framework capable of modeling all security mechanisms and the policies enforceable by such mechanisms.

2.2.1 Static Mechanisms

Static mechanisms enforce policies on whole programs before their execution by analyzing the code of such programs and either accept or reject programs within a finite period of time [46]. If the mechanism accepts the program, then the program is permitted to execute, and if the mechanism rejects the program then the mechanism may output a report detailing the potential policy violations.

Static mechanisms for security are designed to catch potential security vulnerabilities or policy violations before code is put into production. Static mechanisms can be classified as model checkers or static code analyzers. Model checkers can verify properties of models of systems; therefore, these tools do not verify properties of concrete system implementations [6]. Consequently, a challenge for security practitioners that use model checkers to verify properties of models of systems, is verifying whether the model of the system is a correct representation of the actual system. Due to this challenge, this section will only review prior work on static code analyzers that operate on concrete system implementations.

Static code analyzers are typically used during the development lifecycle of a program. Therefore, many static code analyzers operate on a program's source code (e.g., [28, 38, 11]). However, static code analyzers can also operate on untrusted programs for which source code is not available by analyzing programs' intermediate (e.g., [64, 8, 21, 42, 26]) or binary code (e.g., [19, 56, 91]). The enforcement of security policies statically is an attractive option because static code analyzers do not significantly impact a program's runtime performance. However, the problem of determining statically whether an arbitrary program adheres to a nontrivial policy is undecidable [80]. Therefore, engineers of static code analyzers are

forced to implement them to enforce security policies imprecisely. Many static code analyzers enforce policies conservatively (also known as soundly) to ensure that no potential policy violations go undetected. However, enforcing security policies conservatively leads to false positives [28], requiring developers to sift through potentially large amounts of code to determine whether the reports of the analyzer are actually policy violations, which can be burdensome and result in many developers forgoing static analysis and releasing potentially vulnerable code.

Some static code analyzers aim to increase usability and reduce the number of false positives reported by being less sound (e.g., [99, 87]). These tools aim to only report vulnerabilities that can be decided to actually be policy violations. However, such tools often introduce false negatives, which may be considered dangerous as opposed to conservative static code analyzers.

Rice's theorem [80] provides a foundation for the class of policies enforceable by static code analyzers. That is, any nontrivial policy cannot be precisely enforced by a static code analyzer. However, research has shown that approximations of nontrivial policies can be enforced by static code analyzers. For example, rather than enforcing a policy stipulating that all programs must terminate, a static code analyzer can determine whether a program will terminate within some finite, specified number n of computational steps by simulating the program for n computational steps [46]. The static code analyzer will therefore reject all programs that it can simulate for $n + 1$ steps.

To formally define the class of policies enforceable by static code analyzers, Hamlen et al. [46] model security mechanisms as Turing machines and untrusted programs as program machines, which are deterministic Turing machines that operate over 3 infinite-length tapes. The class of policies enforceable by static code analyzers is formally defined in [46] with respect to classes from complexity theory; [46] applies Rice's theorem to state that only the class of recursively decidable policies of program machines are enforceable by static

code analyzers. For every recursively decidable policy P there exists a total, computable function f that can take as input a Turing machine M and decide within a finite period of time whether M satisfies P. If for any policy P, f does not exist then P is not statically enforceable.

Other static enforcement models have employed a functional model of computation in a variant of the lambda calculus (e.g., [98, 15]) or a security-typed language (e.g., [85, 90, 81, 84, 94, 13, 97, 50, 98, 3, 2, 95]) to express the possible terms, or expressions, of a programming language (syntax) and how such terms should be evaluated (semantics). After defining a language's syntax and semantics, typing rules can be formally defined by a set of inference rules, which allows one to determine statically whether every terminating expression of a program will evaluate to a well-defined value with a well-defined type [74]. The static security mechanism in this case is modeled by the rules of the type system.

2.2.2 Dynamic Mechanisms

Dynamic mechanisms enforce policies by monitoring programs during their execution and intervening when necessary. Due to available runtime information, dynamic mechanisms can enforce some policies more precisely than static mechanisms; however, the precision at which policies can be enforced may depend on the capabilities of the dynamic security mechanism.

To understand the capabilities of various dynamic mechanisms, they are generally modeled as security automata consisting of countably many states that operate over a countable alphabet and transition between states via a well-defined transition function. Formally, a dynamic mechanism can be modeled as an automaton $M = (Q, Q_0, E, \delta)$ such that

1. Q is a countable set of states,

2. $Q_0 \subseteq Q$ is a countable set of initial states,

3. E is event set over which M operates, and

4. δ is either a deterministic or nondeterministic transition function that has generally been defined in one of four ways:

   (a) $\delta : Q \times E \to Q$,

   (b) $\delta : Q \times E \to 2^Q$,

   (c) $\delta : Q \times E \to Q \times E$, or

   (d) $\delta : Q \times E \to 2^{Q \times E}$.

Mechanisms whose transition functions are $\delta : Q \times E \to Q$ or $\delta : Q \times E \to 2^Q$ are called deterministic or nondeterministic execution recognizers, respectively. Mechanisms whose transition functions are $\delta : Q \times E \to Q \times E$ or $\delta : Q \times E \to 2^{Q \times E}$ are called deterministic or nondeterministic execution transformers, respectively [61, 17].

Execution recognizers enforce policies by recognizing invalid executions and halting the program, or target system, under consideration. Schneider [86] introduced such recognizers, called Execution Monitors (EMs), as a class of dynamic security mechanisms that can only monitor target system events as they are attempted by the target system. That is, EMs have no knowledge of future events that might be exhibited by the target system. Furthermore, EMs can only respond to policy violations by halting the target system. Schneider showed that under this definition, EMs can only enforce a subclass of the safety properties. EMs cannot enforce all safety properties because mechanisms may not have sufficient power to prevent the violation of some safety properties. For example, EMs cannot stop the passage of time. Later work considers such uncontrollable, also called unsupressable events, as clock ticks [16, 33, 75].

In particular, Basin et al. [16] extended Schneider's work by distinguishing between controllable and uncontrollable events. Controllable events are events that security mecha-

nisms can control whereas uncontrollable events can only be observed by mechanisms but not controlled. Safety properties that depend on uncontrollable events (e.g., clock ticks) cannot be precisely enforced by EMs.

Execution transformers have the ability to not only recognize an invalid execution but to transform an invalid execution into a valid one. Ligatti et al. [61] introduced such transformers as Edit Automata, a class of dynamic security mechanisms that can monitor events as they are attempted by the target system and have the ability to transform system executions by inserting or suppressing events as necessary. With these added capabilities, it is important to constrain the behaviors of security mechanisms to prevent them from transforming executions in arbitrary ways. Ligatti et al. [61] introduced three definitions that describe how an edit automaton can enforce policies: 1) conservative enforcement, 2) precise enforcement, and 3) effective enforcement.

An edit automaton conservatively enforces a security policy if the sequence of mechanism outputs is in the set of executions allowed by the policy. This definition allows mechanisms to completely ignore target system inputs and always output executions that will satisfy the policy. An edit automaton precisely enforces a security policy if the automaton conservatively enforces the policy and preserves the syntax of target system executions that already obey the policy in question. That is, if a target system's execution already satisfies the policy, then the mechanism must output a syntactically equal execution. An edit automaton effectively enforces a security policy if it: 1) conservatively enforces a security policy, and 2) preserves the semantics of valid executions. That is, if a target system's execution already satisfies the policy then the mechanism must output a semantically equivalent execution, that may not necessarily be syntactically equal.

Given these definitions of enforcement, [61] shows that edit automata can precisely enforce safety and some nonsafety properties, and can effectively enforce any property, depending on the definition of semantic equivalence. This result is particularly interesting

for liveness properties because such properties require a mechanism to have knowledge of possible future events of an execution. While edit automata do not have direct knowledge about the future of systems' executions, they do have the ability to buffer events for arbitrary amounts of time. As a result, edit automata can observe arbitrarily long sequences of system events while simultaneously buffering them and delaying their execution, once an event satisfying a liveness property occurs, then the automaton can release the sequence of buffered events to execute. However, this capability may be considered unrealistic in many situations. Alternatively, an edit automaton can prematurely insert an event, or sequence of events, that satisfy a liveness property. Then, if the target system exhibits the event, or sequence of events, satisfying the liveness property, the mechanism can quietly suppress the event(s) because it already output the event(s) satisfying the liveness property.

The formal framework presented in [61] models dynamic mechanisms as execution transformers but model policies in a system-centric way, which requires the hardcoded definitions of conservative, precise, and effective enforcement because system traces are not expressive enough to capture such definitions as a part of the system trace.

Dolzhenko et al. [33] introduced another class of execution transformers as Mandatory Results Automata (MRAs). These mechanisms have the ability to recognize and transform not only events exhibited by target programs but also results that might be returned to target programs by the underlying executing system. MRAs can only process a single event at a time. That is, for every event input to the mechanism, the mechanism must produce an output before receiving another input event. A novel feature of MRAs is that they are able to enforce, what [33] calls, result sanitization policies. Such policies may require a mechanism to sanitize results returned by an underlying executing system before the results can be successfully returned to the target system. For example, a policy may require secret contents of a directory to be hidden from users when they execute the `ls` command. When

the results for the contents of the directory are returned, a mechanism might filter out secret files before displaying the contents of the directory.

Dolzhenko et al. [33] gives three definitions for how MRAs may enforce security policies. They can enforce properties soundly, completely, and precisely. In this framework, properties are encoded in a mechanism-centric way; therefore, properties can take into consideration the effects of security mechanisms on system events. An MRA soundly enforces a property $\hat{p}$ if the set of executions of the MRA is a subset of the executions describing the property $\hat{p}$. That is, the MRA only produces executions that satisfy $\hat{p}$. An MRA completely enforces a policy if the set of executions of the MRA is a superset of the executions of the property $\hat{p}$. That is, every valid execution described by the property $\hat{p}$ is an execution produced by the MRA. Lastly, an MRA precisely enforces a security property if the set of executions describing the MRA equals the set of executions describing the property $\hat{p}$. It is shown that deterministic MRAs precisely enforce a proper subset of the mechanism-centric safety properties while nondeterministic MRAs precisely enforce a proper superset of the mechanism-centric safety properties. Determinsitic MRAs can also enforce some mechanism-centric nonsafety properties by assuming that the underlying executing system will always return events for particular system events; [33] argues that such an assumption is often reasonable and can be seen in practical systems such as CPUs and operating systems that always return a result for particular system events.

Stream Monitoring Automata (SMAs) were introduced in [72]. These mechanisms are execution transformers that operate over infinite sequences of events. The novel feature of these mechanisms is that they are not input enabled. Previous models of mechanisms assumed that mechanisms only output events when provided an input event; [72] gives several practical examples where it is desirable for mechanisms to output events even in the absence of input events. It is shown that, with the capability to output events in the absence of input events, SMAs can enforce some nonsafety properties. The framework used

to reason about policies enforceable by SMAs was a mechanism-centric framework. Due to the expressiveness of the framework, [72] shows that constraints placed on SMAs can be encoded as metapolicies. Metapolicies are defined as policies that place constraints on other policies. A popular constraint of security mechanisms is that they enforce policies transparently. Transparency means that if a target system's execution already satisfies the policy in question, then the mechanism should not modify the execution in any observable way. Transparency can be encoded as a metapolicy by requiring the input and output events of exchanges to be equal when the input event is considered to be a valid event.

Nonuniform runtime mechanisms were introduced in [62]. Nonuniform runtime mechanisms have auxiliary knowledge about programs' executions, such as the results of a static code analyzer that guarantees whether certain program events will occur. Formal characterizations of the class of policies enforceable by nonuniform runtime mechanisms with various capabilities were presented in [27, 65, 76]. It was shown that nonuniform runtime mechanisms are capable of enforcing some nonsafety properties, which uniform runtime mechanisms—mechanisms that only monitor the current execution and have no a priori knowledge about the program—with the same capabilities cannot enforce. Although nonuniform runtime mechanisms aim to amplify the enforcement power of runtime mechanisms by giving the mechanism access to the results of a static code analyzer, this model is not capable of unifying the two enforcement approaches because nonuniform runtime mechanisms do not have direct access to a program's code, whereas static code analyzers do.

Models of runtime mechanisms for black-box reactive programs were introduced in [70]. These mechanisms are execution transformers that have the ability to run additional, isolated copies of a program to examine the difference in program output when given different inputs. However, it is assumed that mechanisms do not have access to programs' source or machine code. With the capability to examine how programs respond to different inputs, these mechanisms were shown to be able to enforce some hypersafety policies. Pre-

vious work on runtime mechanisms only considered them to be able to enforce properties, since they could only examine the current execution.

Hamlen et al. [46] showed that any statically enforceable policy is enforceable by a runtime mechanism if the runtime mechanism can perform code analysis immediately after the program is loaded, illustrating that hyperproperties are indeed enforceable at runtime. However, [46] did not explore the implications of this claim or whether runtime mechanisms, capable of performing code analysis, can improve the precision at which some statically undecidable policies may be enforced.

2.2.3 Hybrid Mechanisms

Hybrid mechanisms enforce policies by combining static code analysis and runtime monitoring (e.g., [82, 96, 58, 59, 88]). These mechanisms enable some policies to be enforced that are difficult to enforce by just a static or dynamic mechanism alone.

Many hybrid security mechanisms proposed in the literature are domain specific, targeting either the web-application or information-flow security domains. For example, [82] presents a mechanism capable of enforcing certain information-flow security policies that cannot be enforced by just a static or dynamic mechanism alone, but can be enforced by combining static code analysis and runtime monitoring.

Russo and Sabelfield [83] show that information-flow policies can be enforced both soundly and permissively with hybrid mechanisms. Because statically determining whether an arbitrary program satisfies a policy, in general, is undecidable, detecting information-flow policy violations statically is not permissive. A sound static analyzer may conservatively reject programs that do not violate the policy. A purely dynamic mechanism, however, may not soundly enforce an information-flow policy, if it is unaware of other possible program executions, but it is more permissive than static analysis due to available runtime informa-

tion. Russo and Sabelfied show that the best of both worlds can be achieved by combining static code analysis and runtime monitoring.

Vogt et al. [96] presented a hybrid mechanism that is able to prevent attackers from obtaining sensitive user information through cross-site scripting attacks. The mechanism performs runtime monitoring to ensure that JavaScript programs only send sensitive information to servers from which they originated. Static analysis is used to detect implicit information flows that may not be able to be detected by the runtime mechanism.

AMNESIA [44] is another hybrid mechanism designed to detect SQL injection attacks. The mechanism uses static code analysis to obtain a model of valid SQL queries that an application is allowed to generate. When an application executes, AMNESIA employs a dynamic mechanism to analyze concrete, dynamically generated SQL queries. These queries are checked against the statically generated model of valid queries. If the query is valid, then the query is allowed to proceed; if the query is found to be invalid, then the query is rejected and reported to security administrators. Several other, similar, hybrid mechanisms have been proposed for ensuring the security of web applications (e.g., [60, 4, 12, 30]).

The majority of hybrid mechanisms proposed in the literature have been domain-specific mechanisms designed to enforce specific policies. Consequently, frameworks for such domain-specific mechanisms are not suitable for reasoning about which policies are enforceable by hybrid mechanisms in general.

However, [46] presented a class of policies enforceable by a particular type of hybrid mechanism called a program rewriter. A program rewriter is a mechanism that is able to modify a program prior to its execution to ensure that it satisfies a particular policy. Program rewriters can be seen as purely static mechanisms since they modify programs prior to their execution; however, they can also be viewed as hybrids when they inline runtime checks to enforce policy-specific constraints at runtime for statically undecidable policies. Program rewriters that inline runtime checks for statically undecidable policies implement Inlined

Reference Monitors (IRMs) [37]. IRMs are popular runtime mechanisms that are embedded within a program's code. IRMs provide the capability for enforcing application-specific policies. Furthermore, IRMs may improve the performance of enforcing policies at runtime by eliminating context switches between the untrusted code and the security mechanism, because the mechanism is embedded within the program's code [47].

Hamlen et al. [46] characterized the class of policies enforceable by program rewriters as the $RW_{\approx}-enforceable$ policies. A policy P is deemed $RW_{\approx}-enforceable$ if three rules can be satisifed: 1) there exists a total, computable rewriting function that inputs an untrusted program, say X, and outputs a program X', 2) the output program X' satisfies P (i.e., $P(X')$), and 3) if the input program X already satisfies policy P then the output program X' is equivalent to the input program (i.e., $P(X) \Rightarrow X \equiv X'$). The $RW_{\approx}-enforceable$ policies includes all statically enforceable policies and all of the policies enforceable by EMs. Although [46] characterizes the class of policies enforceable by program rewriting as the $RW_{\approx}-enforceable$ policies, it is stated that the exact class of policies enforceable by program rewriters with respect to known classes from complexity theory could not be determined, and might not exist, as no known class of the arithmetic hierarchy is equivalent to the class of $RW_{\approx}-enforceable$ policies.

2.2.4 Aspect-Oriented Policy-Specification Languages

Policy-specification languages are designed to aid policy writers in correctly specifying security policies. Policy-specification languages are important because specifying policies correctly can be an error prone and tedious process. Such languages can provide guarantees to the policy writer that the specification actually implements the policy that the policy writer intended through the language's type system and formal verification techniques [45]. This section reviews aspect-oriented policy-specification languages designed for policy enforcement by runtime monitoring. We do not review policy-specification languages designed

for policy enforcement by static code analysis due to the undecidability of statically determining whether an arbitrary program satisfies a policy; therefore, such languages are not powerful enough to synthesize security mechanisms capable of enforcing nontrivial security policies.

A common way to specify security policies is by implementing them as aspects [53]. Aspects allow policy writers to implement security policies in a modular fashion so as to make them easier to reason about and verify the correctness of their implementations. Aspect-oriented programming is a programming paradigm that modularizes cross-cutting concerns [55]. A cross-cutting concern is a concern that is spread throughout several modules of a system. For example, consider a security policy that places stipulations on which files can be accessed by untrusted programs. A program may execute several file operations, spread throughout several modules of its implementation. To ensure that the policy is not violated, every file operation must be verified to satisfy the security policy. Implementing the policy for every single file operation can be tedious and error prone; however, aspect-oriented programming languages provide a way for security policies to be specified in a centralized, modular way.

Security Policy XML (SPoX) [45] is a declarative policy-specification language designed for enforcement by Inlined Reference Monitors (IRMs) that implement execution monitors [86] on Java bytecode applications. It is an aspect-oriented language based on a system-centric view of security policies. That is, the language is capable of specifying what the policy is but not how the policy should be enforced; [45] argues that imperative code fragments that specify how a policy should be enforced are difficult to reason about and can be error prone.

Polymer [18] is an imperative policy-specification language designed for enforcement by IRMs that implement edit automata [61] on Java bytecode applications. Polymer allows

policy writers to specify how policy violations should be handled through the *query* method construct that returns suggestions on how to handle security-relevant events.

Policy Enforcement Toolkit (PoET) and Policy Specification Language (PSLang) [37] are a program-rewriting system and a policy-specification language for implementing security policies for enforcement by IRMs on Java bytecode applications. They are used to implement Java's stack inspection security policy. Naccio [39] is another system designed for specifying safety properties enforceable by IRMs on Java bytecode applications and Win32 executables.

ConScript [67] is a system designed for enforcing security policies via IRMs on JavaScript code in the browser. ConScript policies can be generated automatically through static and dynamic code analysis, which eliminates the need for policies to be developed manually by policy writers.

Several other aspect-oriented policy-specification languages exist (e.g., [10, 49, 9, 48, 51]); however, such languages synthesize runtime monitors that have only been shown to be capable of monitoring security-relevant program events as they are attempted by target programs. Policies that are not properties and liveness properties require mechanisms to be aware of other possible executions, or the possible future events of an execution, respectively. However, as far as we are aware, there does not exist a general policy-specification language capable of synthesizing runtime mechanisms that can enforce all kinds of policies (i.e., hyperproperties, properties, safety, and liveness).

2.2.5 Summary

This section has shown that there exist several models of security mechanisms, with various capabilities, that help us to understand the class of policies enforceable by such mechanisms. Such models are important for understanding what types of policies we can expect security mechanisms to enforce in practice. However, as far as we are aware, no model proposed in the literature has been shown to capture all kinds of security mechanisms, which

makes it difficult to answer the question of whether additional sorts of mechanisms, capable of enforcing an even larger class of security policies, exist.

Hybrid mechanisms have been shown to enforce policies that are difficult to enforce by just a static or dynamic mechanism alone (e.g., [82]), but many hybrid mechanisms proposed in the literature are domain specific, making it difficult to understand the practical limitations of hybrid mechanisms in general. Program rewriters were shown to be a powerful class of hybrid security mechanisms capable of enforcing all of the statically enforceable policies and all of the policies enforceable by execution monitors; however, the exact class of policies enforceable by such mechanisms with respect to known classes from complexity theory is unknown. A general model capable of capturing arbitrary classes of security mechanisms might be a first step to determining whether additional classes of security mechanisms exist and further refining the class of policies enforceable by hybrid mechanisms.

We conclude from prior work that runtime mechanisms can be viewed as generalizations of static mechanisms when runtime mechanisms can access a program's code. This conclusion provides a basis for a unified approach to security policy enforcement.

Chapter 3: A Unified Approach

Security mechanisms are typically categorized as static, dynamic, or hybrid mechanisms. This classification is based on the time at which such mechanisms enforce policies on untrusted programs. Static mechanisms enforce policies on programs before their execution, dynamic mechanisms enforce policies on programs during their execution, and hybrid mechanisms enforce policies on programs by using a combination of static and dynamic enforcement.

While different security mechanisms may differ in the time at which they enforce security policies— relative to the current execution status of a program—all security mechanisms enforce policies on some granularity of a program event before its execution, irrespective of the fact that the program as a whole may or may not be currently executing. Static mechanisms enforce policies on whole programs before their execution, whereas dynamic mechanisms typically enforce policies on program instructions, or statements, before their execution. Rather than classifying mechanisms based on when they operate, this dissertation classifies mechanisms based on the granularity of program code that they monitor. This shift in perspective allows us to cast existing mechanisms into a single framework where all mechanisms can be encoded as runtime mechanisms that operate at one or more levels of program code granularity. This chapter presents a unified framework for reasoning about all security mechanisms and defines levels of granularities at which mechanisms may operate.

3.1 Granular Program Events and Traces

The framework presented in this chapter is derived from the mechanism-centric, trace-based framework presented in [33, 79] and in Chapter 2. The main difference is that previous work considered the event set E to be a set of atomic program events that, when executed, cause a system to take a computational step, which we call fine-grained events. We extend prior work by allowing set E to contain fine-, medium-, and coarse-grained events. Let E_c denote the set of all possible coarse-grained program events, with metavariable e_c ranging over the individual elements of E_c; E_m denotes the set of all possible medium-grained program events, with metavariable e_m ranging over the individual elements of E_m; E_f denotes the set of all possible fine-grained program events, with metavariable e_f ranging over the individual elements of E_f. Therefore, $E = E_c \cup E_m \cup E_f$ and is ranged over by metavariable e.

Following [33, 79], the event set E determines the set $\mathcal{E}$ of possible exchanges, except in this dissertation $\mathcal{E}_f = \{\langle e_f, e' \rangle \mid e_f \in E_f\}$ denotes the set of fine-grained exchanges, $\mathcal{E}_m = \{\langle e_m, e' \rangle \mid e_m \in E_m\}$ denotes the set of medium-grained exchanges, and $\mathcal{E}_c = \{\langle e_c, e' \rangle \mid e_c \in E_c\}$ denotes the set of coarse-grained exchanges. Note that there are no stipulations placed on the granularity of e' in each exchange. These definitions are general in that they do not limit the granularity of the event that can actually be executed. The set of all exchanges is denoted by $\mathcal{E} = \mathcal{E}_f \cup \mathcal{E}_m \cup \mathcal{E}_c$.

The distinction between coarse-, medium-, and fine-grained exchanges allows us to encode hybrid traces within this framework. A trace is hybrid iff it contains at least two exchanges that are of different granularities. Such traces capture the functionality of hybrid mechanisms that combine static and dynamic enforcement by first statically analyzing a whole program and then dynamically enforcing policy-specific constraints on fine-grained program instructions.

Let $\mathcal{E}_f^*$ denote the set of all finite-length, fine-grained traces, $\mathcal{E}_f^\omega$ denote the set of infinite-length, fine-grained traces, and $\mathcal{E}_f^\infty = \mathcal{E}_f^* \cup \mathcal{E}_f^\omega$ denote the set of all possible fine-grained traces. Let $\mathcal{E}_m^*$ denote the set of all finite-length, medium-grained traces, $\mathcal{E}_m^\omega$ denote the set of all infinite-length, medium-grained traces, and $\mathcal{E}_m^\infty = \mathcal{E}_m^* \cup \mathcal{E}_m^\omega$ denote the set of all possible medium-grained traces. Let $\mathcal{E}_c^*$ denote the set of all finite-length, coarse-grained traces, $\mathcal{E}_c^\omega$ denote the set of all infinite-length, coarse-grained traces, and $\mathcal{E}_c^\infty = \mathcal{E}_c^* \cup \mathcal{E}_c^\omega$ denote the set of all possible coarse-grained traces. Let $\mathcal{E}_h^*$ denote the set of all possible finite-length hybrid traces, $\mathcal{E}_h^\omega$ denote the set of all possible infinite-length traces, and $\mathcal{E}_h^\infty = \mathcal{E}_h^* \cup \mathcal{E}_h^\omega$ denote the set of all possible hybrid traces.

The set of all possible finite-length traces is denoted by $\mathcal{E}^* = \mathcal{E}_f^* \cup \mathcal{E}_m^* \cup \mathcal{E}_c^* \cup \mathcal{E}_h^*$, the set of all infinite-length traces is denoted by $\mathcal{E}^\omega = \mathcal{E}_f^\omega \cup \mathcal{E}_m^\omega \cup \mathcal{E}_c^\omega \cup \mathcal{E}_h^\omega$, and the set of all possible traces is denoted by $\mathcal{E}^\infty = \mathcal{E}_f^\infty \cup \mathcal{E}_m^\infty \cup \mathcal{E}_c^\infty \cup \mathcal{E}_h^\infty$. It is important to distinguish between the granularity of program events and traces to ensure that we are able to capture the granularity of program code that existing classes of static, dynamic, and hybrid mechanisms monitor.

3.2 Mechanisms

All mechanisms presented in this chapter are runtime mechanisms that enforce policies by intercepting security-relevant events just before they are about to execute. To capture realistic behaviors of security mechanisms, we model them as automata that can permit, deny [86], modify [46], suppress, or insert [61] program events based on the rules of a policy. These capabilities capture the capabilities of existing mechanisms presented in Chapter 2.

Definition 1. *A security mechanism is an automaton $M = (Q, Q_0, I, O, \delta)$, where*

- *Q is a countable set of automaton states,*
- *$Q_0 \subseteq Q$ is a countable set of initial automaton states,*
- *I is the set of possible inputs to the mechanism,*

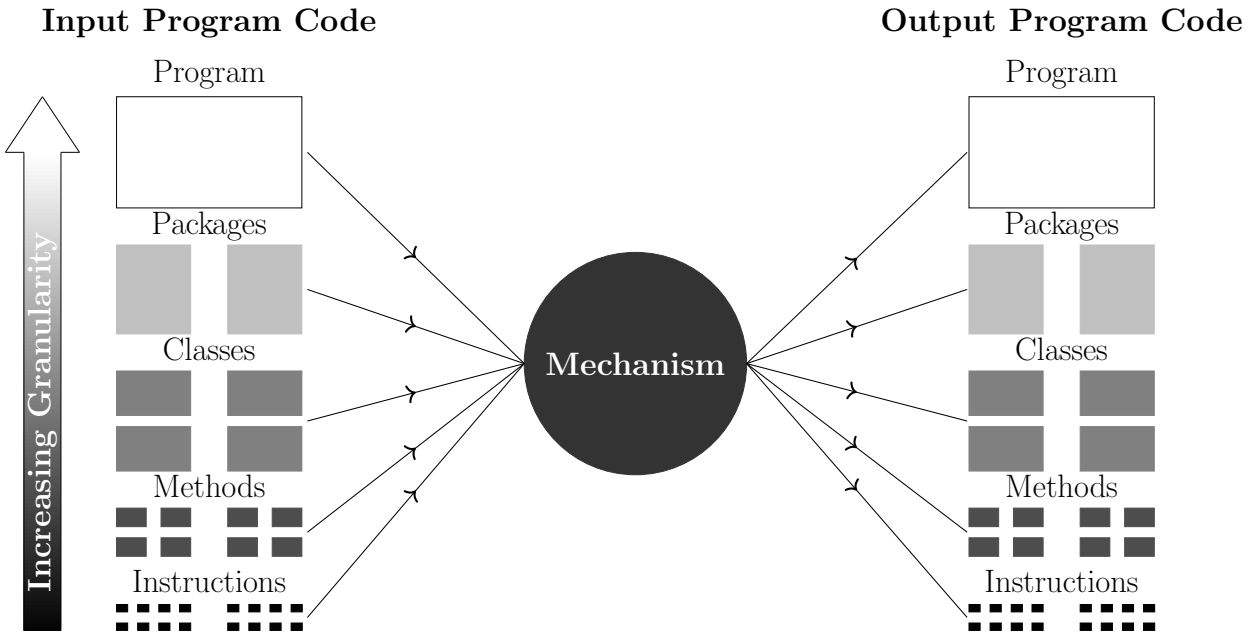

Figure 3.1. A general security mechanism.

- *O is the set of possible outputs from the mechanism, and*
- *δ is a deterministic or nondeterministic transition function. A deterministic δ is a function $\delta : Q \times I \to Q \times O$, and a nondeterministic δ is a function $\delta : Q \times I \to 2^{Q \times O}$*

It is important to distinguish between deterministic and nondeterministic mechanisms because there often exist several possible output events for a single input event. For example, the mechanism can output the input event verbatim, an event to halt the system, or replace the input event with an entirely new event. This definition is general in that it does not limit the extent to which mechanisms can transform input events. However, to constrain mechanisms' behaviors one can define a deterministic transition function.

An input $i \in I$ to or an output $o \in O$ from a mechanism can be any well-defined program construct. For example, Figure 3.1 illustrates how an example Java program is composed of different granular program constructs. At the coarsest granularity are whole programs. A whole program can be decomposed into packages, packages can be decomposed

into classes, classes can be decomposed into methods, and methods can be decomposed into fine-grained program instructions.

3.3 Fine-Grained Policy Enforcement

A *fine-grained policy* ranges over fine-grained traces.

Definition 2. *A policy P_f is fine grained iff*

$$P_f \subseteq 2^{\mathcal{E}_f^\infty}.$$

Definition 3. *A property $\hat{p_f}$ is fine grained iff*

$$\hat{p_f} \subseteq \mathcal{E}_f^\infty.$$

For the rest of this dissertation P_f and $\hat{p_f}$ will denote that the policy, respectively property, under consideration is fine grained. As stated in Section 3.1, only the first event in every exchange is required to be a fine-grained event. In every mechanism-centric trace, the first event in every exchange is an event attempted by a target system; therefore, the first event in every exchange is an event that could be monitored by, or input to, a security mechanism.

Fine-grained events include all atomic program instructions including values contained therein. Instructions might be low-level assembly or micro-code instructions, or high-level language program statements (e.g., method invocations or assignment statements). Values can be primitive or reference types or a collection of values stored in a data structure (e.g., integers, objects, or arrays of values).

Previous work encoded system-centric executions as sequences of atomic events, where each event is an event executed by the system that causes the system to take a computational step. Mechanism-centric executions were encoded as sequences of exchanges, where both events in the exchange are fine-grained events. The first event is an event attempted by

the target system that might cause it to take a computational step, and the second event is the event actually executed by the system, causing it to actually take a computational step. Previous work has shown that all policies can be encoded as fine-grained policies. However, not all policies are enforceable at such a fine-grained level.

In this framework, a runtime mechanism operates at a fine-grained level if it monitors fine-grained events as they are attempted by the target program. Policies enforceable at this level do not require mechanisms to access a program's code. The language monitored by the mechanism determines the granularity for certain types of program events. A single program statement at the source-code level may consist of several instructions at the machine-code level. Policies enforceable at a fine-grained level, at the source-code level, may not be enforceable at a fine-grained level at the machine-code level. For example, consider again the "no `FileWriter.write()` after `FileReader.read()`" policy presented in Chapter 2. `FileReader.read()` and `FileWriter.write()` are Java method calls. A runtime mechanism can enforce this policy at a fine-grained level at either the source or bytecode level. At the source-code level, a mechanism can directly monitor `FileWriter.write()` and `FileReader.read()` method invocations. At the bytecode level, a mechanism can monitor java bytecode `invokevirtual` instructions for the `FileWriter.write()` and `FileReader.read()` method calls. If a program attempts to execute `FileWriter.write()` after executing `FileReader.read()`, one possible way the mechanism can respond to the attempted policy violation is by outputting `System.exit(1)` to halt the target program. The mechanism in this case only monitors the invocations of the `read()` and `write()` methods and does not analyze the code body of these methods. A mechanism enforcing such a policy in this way, by monitoring individual program statements, operates at a fine-grained level.

Now consider a runtime mechanism attempting to enforce the "no `FileWriter.write()` after `FileReader.read()`" policy at the machine-code level. Such a mechanism may not be able to enforce this policy at a fine-grained level for at least two rea-

sons: 1) the language-specific abstractions `Filewriter.write()` and `FileReader.read()` may not exist at the machine-code level, and 2) an optimizing compiler may have replaced all function call and return statements with the code bodies of functions (i.e., inlining), thus eliminating a monitor's ability to monitor function calls.

Fine-grained policy enforcement is beneficial when satisfaction of the policy depends on runtime values. For example, consider again the "no `FileWriter.write()` after `FileReader.read()`" policy except programs can now write to files after a file read if and only if the name of the file being written to is `log.txt`. Assuming the file name is only available at runtime, the policy can be enforced at a fine-grained level because a mechanism only needs to examine the value of the object instance invoking the `write` method (i.e., the name of the file bound to the `FileWriter` object).

Existing mechanisms that operate at this level of granularity include all runtime mechanisms that enforce policies by only monitoring the current fine-grained program execution and do not access a program's code. Such mechanisms include execution monitors [86], edit automata [61], mandatory results automata [33], and stream monitoring automata [72].

Mechanisms operating at this level cannot, in general, precisely enforce hyperproperties [32] that may require relationships to hold between different executions of a program. This limitation of fine-grained mechanisms exists because such mechanisms have no knowledge of other possible program executions when only monitoring program events as they are attempted during the current execution.

3.4 Medium-Grained Policy Enforcement

A *medium-grained policy* ranges over medium-grained traces.

Definition 4. *A policy P_m is medium-grained iff*

$$P_m \subseteq 2^{\mathcal{E}_m^\infty}.$$

Definition 5. *A property $\hat{p_m}$ is medium grained iff*

$$\hat{p_m} \subseteq \mathcal{E}_m^\infty.$$

Henceforth, P_m and $\hat{p_m}$ will denote that a policy, respectively property, is medium grained. Intuitively, medium-grained events exclusively capture all well-defined, modular language constructs in between fine-grained program statements and whole programs. This level contains the broadest range of events as different languages contain different constructs. For example, some medium-grained constructs from the Java programming language include packages, classes, and methods. More general medium-grained constructs include loops and conditional statements.

When encoding medium-grained traces, the entire medium-grained construct appears on the trace because the code body of the construct is considered to be security relevant. For example, the "no `FileWriter.write()` after `FileReader.read()` policy" was shown in Section 3.3 to be enforceable at a fine-grained level because only the invocations of the `read` and `write` methods were considered security relevant; but suppose, additionally, that the policy placed constraints on events executed within `FileWriter.write()` and `FileReader.read()`. For example, the output stream, respectively the input stream, should be closed before the end of the method's execution. Ideally, a mechanism should be able to determine whether the code body of the `FileWriter.write()` and `FileReader.read()` methods satisfy the policy before allowing each method to execute. A mechanism attempting to enforce such a policy at a fine-grained level may have to let the `FileWriter.write()` and `FileReader.read()` methods execute partially until it observes an attempted policy violation. That is, attempting to return before closing the output (input) stream, at which point the mechanism might either attempt to correct the execution by inserting an event to close the output (input) stream, if it has the capability to do so (e.g., [61]), or attempt to roll back the effects of the method's partial execution (e.g., [22]). An alternative way to enforce such a policy would be for the

mechanism to analyze the code body of the `FileWriter.write()` and `FileReader.read()` methods before their execution to ensure that each corresponding method conforms to the policy at hand.

In practice, a mechanism enforcing a policy at this level of granularity must be able to intercept execution at a well-defined point in a program's execution to ensure that the mechanism can adequately analyze the medium-grained event before its execution. For example, consider the application-specific policy stating "all resources acquired within an application-defined method must be released before the end of the method's execution" [99, 66]. This policy can be enforced at a medium-grained level in the following way:

1. The mechanism can intercept execution at the invocation of every application-defined method.

2. With access to the program's code, the mechanism can perform code analysis on the body of the intercepted method.

3. If the mechanism can determine that every resource acquired within the method is released before the end of the method's execution, then the mechanism can allow the method to proceed to execute. If the mechanism determines that the program will potentially violate the policy, then the mechanism can enforce the policy in a way described by the policy writer (e.g., halting the program).

Such a policy is typically enforced by a static code analyzer because it can examine all possible execution paths and determine if any of the paths may lead to a policy violation. However, a sound static code analyzer will reject programs that contain both secure and insecure paths. The benefit of enforcing such a policy at runtime over enforcing it statically is to more accurately determine the potential execution paths of a program. For example, consider the following Java method:

Listing 3.1. An example Java method with an unreleased file resource

```
1  public void writeSubString(int index, String s){
2      FileWriter writer = null;
3      try{
4          writer = new FileWriter(new File("substrings.txt"));
5          if(index < 10){
6              String newS = s.substring(0,index);
7              writer.write(newS);
8              writer.close();
9          } else{
10             String newS = s.substring(index);
11             writer.write(newS);
12         }
13     } catch(IOException e){
14         if(writer != null){
15             writer.close();
16         }
17     }
18 }
```

This method contains both secure and insecure execution paths, with respect to the policy stating "resources acquired within an application-defined method must be released before the end of the method's execution". The secure execution path may be taken when the runtime argument for `index` is less than 10. The insecure execution path may be taken when the runtime argument for `index` is greater than or equal to 10. A sound static analyzer will take into account all possible executions of this method and report that the `FileWriter`

object may not be closed. Assuming that the source code cannot be simply edited, because it's not available, the policy may be enforced more precisely at runtime if the mechanism can determine which branch may be taken by examining the program's intermediate bytecode, given the necessary runtime values.

Suppose `writeSubString(5,"Hello, World!")` is attempted by the program. Knowledge of the runtime values allows a medium-grained mechanism to determine that the `if` branch may be taken and, if so, the `FileWriter` object will be closed; therefore, the method can proceed to execute. Now assuming `writeSubString(10, "Hello, World!")` is attempted, the mechanism can then determine that the `else` branch may be taken and that the `FileWriter` object may not be closed. The mechanism can still allow the method to proceed but, if needed, insert an event to close the `FileWriter` object just before the method returns.

Medium-grained policies allow policy writers to take advantage of the modular abstractions of programming languages. Rather than encoding policies as sets of sequences of fine-grained events, or exchanges, policy writers can reason about some policies more abstractly by encoding them as sequences of modular events or exchanges. For example, the "resources acquired within a method must be released before the end of the method's execution" property, notated RAA henceforth, places constraints over language-specific abstractions (i.e., methods). RAA can be encoded at a medium-grained level as the set of all traces containing sequences of valid methods. Let $R \subseteq E_m$ be the set of all valid methods with respect to RAA. One way to encode RAA as a medium-grained property would be

$$RAA_m = \{x \in \mathcal{E}_m^\infty \mid \forall \langle e, e' \rangle \in x : (e \in R \Rightarrow e = e') \wedge e' \in R\}.$$

This policy consists of traces containing sequences of medium-grained exchanges. It states that, for every exchange in a trace, if the input event is an element of R (i.e., the input event satisfies the constraint that all resources acquired within a method are released

before the end of the method's execution) then the input event is allowed to execute (i.e., the output event equals the input event); if the input event is not an element of R (i.e., the input event does not satisfy the policy constraint) then the input event is not allowed to execute and an event satisfying the constraint is executed instead (i.e., an event that is an element of R).

In practice, a runtime monitor may enforce RAA_m by intercepting program execution at all method invocations and perform code analysis on the corresponding method body. If the monitor can determine whether the method is an element of R, then the monitor can allow the method to proceed uninterrupted. If the monitor determines that the method is not an element of R, or if the monitor cannot decide whether the method is an element of R, perhaps due to control flow that depends on runtime information that cannot be determined at the time of method invocation, then the monitor can output a default method that satisfies R.

As discussed in Section 3.3, fine-grained hyperproperties requiring relationships to hold between different executions of a system cannot, in general, be precisely enforced by mechanisms that operate at a fine-grained level, because they are unaware of other possible system executions. On the contrary, some hyperproperties, requiring relationships to hold between executions of a medium-grained event, can be enforced by mechanisms operating at a medium-grained level. For example, consider the following method:

Listing 3.2. A Java method with an implicit information flow leak

```
1  public int leak(int guess){
2      guess == secret ? return 0 : return 1;
3  }
```

This method returns 0 if argument `guess` equals `secret` and returns 1 otherwise. An implicit information flow exists within this method because the method's return value depends on the value of `secret`. Consider the information-flow policy stating "method return values must not reveal information about `secret` values", notated MNI henceforth—a variation of noninterference [43], which states that high, or secret inputs, have no observable effect on low, or public, outputs. This policy is a hyperproperty at a fine-grained level because an execution $x \in \mathcal{E}_f^\infty$ containing both secret and public values must be compared with a, so called, low-equivalent execution $x' \in \mathcal{E}_f^\infty$, which is the same execution as x with all secret inputs removed, such that x' only contains low input values and public outputs, to determine whether there is a distinguishable difference between the two executions. Let $\mathcal{E}_{f_H}^\infty$ be the set of all fine-grained executions containing both secret and nonsecret inputs, and public outputs such that $\mathcal{E}_{f_h}^\infty \subseteq \mathcal{E}_f^\infty$; let $\mathcal{E}_{f_L}^\infty$ be the set of all fine-grained executions containing only nonsecret inputs and public outputs such that $\mathcal{E}_{f_L}^\infty \subseteq \mathcal{E}_f^\infty$. Let $NI : \mathcal{E}_{f_H}^\infty \times \mathcal{E}_{f_L}^\infty \rightarrow \{true, false\}$ be a predicate that inputs two fine-grained traces $x \in \mathcal{E}_{f_H}^\infty$ and $x' \in \mathcal{E}_{f_L}^\infty$ such that $x' \approx_L x$ (i.e., x' is low-equivalent to x) and outputs $true$ if x and x' are noninterfering, meaning that x' does not reveal any information about the secret values in x, and $false$ otherwise. MNI can be encoded as a fine-grained hyperproperty in the following way:

$$MNI_f = \{X \subseteq \mathcal{E}_f^\infty \mid \forall x \in X : \exists x' \in X : x' \approx_L x \wedge NI(x, x')\}.$$

Because MNI is only concerned with method return values revealing information about secret values, we can encode MNI as a medium-grained property ranging over the possible sequences of method executions. Let $NI \subseteq E_m$ now refer to the set of all noninterfering methods (i.e., the set of all methods where the low-execution paths of the methods do not reveal any information about secret values in the high-execution paths of the methods), and $false$ otherwise. MNI can now be encoded as a medium-grained property in the following way:

$$MNI_m = \{x \in \mathcal{E}_m^\infty \mid \forall \langle e, e' \rangle \in x : (e \in NI \Rightarrow e = e') \wedge e' \in NI\}$$

Notice that the fine-grained encoding of MNI (i.e., MNI_f) is a policy that is not a property, because the policy defines a relationship between two fine-grained executions. The medium-grained encoding of MNI (i.e., MNI_m) is a property, because the property holds on each medium-grained trace in isolation. This change in policy classification is due to the level of granularity at which the policy is encoded. Conceptually, medium-grained events capture the code-body of medium grained events. A mechanism enforcing a policy at a medium-grained level can analyze the body of medium-grained events before allowing them to execute.

Runtime code analysis allows for more precise policy enforcement—as opposed to static code analysis—by security mechanisms when they can determine program execution paths based on runtime values. However, to enforce policies at this level of granularity, policy writers must be familiar with low-level intermediate or machine code when source code is not available.

Such medium-grained mechanisms have been used to enforce information-flow policies, such as information-release policies (e.g., [7]), which is a policy that downgrades the security level of some information under certain conditions, thus allowing some information to flow from secret values to nonsecret values. These mechanisms operate by intercepting program execution at the guard of a conditional statement, and they perform code analysis on the body of the statement to determine whether allowing the conditional to execute will leak more sensitive information than what is allowed by the policy.

3.5 Coarse-Grained Policy Enforcement

A *coarse-grained policy* ranges over coarse-grained traces.

Definition 6. *A policy P_c is coarse grained iff*

$$P_c \subseteq 2^{\mathcal{E}_c^\infty}.$$

Definition 7. *A property $\hat{p_c}$ is coarse-grained iff*

$$\hat{p_c} \subseteq \mathcal{E}_c^\infty.$$

Coarse-grained program events are whole programs. Policies can be enforced at this level of granularity by performing code analysis on a whole program's source, intermediate, or binary code at runtime. Static code analyzers also analyze whole programs but cannot take into account runtime information.

Static code analyzers can be encoded as runtime mechanisms by implementing them to intercept program execution at a program's entry point. For example, in many high-level languages (e.g., Java, C/C++, and Rust) the entry point is at the `main` method. If a program takes command line arguments, these arguments may be used to predict the execution paths of the program. If the arguments can be used to rule out a number of execution paths, then coarse-grained mechanisms can be more precise than statically operating mechanisms.

Many coarse-grained policies can be encoded as safety properties, where each trace in the property is a single exchange containing the input program and an output program event that satisfies the policy constraint. Recall that the first event in an exchange is an attempted event, and the second event in an exchange is an event that is actually executed.

To illustrate how policies can be encoded as single-exchanged, coarse-grained properties, consider an example confidentiality policy, notated NNA, stipulating that programs cannot access the network, perhaps to prevent them from sending sensitive information over the network. Let $NA \subseteq \mathcal{E}_c$ be the set of all coarse-grained exchanges where the second event in the exchange does not contain a network access. NNA can then be encoded in the following way:

$$NNA = \{\langle e, e' \rangle \in \mathcal{E}_c \mid \langle e, e' \rangle \in NA\}$$

.

This coarse-grained policy encoding can capture the executions of static code analyzers and coarse-grained runtime monitors, where these mechanisms input a coarse-grained event (i.e., a whole program) and output an event possibly satisfying the policy at hand. We can determine whether a static code analyzer or coarse-grained runtime monitor soundly enforces a policy by determining whether the set of mechanism executions is a subset of the policy executions.

Due to the undecidability of coarse-grained code analysis, runtime mechanisms operating at this level cannot always precisely enforce policies; however, in some cases where the decidability of the policy only depends on the runtime information gathered at the program's entry point (i.e., the command line arguments), coarse-grained runtime monitors can be more permissive than static code analyzers. Programs containing both secure and insecure runs, where the security of such programs depends on the values of command line arguments, will always be rejected by static code analyzers. In contrast, coarse-grained runtime monitors may be able to accept only the secure runs of such programs on the basis of their runtime values.

3.6 Hybrid Policy Enforcement

Hybrid policies range over hybrid executions, where hybrid executions contain exchanges encoded at two or more levels of granularity.

Definition 8. *A policy P_h is hybrid iff*

$$P_h \subseteq 2^{\mathcal{E}_h^\infty}$$

Definition 9. *A property $\hat{p_h}$ is hybrid iff*

$$\hat{p_h} \subseteq \mathcal{E}_h^\infty$$

Traditionally, hybrid mechanisms enforce policies by combining static code analysis and runtime enforcement. We've shown in Section 3.5 that static code analyzers can be encoded as runtime mechanisms that monitor code at a coarse-grained level and in Section 3.3 that traditional, general-purpose runtime mechanisms are fine-grained mechanisms. Traditional hybrid mechanisms can be encoded in this framework as pure runtime mechanisms that monitor both coarse- and fine-grained program events. However, our approach to hybrid policy enforcement is more general because a hybrid policy ranging over hybrid executions can contain coarse-, medium-, and fine-grained exchanges.

Mechanisms that enforce policies exclusively at coarse- or medium-grained levels may not always be able to precisely predict control flow when medium-grained events are deeply nested within coarse- and other medium-grained events (e.g., nested method calls). However, hybrid runtime mechanisms can overcome this limitation by refining policy enforcement during a program's execution. For example, program rewriters can be encoded as hybrid runtime mechanisms by intervening at a program's entry point for coarse-grained code analysis. If code modification or inlined checks are necessary, the mechanism can output a new version of the program with the necessary modifications. A mechanism operating in this way is similar to runtime mechanisms that replace unsafe library function calls with safe versions (e.g., [14]), except the hybrid runtime mechanism outputs a coarser-grained program event, namely a safe program, with inlined checks for medium- or fine-grained events. If the mechanism cannot decide how to respond when a medium-grained event is reached during execution, perhaps due to nested method calls depending on runtime information, the mechanism can further refine policy enforcement by outputting a new medium-grained event with inlined checks to collect runtime information when the nested method calls are reached.

Chapter 4: Fine-Grained Policy Enforcement on Coauthentication

This chapter presents Coauthentication, which is a family of cryptographic protocols for authentication [63]. Section 4.1 presents a brief overview of Coauthentication. Section 4.2 presents a protocol variant of Coauthentication, and Section 4.3 demonstrates how forward secrecy is a property that can be enforced at a fine-grained level on the protocol variant presented in Section 4.2. Section 4.4 presents formal verification results of the forward-secure Coauthentication protocol, illustrating that the protocol does indeed satisfy forward secrecy.

4.1 Coauthentication

Coauthentication [63] is a multi-instance, single-factor authentication method designed to mitigate theft-based, phishing, man-in-the-middle, and replay attacks, which are all common attacks for single-factor authentication methods [71]. Possible single-factor authentication methods are either knowledge-based (e.g., PINs or passwords), token-based (e.g., authentication card), or biometrics-based (e.g., fingerprints or retina scan).

Multi-factor authentication methods aim to improve security by combining two or more factors. For example, token- and knowledge-based factors can be combined to withdraw money from ATM machines (i.e., a debit card and PIN). Multi-factor authentication methods improve security because each factor must be compromised in order for an attack to be successful [68]. However, when combining multiple factors, the overall authentication method inherits each of the weaknesses and inconveniences associated with each factor. For example, combining a debit card and PIN requires a user to remember to carry the debit card and

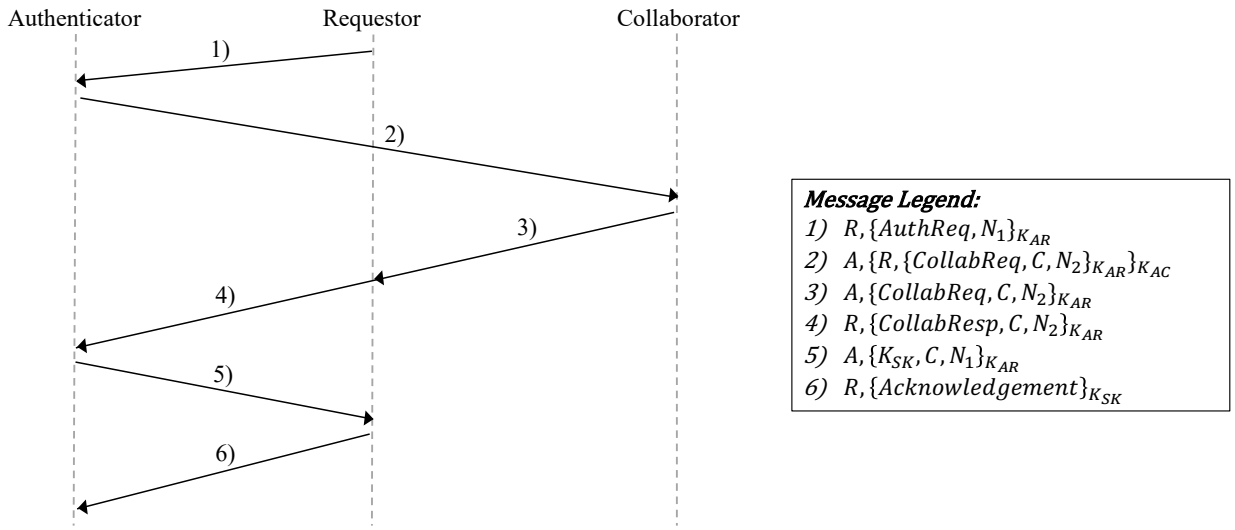

Figure 4.1. A variation of a 2-device Coauthentication protocol.

remember the PIN. If a user loses the debit card, the user cannot authenticate to an ATM with knowledge of the PIN alone; conversely, if the user forgets the PIN, in most cases, the user cannot withdraw money with the debit card alone.

Coauthentication is designed to improve security in a way that multi-factor authentication does (i.e., requiring successful attacks to compromise all authentication factors involved) but without inheriting the weaknesses and inconveniences of multiple factors. Coauthentication is a token-based authentication method that requires users to use multiple preregistered devices to authenticate to a system or service. Because Coauthentication only requires devices, Coauthentication is a single-factor method; however, we call it a multi-instance, single-factor method because Coauthentication requires at least two devices to authenticate to a system or service (i.e., multiple instances of the required factor).

The next section introduces a variation of a 2-device Coauthentication protocol.

4.2 Coauthentication Protocol

The 2-device Coauthentication protocol, shown in Figure 4.1, consists of three components: a requesting device, called the requestor, a collaborating device, called the collaborator, and an authentication server, called the authenticator. During device registration, the authenticator and the requestor share a secret key K_{AR}, and the authenticator and the collaborator share a secret key K_{AC}.

To authenticate to a service:

1. The requesting device initiates a Coauthentication session by sending its identifier R and a message encrypted with key K_{AR} to the authenticator. The encrypted message contains an authentication request $AuthReq$ and a nonce N_1, which will authenticate the authenticator to the requestor.

2. The authenticator decrypts message 1) and sends its identifier A along with a doubly encrypted message to the collaborating device; the inner message contains a collaboration request $CollabReq$, the collaborator's identifier C, and a nonce N_2, which will authenticate the collaborator to the authenticator, all encrypted with key K_{AR}. The outer message contains the inner message, and the requestor's identifier R encrypted with key K_{AC}.

3. The collaborator decrypts the outer message of message 2) and forwards the authenticator's identifier A and the inner message to the requestor. The authenticator's identifier is sent to inform the requestor that the message originated from the authenticator. Therefore, the requestor should use the key shared between itself and the authenticator to decrypt the message (i.e., K_{AR}).

4. The requestor decrypts message 3) and sends its identifier R and a message encrypted with K_{AR} containing a collaboration response $CollabResp$, the collaborator's identifier C, and nonce N_2 to the authenticator.

5. The authenticator decrypts message 4) and sends its identifier A and a message encrypted with K_{AR} containing a session key K_{SK}, the collaborator's identifier C, and nonce N_1 to the requestor.

6. The requestor decrypts message 6) with K_{AR} and sends its identifier R and a message encrypted with the session key K_{SK} containing an acknowledgement to the authenticator.

Given $n \geq 2$ preregistered devices, Coauthentication requires at least $m \geq 2$ of the preregistered devices to participate in an authentication. Therefore, the minimum number of devices required for Coauthentication is two devices. The security of Couathentication improves when $n > 2$. For example, if the number of preregistered devices $n = 3$ then a user can authenticate with any number of m devices such that $n \geq m \geq 2$, providing flexibility for a user to authenticate with any 2 out of the 3 preregistered devices. With the $n \geq m \geq 2$ constraint, Coauthentication protects against the compromise of at most $m - 1$ cryptographic keys. In the 2-device Coauthentication protocol shown in Figure 4.1, Coauthentication protects against the compromise of at most one cryptographic key.

4.3 Enforcing Forward Secrecy

Forward secrecy is a desirable property for many authentication protocols. It requires the session keys, which are short-term cryptographic keys, of previous sessions to be protected from adversaries in the event that an adversary obtains a long-term cryptographic key in a future session [73].

The protocol shown in Figure 4.1 does not satisfy forward secrecy. If an attacker obtains K_{AR}, the attacker can use K_{AR} to obtain the session keys of previous Coauthentication sessions, because K_{AR} is never updated.

We can enforce forward secrecy on the 2-device Coauthentication protocol shown in Figure 4.1 at a fine-grained level by equipping each preregistered, target device (i.e., the requestor and collaborator) and the authenticator with a runtime monitor that enforces forward secrecy, as shown in Figure 4.2. Each monitor monitors all messages incoming to and outgoing from its target device and has the ability to modify messages as necessary. Each monitor can be viewed as an inlined reference monitor that is inlined into its corresponding target device; therefore, all messages sent between a target device and its corresponding runtime monitor are local to the device and not publicly observable.

Device registration for the protocol shown in Figure 4.2 is similar to device registration for the protocol shown in Figure 4.1. The authenticator and requestor share a secret key K_{AR}, and the authenticator and the collaborator share a secret key K_{AC}. Each runtime monitor has access to the secret keys stored by its corresponding target device. That is, the requestor monitor has access to K_{AR}, the collaborator monitor has access to K_{AC}, and the authenticator monitor has access to all keys stored by the authenticator (i.e., K_{AR} and K_{AC}). Furthermore, the authenticator monitor and the requestor monitor also initially share secret key K_{AR}, and the authenticator monitor and the collaborator monitor initially share secret key K_{AC} at device registration. During the protocol, the secret keys shared between each device and the authenticator will remain the same, while the secret keys shared between the runtime monitors will be updated to ensure forward secrecy.

At the start of the coauthentication session, the requestor will attempt to send an authentication request to the authenticator. This message is intercepted by the requestor's monitor, decrypted with the key shared between the requestor and authenticator K_{AR}, re-encrypted with the key shared between the requestor monitor and the authenticator monitor,

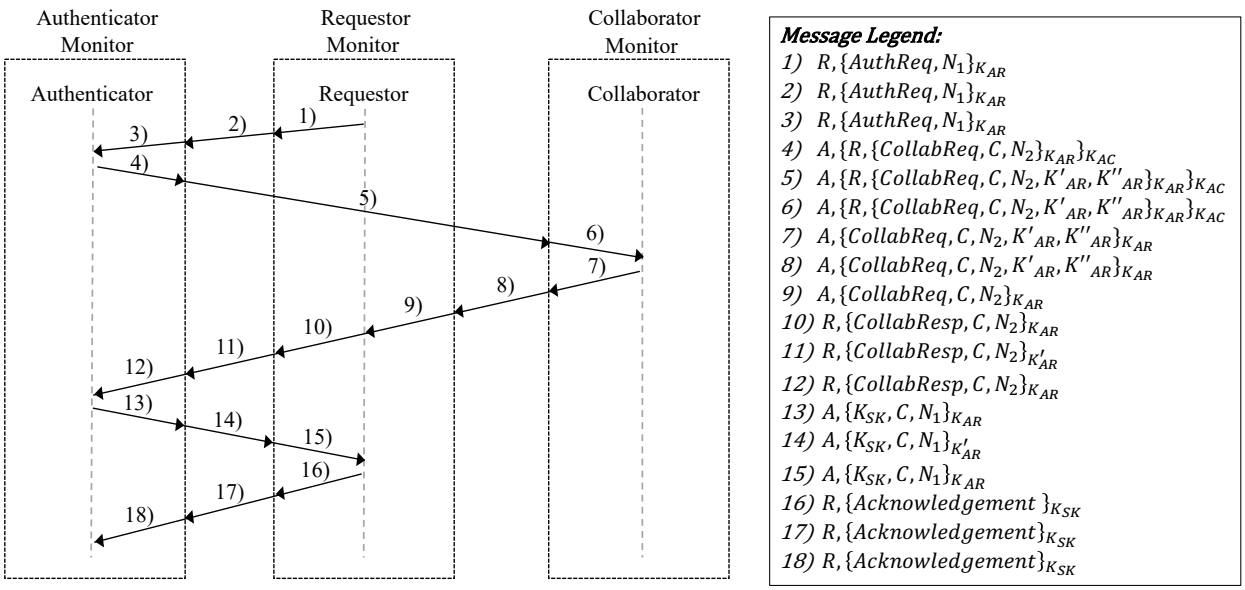

Figure 4.2. Enforcing forward secrecy on a 2-device couathentication protocol.

which is also K_{AR} for the first run of coauthentication, and sent to the authenticator monitor. The authenticator monitor decrypts the message with the key shared between itself and the requestor monitor K_{AR}, re-encrypts with the key shared between the authenticator and requestor K_{AR}, and forwards the message to the authenticator.

The authenticator then attempts to send a collaboration request to the collaborator, which is a doubly encrypted message. The inner message is encrypted with the secret key shared between the authenticator and requestor K_{AR}, and the outer message is encrypted with the secret key shared between the authenticator and the collaborator K_{AC}. The authenticator monitor intercepts this message, decrypts the message first with K_{AC} and then with K_{AR}, injects two new keys K'_{AR} and K''_{AR} into the inner message, re-encrypts the inner message with the key shared between itself and the requestor monitor K_{AR} and re-encrypts the outer message with the secret key shared between itself and the collaborator monitor K_{AC}, and sends the message to the collaborator monitor. The collaborator monitor decrypts the message with the key shared between itself and the authenticator monitor K_{AC}, re-encrypts

the message with the key shared between the collaborator and the authenticator K_{AC}, and forwards the message to the collaborator.

The collaborator decrypts the message with the key shared between itself and the authenticator K_{AC} and then attempts to forward the inner encrypted message to the requestor. The collaborator monitor intercepts the message, and sends the message to the requestor monitor over a private channel, assumed to be inaccessible to an attacker. The requestor monitor decrypts the message with the key shared between itself and the authenticator monitor, extracts the two new keys K'_{AR} and K''_{AR}, updates the key shared between itself and the authenticator monitor to K'_{AR}, re-encrypts the rest of the message with the key shared between the requestor and authenticator K_{AR}, and sends the message to the requestor.

The requestor then decrypts the message with the key shared between itself and the authenticator K_{AR}, and attempts to send a collaboration response to the authenticator, encrypted with K_{AR}. The requestor monitor intercepts this message, decrypts it with the key shared between the requestor and the authenticator K_{AR}, re-encrypts it with the key shared between itself and the authenticator monitor K'_{AR}, and sends the message to the authenticator monitor. The authenticator monitor then decrypts the message with the key shared between itself and the requestor monitor K'_{AR}, re-encrypts it with the key shared between the authenticator and the requestor K_{AR}, and sends the message to the authenticator.

The authenticator then decrypts the message with K_{AR} and attempts to send the session key encrypted with K_{AR} to the requestor. The authenticator monitor then decrypts the message with K_{AR}, re-encrypts it with K'_{AR}, and sends the message to the requestor monitor. The requestor monitor then decrypts the message with K'_{AR}, re-encrypts it with K_{AR}, and sends the message to the requestor.

The requestor decrypts the message with K_{AR}, extracts the session key, and attempts to send an acknowledgement to the authenticator encrypted with the session key K_{SK}. The requestor monitor then forwards the message to the authenticator monitor, and the

authenticator monitor then forwards the message to the authenticator. Secret key K'_{AR} is discarded at the end of the current coauthentication session, and the authenticator monitor and requestor monitor both update the key shared between themselves to secret key K''_{AR}.

On the next Coauthentication session, the requestor will attempt to send an authentication request encrypted with K_{AR}. The requestor monitor will then intercept this message, decrypt it with K_{AR} because it can access the key shared between the requestor and authenticator, re-encrypt the authentication request with the key shared between itself and the authenticator monitor K''_{AR}, and send the message to the authenticator monitor. The rest of the protocol will continue as pictured in Figure 4.2, with each monitor decrypting all messages outgoing from its target device with the key shared between the target device device and the destination device, and re-encrypting the messages with the key shared between itself and the runtime monitor of the destination device; conversely, each monitor decrypts each incoming message from a runtime monitor with the key shared between itself and the runtime monitor, and re-encrypts the message with the key shared between its corresponding target device and the origin device.

4.4 Formal Evaluation

The principal security properties of the example Coauthentication protocols shown in Figures 4.1 and 4.2 have been formally verified with ProVerif [23, 24]. ProVerif uses a resolution-based strategy to verify that protocols satisfy desired security properties. A benefit of using ProVerif is that it can model arbitrarily many sessions of a protocol running concurrently.

4.4.1 Protocol Modeling

The protocol encodings faithfully follow the communications shown in Figures 4.1 and 4.2. To model key updates in the protocol shown in Figure 4.2, we used key tables [25,

p.37]. These key tables are only accessible to the runtime monitors of the legitimate actors (i.e., Requestor Monitor, Authenticator Monitor, and Collaborator Monitor) of the protocol. The protocols dynamically generate new keys (i.e., K'_{AR} and K''_{AR}) during an authentication session, and at the end of the session, the new long-term key (K''_{AR}) gets inserted into the key table.

Each protocol session ran six processes (i.e., authenticator, authenticator monitor, requestor, requestor monitor, collaborator, and collaborator monitor) and the main ProVerif process considered arbitrarily many sessions of a protocol running concurrently.

The ProVerif encodings of the Coauthentication protocols shown in Figures 4.1 and 4.2, and the properties verified, are available online [35].

4.4.2 Attack Models and Assumptions

Coauthentication, like multi-factor techniques, protects against theft of any one authentication secret. The secrets in Coauthentication are cryptographic keys. Theft of Coauthentication secrets may occur in any way, including by remotely compromising devices to obtain their stored keys or physically stealing devices.

Attackers are assumed to be active and can eavesdrop on, insert, delete, and modify communications. Attackers may mount replay and man-in-the-middle attacks and are not constrained to operate according to any of the protocols.

Attackers are however assumed to be incapable of cryptanalysis; attackers can only infer plaintexts from ciphertexts when also having the required secret key. Without such an assumption, attackers could extract credentials like session keys simply by monitoring and cryptanalyzing legitimate authentications.

The Coauthentication protocol presented in Figure 4.2 protects against attackers who know all of the secrets stored on a device that the victim user possesses. We call such attacks *key-duplication attacks*. For example, an attacker may duplicate a device's secret keys by

remotely compromising the device. Alternatively, the attacker may physically steal a device, duplicate all keys accessible to the device, and return the device to the victim user, who may be unaware of the theft and duplication.

To protect against key-duplication attacks, the Coauthentication protocol presented in Figure 4.2 assumes that a private communication channel, inaccessible to attackers, exists between the requestor and collaborator devices. Such an assumption is necessary because the duplicated keys must be updated through some channel inaccessible to the attacker; otherwise, the attacker—who has all of the victim device D's keys—could decrypt and obtain any updated keys sent to D, and modify any updated keys sent from D. Private channels may be implemented with short-range communications, such as NFC, zigbee, wireless USB, infrared, or near-field magnetic induction, under the assumption that attackers cannot access such communications because they are on direct, device-to-device channels.

The Coauthentication protocol presented in Figure 4.1 does not require a private channel between the requestor and collaborator devices; consequently, this protocol does not protect against key-duplication attacks. Therefore, we assume a weaker attack model for this protocol, one that assumes authentication secrets (K_{AR} and K_{AC}) are only accessible to attackers through device theft (without duplicating the keys, and returning, the devices). In other words, the attack model for this all-public-channel protocol assumes that if an attacker obtained a device D's authentication secret, then D's legitimate user no longer posesses D. In terms of the ProVerif encodings, this weaker attack model means that, in cases where attackers are assumed to know K_{AR}, the collaborator does not respond to collaboration requests. The justification is that if an attacker has acquired K_{AR}, then by assumption the legitimate user does not possess the requestor, so collaboration requests must be for unauthorized, attacker-initiated authentications. It is assumed that, with appropriate collaboration policies, users do not approve collaborations for unauthorized authentications.

Table 4.1. Verification setup of each protocol for three different runs.

Protocol	Attackers' Knowledge		
	Run 1	Run 2	Run 3
Figure 4.1	No secrets	K_{AC}	K_{AR}
Figure 4.2	No secrets	K_{AC}	K_{AR} and K''_{AR}

Both Coauthentication protocols presented in this dissertation assume that devices in the user's possession run as intended during the Coauthentication process. Without such an assumption, malware on the user's requestor device could simply leak decrypted session keys or any other unencrypted private data, and malware on the user's collaborator device could simply approve an attacker's authentication requests. Protecting against malware that is actively running on a device in the user's possession, while the user is authenticating, is beyond the scope of coauthentication.

Both Coauthentication protocols presented in this dissertation also assume that authenticators run as intended during the Coauthentication process. Without such an assumption, malware on the authenticator could simply leak secrets or allow all authentication requests. Protecting against malware on authenticators is beyond the scope of Coauthentication.

4.4.3 Verification Setup

Each protocol was verified in 3 runs; as shown in Table 4.1.

1. The first run began with attackers knowing no secret keys.

2. The second run began with attackers knowing the long-term key accessible to the collaborator. That is, attackers were given K_{AC};

Table 4.2. Verification results,"✓" indicates that ProVerif proved the property.

Protocol	*P1: Secrecy of the K_{SK}*			*P2: Authentication of R to A*			*P3: Authentication of A to R*		
	Run 1	Run 2	Run 3	Run 1	Run 2	Run 3	Run 1	Run 2	Run 3
Figure 4.1	✓	✓	✓	✓	✓	✓	✓	✓	✓
Figure 4.2	✓	✓	✓	✓	✓	✓	✓	✓	✓

3. The third run began with attackers knowing all the long-term keys accessible to the requestor. For the protocol shown in Figure 4.1, attackers were given K_{AR}, and for the protocol shown in Figure 4.2, attackers were given K_{AR} and K''_{AR}.

In all 3 runs of each of the protocols, we verified the following security properties.

1. *P1*: Secrecy of the session key.

   - The session key K_{SK} is only known to the authenticator and requestor. This property subsumes forward secrecy of session keys (K_{SK}) in the third run of the protocol shown in Figure 4.2 because knowing the requestor's future authentication secret K''_{AR} does not leak session keys.

2. *P2*: Authentication of R to A.

   - With one exception, we specified authentication of R to A as requiring that if the authenticator receives an acknowledgment of a session key (and therefore believes it shares the session key with the requestor) then the requestor was indeed its interlocutor and the collaborator indeed collaborated. This is an event-based property [100] having the form

$$endA \implies (beginA \wedge collabA),$$

where *endA* refers to the event of A receiving the acknowledgment, *beginA* to R sending the authentication request, and *collabA* to C sending its participation message (in the eighth message of Figure 4.2 and the third message of Figure 4.1). The one exception to encoding *P2* in this way is for the second run of the all-public-channel protocol (Figure 4.1), where the attacker is given the long-term key accessible to the collaborator (i.e., K_{AC}). In this case, the attacker may use the collaborator's key to obtain and collaborate with legitimate authentication requests, thus helping legitimate authentications succeed, which we do not consider an attack. Therefore, for the second run of the Figure 4.1 protocols, we specify property *P2* as only requiring

$$endA \implies beginA,$$

that is, if the authenticator believes it shares the session key with the requestor then the requestor was indeed its interlocutor (but the attacker, rather than the collaborator, may have collaborated).

3. *P3*: Authentication of A to R.

   - This property is symmetric to *P2* and, with one exception, requires that if the requestor sends an acknowledgment of a session key (and therefore believes it shares the session key with the authenticator) then the authenticator was indeed its interlocutor and the collaborator indeed collaborated. This property has the form

     $$endR \implies (beginR \wedge collabR),$$

     where *endR* refers to R sending the acknowledgment, *beginR* to A receiving the authentication request, and *collabR* to C sending its participation message.

As with *P2*, the one exception to encoding *P3* in this way is for the second run of the all-public-channel protocol (Figure 4.1), in which case *P3* only requires

$$endR \implies beginR,$$

for the same reason explained for property *P2*.

4.4.4 Verification Results

Table 4.2 shows the verification results. ProVerif found no attacks on any of properties *P1–P3* in any runs of either of the protocols. That is, ProVerif did not refute any of *P1–P3* in any runs of either of the protocols.

ProVerif proved *P1–P3* for all three runs of both protocols. We also note that these results are for the stronger, injective-correspondence versions of properties *P2* and *P3*. The injective-correspondence versions require there to be a unique predecessor event for each end event [25, pp.19–22]; for example, the injective version of *P2* requires that for each *endA* event there exists a unique *beginA* predecessor event. The non-injective versions allow end events to have non-unique predecessor events. ProVerif was able to prove the weaker, non-injective version of property *P2* for all runs of both protocols.

Chapter 5: Implementation

This chapter presents details about our prototype implementation of the theoretical framework presented in Chapter 3 and explains how granular security policies can be enforced on Java bytecode applications at runtime.

5.1 Architecture

The implementation is composed of 3 core components: AspectJ, a code analyzer, and JaBRO.

5.1.1 AspectJ

AspectJ is an aspect-oriented Java language extension that allows policy writers to define *aspects* [54], similar to Java classes. Aspects contain *pointcuts*, which capture one or more *join points*, and *advice*. "Join points are well-defined points in the execution of the program" [54, p. 329]. Table 5.1 lists the join point types allowed by AspectJ and the possible policy granularities enforceable at each join point. The `main` method execution is the only join point that can be used to enforce a coarse-grained policy because it is the entry point of the program. Any join point that has a body of code provides the capability for enforcing medium-grained policies, and every join point provides the capability for enforcing fine-grained policies, because fine-grained policies do not require code analysis. Advice is a block of code written by the policy writer that directs a mechanism on how to enforce a particular policy. *Before* advice directs a mechanism to execute some block of code before a security-relevant event occurs. For example, a policy may require all file system operations

Table 5.1. Possible policy granularities enforceable at each AspectJ join point type.

AspectJ join point type	**Coarse**	**Medium**	**Fine**
Main method execution	✓	✓	✓
Non-main method execution		✓	✓
Method call		✓	✓
Constructor call		✓	✓
Static-initializer execution		✓	✓
Object pre-initialization		✓	✓
Object initialization		✓	✓
Handler execution		✓	✓
Advice execution		✓	✓
Field reference			✓
Field assignment			✓

to be logged. Therefore, the mechanism can intercept all file system operations and log them before allowing them to proceed. *After* advice directs a mechanism to execute some block of code after a security-relevant event occurs, but before proceeding to the next event. For example, a mechanism may be required to execute additional code after an exception is thrown. Lastly, *around* advice directs a mechanism to execute a block of code instead of a security-relevant event. For example, a mechanism can replace all unsafe library methods with safe versions.

5.1.2 Code Analyzer

The code analyzer is code written by the policy writer to analyze Java class files (e.g., data or control flow analysis). This analysis ensures that a body of code adheres to the security policy in question. For example, a policy may disallow Java applications from running external programs. To enforce this policy, a code analyzer can analyze all class files of an application to ensure that the application does not use the `java.lang.Runtime.exec()` method.

5.1.3 JaBRO

JaBRO (Java Bytecode Rewriter and Optimizer) is an extensible Java library that we have developed to extend the functionality of AspectJ to enable code analysis on optimized bytecode at runtime. It is composed of Javassist [29], which is a Java library for editing Java bytecode, and Soot [93], which is a program analysis and code optimization tool for Java bytecode.

JaBRO uses Javassist to rewrite Java bytecode to include runtime arguments that are observed during a program's execution. JaBRO is currently able to rewrite Java bytecode to include runtime arguments of all primitive types, and `String` and `File` objects. Once JaBRO rewrites the bytecode of the original application to include runtime information, JaBRO invokes Soot to propagate the runtime information throughout the application and performs program optimizations where necessary. JaBRO is extensible because policy writers can extend JaBRO to be able to rewrite Java bytecode to include runtime arguments of arbitrary types.

The following code in Listing 5.1 illustrates how a policy writer may write an aspect that uses JaBRO to enforce a policy on the example `writeSubString` method presented in Chapter 3.

Listing 5.1. An example aspect for enforcing a medium-grained policy

```
1  public aspect Monitor{
2      void around(int i, String s):
3          call(public void writeSubString(int,String))
4          && args(i,s){
5              String className = thisJoinPoint.getSignature()
6                  .getDeclaringTypeName();
7              String methodName = thisJoinPoint.getSignature()
8                  .getName();
9              String dependencies = System
10                 .getProperty(``user.dir'')
11                 +File.separator+``example.jar'';
12
13             try{
14                 JaBRO j = new JaBRO(className, methodName,
15                     new Object[]{i,s}, dependencies);
16                 File f = j.runMediumMethod();
17                 CodeAnalyzer c = new CodeAnalyzer(f);
18                 if(c.checkResourcesReleased()){
19                     proceed(i,s);
20                 }else{
21                     defaultMethod();
22                 }
23             } catch(Exception e){/* omitted */}
24         }
25 }
```

The advice and pointcut is specified in lines 2–4. This policy uses *around* advice because if the method violates the policy, the policy writer can direct the mechanism to output an alternative method that does not violate the policy. The advice takes two arguments `String s` and `int i`, which corresponds to the runtime arguments for the `writeSubString` method. The pointcut `call(public void writeSubString(int,String))` denotes the security-relevant event to intercept, and `args(i,s)` binds `int i` and `String s` to the runtime arguments of `writeSubString`. Lines 5-6 use the `thisJoinPoint` construct to obtain `writeSubString`'s enclosing class name (i.e., the class that `writeSubString` is declared

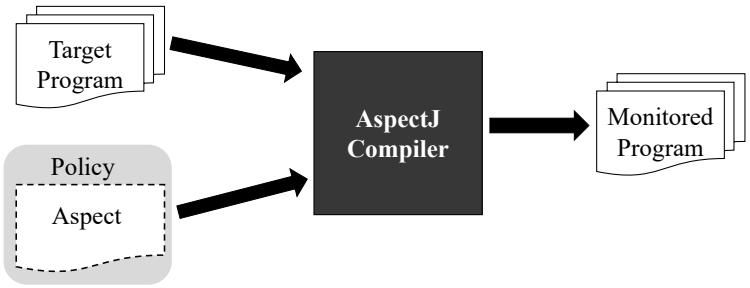

Figure 5.1. Weaving a fine-grained policy into a target program.

in). Lines 7-8 use the `thisJoinPoint` construct to get the fully qualified method name for `writeSubString`. The program's dependencies are retrieved from the path to the program in line 9. Java applications can be packaged as JAR files that contain all of an application's dependent class files. We can retrieve the application dependencies by specifying the path to the application's JAR file. In this case, we assume `writeSubString` is a part of the `example.jar` application. Lines 14-15 create a new JaBRO object that takes as input the method's enclosing class name, the fully qualified method name, an array of the method's arguments, and the program's dependencies. JaBRO is then invoked for a medium-grained method and returns the class file with an optimized `writeSubString` method in line 16. The optimized class file is passed to the code analyzer in line 17. If `writeSubString` satisfies the policy then `writeSubString` proceeds with its original runtime arguments in line 19, otherwise the policy writer specifies how the mechanism should handle the policy violation in line 21 (i.e., execute a `defaultMethod()` assumed to satisfy the policy at hand).

JaBRO allows policies that are conservatively enforced statically to be enforced more precisely at runtime by taking into account runtime information. Policy writers can use JaBRO as an external library to aid in precise policy enforcement when writing policies to be enforced at runtime. JaBRO is publicly available online [34].

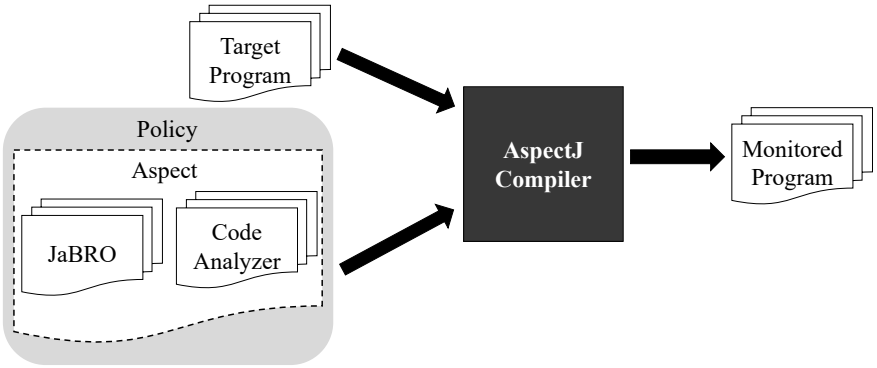

Figure 5.2. Weaving a coarse- or medium-grained policy into a target program.

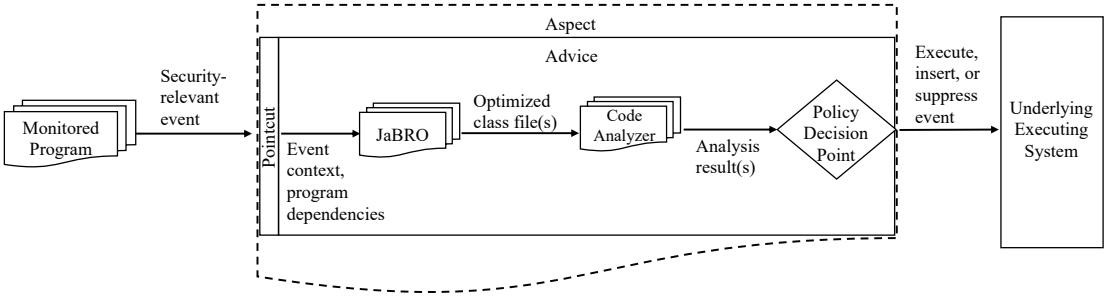

Figure 5.3. Runtime enforcement of coarse- or medium-grained policy.

5.2 Policy Enforcement

Coarse- and medium-grained policies are composed of an aspect, JaBRO, and a code analyzer. Fine-grained policies only consist of an aspect and can be implemented with just the native AspectJ language constructs because they do not require code analysis. To monitor the target program, the AspectJ compiler is used to weave the policy into the target program, producing a self-monitoring program as shown in Figures 5.1 and 5.2.

Figure 5.3 illustrates how coarse- and medium-grained policies are enforced during a monitored program's execution:

1. A security-relevant event is intercepted by a pointcut.

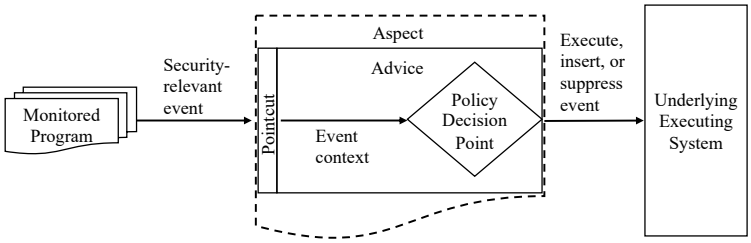

Figure 5.4. Runtime enforcment of a fine-grained policy.

2. Event context information, which includes the event name, its enclosing class name, and runtime arguments, and the program's dependencies are input to JaBRO, which is invoked from inside of the advice.

   - AspectJ provides the capability to obtain the event context information through the join point construct.
   - The program's dependencies are needed to resolve type information during the optimization process.

   JaBRO uses the event's context information to obtain the class file in which the event is declared from the system search path. JaBRO then rewrites the event to include its runtime arguments within the event's code body. If the policy is coarse grained, JaBRO optimizes all of the program's class files. If the policy is medium grained, JaBRO only optimizes the class file that the event is declared in.

3. Code analysis is performed on the optimized class file(s), and an analysis result is output.

4. The policy decision point decides to execute, suppress, or insert an event based on the analysis result.

To further illustrate JaBRO's functionality and its importance, consider the `writeSubString(5,"Hello, World!")` method call presented in Chapter 3. After obtain-

ing `writeSubString`'s enclosing class file, JaBRO first rewrites the method by inserting the method's runtime arguments in the following way:

Listing 5.2. JaBRO rewriting a method with its runtime argumented

```
1  public void writeSubString(int index,String s){
2      int index = 5;
3      String s = "Hello, World!";
4      FileWriter writer = null;
5      try{
6          writer = new FileWriter(new File("substrings.txt"));
7          if(index < 10){
8              String newS = s.substring(0,index);
9              writer.write(newS);
10             writer.close();
11         } else{
12             String newS = s.substring(index);
13             writer.write(newS);
14         }
15     } catch(IOException e){
16         if(writer != null){
17             writer.close();
18         }
19     }
20 }
```

After rewriting, JaBRO propagates the arguments throughout the method in the following way:

Listing 5.3. JaBRO propagating runtime values throughout the method

```
1  public void writeSubString(int index, String s){
2      int index = 5;
3      String s = "Hello, World!";
4      FileWriter writer = null;
5      try{
6          writer = new FileWriter(new File("substrings.txt"));
7          if(5 < 10){
8              String newS = "Hello, World!".substring(0,5);
9              writer.write(newS);
10             writer.close();
11         } else{
12             String newS = "Hello, World!".substring(5);
13             writer.write(newS);
14         }
15     } catch(IOException e){
16         if(writer != null){
17             writer.close();
18         }
19     }
20 }
```

After propagating the arguments throughout the method, JaBRO optimizes the method in the following way:

Listing 5.4. JaBRO optimizing a method

```
1  public void writeSubString(int index, String s){
2      FileWriter writer = null;
3      try{
4          writer = new FileWriter(new File("substrings.txt"));
5          String newS = "Hello, World!".substring(0,5);
6          writer.write(newS);
7          writer.close();
8      } catch(IOException e){
9          if(writer != null){
10             writer.close();
11         }
12     }
13 }
```

Given the runtime values, JaBRO is able to eliminate the `else` branch. A policy stipulating that file resources acquired in a method be released before the end of the method's execution can be enforced more precisely by performing code analysis on the optimized method, shown in Listing 5.4, rather than on the original method, shown in Listing 3.1.

Figure 5.4 illustrates how policies can be enforced at a fine-grained level. Similar to coarse- and medium-grained policies, security-relevant events are intercepted by pointcuts defined in the aspect. The policy decision point can then make an enforcement decision, which may use event context information.

Chapter 6: Empirical Evaluation

This chapter presents details about the empirical evaluation of our implementation.

6.1 Case Study

To evaluate the effectiveness of the implementation, we enforced three security policies on two popular, open-source Java applications. The first application was the US National Archives and Resource Administration's (NARA) file analyzer and metadata harvester [92], which analyzes and collects metadata on files, and can execute tests on files in a given directory. The second application was JPlag [1], which is a software plagiarism detection tool. Both of these applications can conduct operations on files and have the ability to access potentially sensitive information stored on a system; therefore, these capabilities permitted us to express practical policies over the applications.

We implemented policies to cover each level of granularity (i.e, coarse, medium, and fine). The coarse-grained policy disallowed network connections by prohibiting the use of `java.net.Socket`. This policy was enforced to ensure that neither application could exfiltrate sensitive information over the network. The medium-grained policy required file resources acquired within a method to be released before the end of the method's execution [66]. Because both applications were able to access the file system and perform file operations, it was imperative to ensure that the applications could not deplete system resources. The fine-grained policy required files containing sensitive information to be hidden from the applications. This policy was enforced by checking whether the values of file oper-

Table 6.1. Average experimental performance results of 100 runs.

Application	Policy								
	Coarse			Medium			Fine		
	AspectJ weaving (ms)	Unmonitored (ms)	Monitored (ms)	AspectJ weaving (ms)	Unmonitored (ms)	Monitored (ms)	AspectJ weaving (ms)	Unmonitored (ms)	Monitored (ms)
NARA File Analyzer and Metadata Harvester	2290	11	33540	2910	0.26	1010	2910	0.10	0.11
JPlag (v2.11.8)	2590	392	32550	3580	0.39	1690	2520	0.02	0.03

ations (i.e., file names) were contained in a sensitive file list. This policy was important to limit the access rights of the applications.

To evaluate the overhead introduced by the implementation, we measured the average time to weave and enforce each policy. Results are shown in Table 6.1. The AspectJ compiler was used to weave each policy into each target application as shown in Figures 5.1 and 5.2. The average execution time of each program event was measured without enforcing the policy, indicated by the columns labeled unmonitored, and with enforcing the policy, indicated by the columns labeled monitored. The NARA file analyzer and metadata harvester application required user interaction; therefore, instead of measuring the total execution time of the application, we measured the time to enforce the coarse-grained policy, for the monitored column, and the time to start-up the application, for the unmonitored colunn. Each policy was enforced separately on each application. Each experiment was conducted 100 times on a MacBook Pro laptop with a 2.9 GHz Quad-Core Intel Core i7 processor and 16 GB of RAM.

6.2 Summary of Results

Each of the policies were successfully enforced on both applications. The results in Table 6.1 indicate that the execution times of the applications were significantly impacted

by the enforcement of the coarse- and medium-grained policies, which is due to the execution time of JaBRO and the code analyzer. At a coarse-grained level, the entire application must be traversed twice: first, JaBRO must rewrite the `main` method and optimize the entire application, and second, code analysis must be conducted on the entire optimized application. The overhead introduced by coarse-grained policy enforcement is likely unacceptable for time-sensitive applications. However, for security-critical applications where runtime performance is not a concern and static analysis is too conservative, the approach may be advantageous. The overhead introduced by medium-grained policy enforcement may be more acceptable due to the smaller code fragment traversed by JaBRO and the code analyzer. The fine-grained policy only added 0.01 ms to the execution time of the fine-grained event for both applications.

We expected coarse-grained policies to be weaved significantly faster than medium- and fine-grained policies because the policy only needs to be weaved at a single point (i.e., the `main` method); however, the AspectJ compiler cannot differentiate between policy granularities and thus searched through the entire application looking for all possible matches to the `main` join point. The weaving time for coarse-grained policies may be improved by implementing a custom bytecode rewriter tailored for weaving coarse-grained policies.

Chapter 7: Conclusions and Future Work

This dissertation has aimed to improve our understanding of security mechanisms and how they enforce policies by presenting a unifying theory of security mechanisms. The theory included a framework in which all mechanisms are encoded as runtime mechanisms that operate at one or more levels of program code granularity. The practicality of the theoretical framework was demonstrated through a prototype implementation for enforcing security policies on Java bytecode applications. This chapter summarizes this dissertation's contributions (Section 7.1) and presents directions for future work (Section 7.2).

7.1 Summary

The first contribution of this dissertation is a formal framework for reasoning about security policy enforcement. The framework casts existing classes of security mechanisms into runtime mechanisms that operate at one or more levels of code granularity. A new taxonomy of security policies was presented based on the granularity of well-defined, modular program constructs that mechanisms analyze to enforce policies. We have shown that previous work has encoded security policies as fine-grained policies, but not all security mechanisms enforce policies at such a fine-grained level. Some policies can be encoded at a coarse- or medium-grained level, and such granular policy encodings can capture the executions of security mechanisms that operate at one or more levels of code granularity. This dissertation also presented a practical example of how forward secrecy can be enforced at a fine-grained level on an authentication method called Coauthentication [63].

The second contribution of this dissertation is a prototype implementation of the theoretical framework for enforcing security policies at various levels of code granularity on Java bytecode applications. The effectiveness and precision of the implementation hinges on an extensible Java library that we have developed called JaBRO. JaBRO allows runtime mechanisms operating at medium- and coarse-grained levels to enforce policies more precisely than statically operating mechanisms. We have evaluated the overhead introduced by the implementation by enforcing security policies at each level of code granularity. Coarse-grained policy enforcement significantly inhibits the runtime performance of applications because applications must be traversed twice: first by JaBRO and second by the code analyzer. We realize that the overhead introduced by coarse-grained policy enforcement may be unacceptable for time-sensitive applications; however, coarse-grained policy enforcement can still be beneficial for security-critical applications in cases where static enforcement is too conservative. Medium-grained policy enforcement may be more acceptable, depending on the size of the medium-grained event, and fine-grained policy enforcement did not significantly inhibit runtime performance

7.2 Future Work

The work presented in this dissertation can be extended in several ways. This section will discuss two possible directions for future work.

7.2.1 Additional Theoretical Analysis

One possible direction for future work would be additional theoretical analysis to determine the class of policies enforceable by runtime mechanisms with constrained sets of capabilities that operate at one or more levels of code granularity. Previous work has shown the class of policies enforceable by various runtime mechanisms with constrained sets of capabilities that only operate at the fine-grained level (e.g., execution monitors, edit

automata, mandatory results automata, stream monitoring automata, etc.). It would be interesting to see which class of policies are enforceable by each sort of mechanism when operating at each level of granularity and at two or more levels of granularity.

7.2.2 Domain-Specific Policy-Specification Languages

Another possible direction for future work would be to design and implement domain-specific policy-specification languages for granular security policies based on various language constructs. Policy-specification languages are important for policy writers because they can provide guarantees that the policy implemented is the policy intended through the languages type system and formal verification techniques [45]. Domain-specific policy-specification languages, designed around language constructs, might enable the synthesis of new kinds of runtime security mechanisms.

References

[1] JPlag - Detecting software plagiarism. https://github.com/jplag/jplag, 2015.

[2] Martín Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 33–44, 2002.

[3] Martín Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1):102–146, 2005.

[4] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. NAVEX: Precise and scalable exploit generation for dynamic web applications. In *USENIXSecurity Symposium*, pages 377–392, 2018.

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

[6] Robert C Armstrong, Ratish J Punnoose, Matthew H Wong, and Jackson R Mayo. Survey of existing tools for formal verification. *SANDIA REPORT SAND2014-20533*, 2014.

[7] Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *IEEE Computer Security Foundations Symposium*, pages 43–59. IEEE, 2009.

[8] Marco Avvenuti, Cinzia Bernardeschi, and Nicoletta De Francesco. Java bytecode verification for secure information flow. *ACM SIGPLAN Notices*, 38(12):20–27, 2003.

[9] Samiha Ayed, Muhammad Sabir Idrees, Nora Cuppens, and Frederic Cuppens. Achieving dynamicity in security policies enforcement using aspects. *International Journal of Information Security*, 17(1):83–103, 2018.

[10] Samiha Ayed, Muhammad Sabir Idrees, Nora Cuppens-Boulahia, Frédéric Cuppens, Monica Pinto, and Lidia Fuentes. Security aspects: a framework for enforcement of security policies using AOP. In *International Conference on Signal-Image Technology & Internet-Based Systems*, pages 301–308. IEEE, 2013.

[11] Dejan Baca, Kai Petersen, Bengt Carlsson, and Lars Lundberg. Static code analysis to detect software security vulnerabilities-does experience matter? In *International Conference on Availability, Reliability and Security*, pages 804–810. IEEE, 2009.

[12] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy*, pages 387–401, 2008.

[13] Anindya Banerjee, David A Naumann, and Stan Rosenberg. Expressive declassification policies and modular static enforcement. In *IEEE Symposium on Security and Privacy*, pages 339–353, 2008.

[14] Arash Baratloo, Navjot Singh, Timothy K Tsai, et al. Transparent run-time defense against stack-smashing attacks. In *USENIX Annual Technical Conference*, pages 251–262, 2000.

[15] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. Enforcing secure service composition. In *IEEE Computer Security Foundations Workshop*, pages 211–223, 2005.

[16] David Basin, Vincent Jugé, Felix Klaedtke, and Eugen Zălinescu. Enforceable security policies revisited. *ACM Transactions on Information and System Security*, 16(1):309–328, 2013.

[17] Lujo Bauer, Jarred Ligatti, and David Walker. More enforceable security policies. In *Proceedings of the Workshop on Foundations of Computer Security*. Citeseer, 2002.

[18] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with polymer. In *ACM Conference on Programming Language Design and Implementation*, volume 40, pages 305–314, 2005.

[19] Jean Bergeron, Mourad Debbabi, Jules Desharnais, Mourad M Erhioui, Yvan Lavoie, and Nadia Tawbi. Static detection of malicious code in executable programs. *Symposium on Requirements Engineering for Information Security*, 2001(184-189):79, 2001.

[20] Frédéric Besson, Nataliia Bielova, and Thomas Jensen. Hybrid information flow monitoring against web tracking. In *IEEE Computer Security Foundations Symposium*, pages 240–254. IEEE, 2013.

[21] Gaowei Bian, Ken Nakayama, Yoshitake Kobayashi, and Mamoru Maekawa. Java bytecode dependence analysis for secure information flow. *International Journal of Network Security*, 4(1):59–68, 2007.

[22] Arnar Birgisson, Mohan Dhawan, Ulfar Erlingsson, Vinod Ganapathy, and Liviu Iftode. Enforcing authorization policies using transactional memory introspection. In *ACM Conference on Computer and Communications Security*, pages 223–234, 2008.

[23] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *IEEE Computer Security Foundations Workshop*, pages 82–96, June 2001.

[24] Bruno Blanchet. ProVerif: Cryptographic protocol verifier in the formal model, 2016. http://prosecco.gforge.inria.fr/personal/bblanche/proverif/. Accessed 20 August 2019.

[25] Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre. ProVerif 1.98pl1: Automatic cryptographic protocol verifier, user manual and tutorial, December 2017. http://prosecco.gforge.inria.fr/personal/bblanche/proverif/manual.pdf. Accessed 20 August 2019.

[26] Franck Cassez, Anthony M Sloane, Matthew Roberts, Matthew Pigram, Pongsak Suvanpong, and Pablo Gonzalez de Aledo. Skink: Static analysis of programs in llvm intermediate representation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 380–384. Springer, 2017.

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

[28] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79, 2004.

[29] Shigeru Chiba. Javassist: Java bytecode engineering made simple. *Java Developer's Journal*, 9(1), 2004.

[30] Angelo Ciampa, Corrado Aaron Visaggio, and Massimiliano Di Penta. A heuristic-based approach for detecting SQL-injection vulnerabilities in web applications. In *International Workshop on Software Engineering for Secure Systems*, pages 43–49, 2010.

[31] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

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

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

[34] Shamaria Engram. JaBRO. https://github.com/shamaria/JaBRO, 2020.

[35] Shamaria Engram. ProVerif Coauthentication Files. https://github.com/shamaria/coauthentication-formal-models, 2020.

[36] Shamaria Engram and Jay Ligatti. Through the lens of code granularity: A unified approach to security policy enforcement. In *IEEE Conference on Applications, Information, and Network Security*. IEEE, 2020. To appear.

[37] Ulfar Erlingsson and Fred B Schneider. IRM enforcement of java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, 2000.

[38] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

[39] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, pages 32–45, 1999.

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

[41] David Gelles. Boeing 737 max: What's happened after the 2 deadly crashes? ttps://www.nytimes.com/interactive/2019/business/boeing-737-crases.html, 2019. Online; accessed 31-July-2020.

[42] Samir Genaim and Fausto Spoto. Information flow analysis for java bytecode. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 346–362. Springer, 2005.

[43] Joseph A Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–11. IEEE, 1982.

[44] William GJ Halfond and Alessandro Orso. AMNESIA: analysis and monitoring for neutralizing sql-injection attacks. In *International Conference on Automated Software Engineering*, pages 174–183, 2005.

[45] Kevin W Hamlen and Micah Jones. Aspect-oriented in-lined reference monitors. In *Workshop on Programming languages and analysis for security*, pages 11–20, 2008.

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

[47] Kevin William Hamlen. *Security policy enforcement by automated program-rewriting*. PhD thesis, 2006.

[48] José-Miguel Horcas, Mónica Pinto, and Lidia Fuentes. Closing the gap between the specification and enforcement of security policies. In *International Conference on Trust, Privacy and Security in Digital Business*, pages 106–118. Springer, 2014.

[49] Jose-Miguel Horcas, Mónica Pinto, Lidia Fuentes, Wissam Mallouli, and Edgardo Montes de Oca. An approach for deploying and monitoring dynamic security policies. *Computers & Security*, 58:20–38, 2016.

[50] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *International Conference on World Wide Web*, pages 40–52. ACM, 2004.

[51] Muhammad Sabir Idrees, Samiha Ayed, Nora Cuppens-Boulahia, and Frédéric Cuppens. Dynamic security policies enforcement and adaptation using aspects. In *International Conference on Trust, Security, and Privacy in Computing and Communcations*, volume 1, pages 1374–1379. IEEE, 2015.

[52] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *International Conference on Software Engineering*, pages 672–681. IEEE, 2013.

[53] Micah Jones and Kevin W Hamlen. Disambiguating aspect-oriented security policies. In *International Conference on Aspect-Oriented Software Development*, pages 193–204, 2010.

[54] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of aspectj. In *European Conference on Object-Oriented Programming*, pages 327–354. Springer, 2001.

[55] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, pages 220–242. Springer, 1997.

[56] Johannes Kinder and Helmut Veith. Precise static analysis of untrusted driver binaries. In *Formal Methods in Computer Aided Design*, pages 43–50. IEEE, 2010.

[57] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[58] Gurvan Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Computer Security Foundations Symposium*, pages 218–232. IEEE, 2007.

[59] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David A Schmidt. Automata-based confidentiality monitoring. In *Annual Asian Computing Science Conference*, pages 75–89. Springer, 2006.

[60] Inyong Lee, Soonki Jeong, Sangsoo Yeo, and Jongsub Moon. A novel method for SQL injection attack detection based on removing SQL query attribute values. *Mathematical and Computer Modelling*, 55(1-2):58–68, 2012.

[61] 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.

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

[63] Jay Ligatti, Cagri Cetin, Shamaria Engram, Jean-Baptiste Subils, and Dmitry Goldgof. Coauthentication. In *Symposium on Applied Computing*, pages 1906–1915, 2019.

[64] V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX Security Symposium*, volume 14, pages 18–18, 2005.

[65] 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.

[66] Michael Martin, Benjamin Livshits, and Monica S Lam. Finding application errors and security flaws using PQL: a program query language. In *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 365–383. ACM, 2005.

[67] Leo A Meyerovich and Benjamin Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *IEEE Symposium on Security and Privacy*, pages 481–496, 2010.

[68] National Institute of Standards and Technology. Back to basics: Multi-factor authentication (MFA), November 2016. https://www.nist.gov/itl/tig/back-basics-multi-factor-authentication. Accessed 20 August 2019.

[69] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *ACM Symposium on Information, Computer and Communications Security*, pages 328–332, 2010.

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

[71] Lawrence O'Gorman. Comparing passwords, tokens, and biometrics for user authentication. *Proceedings of the IEEE*, 91(12):2021–2040, 2003.

[72] Hernan Palombo, Egor Dolzhenko, Jay Ligatti, and Hao Zheng. Stream-monitoring automata. In *International Conference on Software and Computer Applications*, pages 313–320, 2020.

[73] DongGook Park, Colin Boyd, and Sang-Jae Moon. Forward secrecy and its application to future mobile communications security. In *International Workshop on Public Key Cryptography*, pages 433–445. Springer, 2000.

[74] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[75] Srinivas Pinisetty, Ylies Falcone, Thierry Jéron, Hervé Marchand, Antoine Rollet, and Omer Landry Nguena Timo. Runtime enforcement of timed properties. In *International Conference on Runtime Verification*, pages 229–244. Springer, 2012.

[76] 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.

[77] Amir Pnueli. The temporal logic of programs. In *IEEE Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[78] US-CERT Publications. Security tip (st04-010) using caution with email attachments. https://www.us-cert.gov/ncas/tips/ST04-010, 2009. [Online; accessed 17-May-2019].

[79] Donald Ray and Jay Ligatti. A theory of gray security policies. In *European Symposium on Research in Computer Security*, pages 481–499. Springer, 2015.

[80] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[81] William K Robertson and Giovanni Vigna. Static enforcement of web application integrity through strong typing. In *USENIX Security Symposium*, volume 9, pages 283–298, 2009.

[82] Bruno PS Rocha, Mauro Conti, Sandro Etalle, and Bruno Crispo. Hybrid static-runtime information flow and declassification enforcement. *IEEE Transactions on Information Forensics and Security*, 8(8):1294–1305, 2013.

[83] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Computer Security Foundations Symposium*, pages 186–199. IEEE, 2010.

[84] Andrei Sabelfeld and Heiko Mantel. Static confidentiality enforcement for distributed programs. In *International Static Analysis Symposium*, pages 376–394. Springer, 2002.

[85] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

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

[87] Yonghee Shin and Laurie Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *International Symposium on Empirical Software Engineering and Measurement*, pages 315–317, 2008.

[88] Paritosh Shroff, Scott Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. In *Computer Security Foundations Symposium*, pages 203–217. IEEE, 2007.

[89] Michael Sipser. *Introduction to the Theory of Computation*. Cengage learning, 2012.

[90] Christian Skalka and Scott Smith. Static enforcement of security with types. *ACM International Conference on Functional Programming*, 35(9):34–45, 2000.

[91] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*, pages 1–25. Springer, 2008.

[92] US National Archives and Resource Administration. NARA file analyzer and metadata harvester. https://github.com/usnationalarchives/File-Analyzer, 2016.

[93] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010.

[94] Bart van Delft, Sebastian Hunt, and David Sands. Very static enforcement of dynamic policies. In *International Conference on Principles of Security and Trust*, pages 32–52. Springer, 2015.

[95] Jeffrey A Vaughan and Steve Zdancewic. A cryptographic decentralized label model. In *IEEE Symposium on Security and Privacy*, pages 192–206. IEEE, 2007.

[96] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Network and Distributed System Security Symposium*, 2007.

[97] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *Colloquium on Trees in Algebra and Programming*, pages 607–621. Springer, 1997.

[98] David Walker. A type system for expressive security policies. In *Symposium on Principles of Programming Languages*, pages 254–267. ACM, 2000.

[99] Westley Weimer and George C Necula. Finding and preventing run-time error handling mistakes. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 419–431, 2004.

[100] Thomas Y.C. Woo and Simon S. Lam. A semantic model for authentication protocols. In *IEEE Symposium on Security and Privacy*, pages 178–194, 1993.

Appendix A: Copyright Permissions

The following is permission to use the content of [63] and [36] in Chapter 4, and Chapters 1–3 and 5–7, respectively.

ACM Copyright and Audio/Video Release

Title of the Work: Coauthentication

Author/Presenter(s): Jay Ligatti:University of South Florida;Cagri Cetin:University of South Florida;Shamaria Engram:University of South Florida;Jean-Baptiste Subils:University of South Florida;Dmitry Goldgof:University of South Florida.
Type of material: Full Paper

Publication and/or Conference Name: SAC '19: Symposium on Applied Computing Proceedings

I. Copyright Transfer, Reserved Rights and Permitted Uses

* Your Copyright Transfer is conditional upon you agreeing to the terms set out below.

Copyright to the Work and to any supplemental files integral to the Work which are submitted with it for review and publication such as an extended proof, a PowerPoint outline, or appendices that may exceed a printed page limit, (including without limitation, the right to publish the Work in whole or in part in any and all forms of media, now or hereafter known) is hereby transferred to the ACM (for Government work, to the extent transferable) effective as of the date of this agreement, on the understanding that the Work has been accepted for publication by ACM.

Reserved Rights and Permitted Uses

(a) All rights and permissions the author has not granted to ACM are reserved to the Owner, including all other proprietary rights such as patent or trademark rights.

(b) Furthermore, notwithstanding the exclusive rights the Owner has granted to ACM, Owner shall have the right to do the following:

(i) Reuse any portion of the Work, without fee, in any future works written or edited by the Author, including books, lectures and presentations in any and all media.

(ii) Create a "Major Revision" which is wholly owned by the author

(iii) Post the Accepted Version of the Work on (1) the Author's home page, (2) the Owner's institutional repository, (3) any repository legally mandated by an agency funding the research on which the Work is based, and (4) any non-commercial repository or aggregation that does not duplicate ACM tables of contents, i.e., whose patterns of links do not substantially duplicate an ACM-copyrighted volume or issue. Non-commercial repositories are here understood as repositories owned by non-profit organizations that do not charge a fee for accessing deposited articles and that do not sell advertising or otherwise profit from serving articles.

(iv) Post an "Author-Izer" link enabling free downloads of the Version of Record in the ACM Digital Library on (1) the Author's home page or (2) the Owner's institutional repository;

(v) Prior to commencement of the ACM peer review process, post the version of the Work as submitted to ACM ("Submitted Version" or any earlier versions) to non-peer reviewed servers;

(vi) Make free distributions of the final published Version of Record internally to the Owner's employees, if applicable;

(vii) Make free distributions of the published Version of Record for Classroom and Personal Use;

(viii) Bundle the Work in any of Owner's software distributions; and

(ix) Use any Auxiliary Material independent from the Work. (x) If your paper is withdrawn before it is

published in the ACM Digital Library, the rights revert back to the author(s).

When preparing your paper for submission using the ACM TeX templates, the rights and permissions information and the bibliographic strip must appear on the lower left hand portion of the first page.

The new ACM Consolidated TeX template Version 1.3 and above automatically creates and positions these text blocks for you based on the code snippet which is system-generated based on your rights management choice and this particular conference.

Please copy and paste the following code snippet into your TeX file between \begin{document} and \maketitle, either after or before CCS codes.

```
\copyrightyear{2019}
\acmYear{2019}
\setcopyright{acmcopyright}
\acmConference[SAC '19]{The 34th ACM/SIGAPP Symposium on Applied
Computing}{April 8--12, 2019}{Limassol, Cyprus}
\acmBooktitle{The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19),
April 8--12, 2019, Limassol, Cyprus}
\acmPrice{15.00}
\acmDOI{10.1145/3297280.3297466}
\acmISBN{978-1-4503-5933-7/19/04}
```

ACM TeX template .cls version 2.8, automatically creates and positions these text blocks for you based on the code snippet which is system-generated based on your rights management choice and this particular conference.
Please copy and paste the following code snippet into your TeX file between \begin{document} and \maketitle, either after or before CCS codes.

```
\CopyrightYear{2019}
\setcopyright{acmcopyright}
\conferenceinfo{SAC '19,}{April 8--12, 2019, Limassol, Cyprus}
\isbn{978-1-4503-5933-7/19/04}\acmPrice{$15.00}
\doi{https://doi.org/10.1145/3297280.3297466}
```

If you are using the ACM Microsoft Word template, or still using an older version of the ACM TeX template, or the current versions of the ACM SIGCHI, SIGGRAPH, or SIGPLAN TeX templates, you must copy and paste the following text block into your document as per the instructions provided with the templates you are using:

SAC '19, April 8–12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3297466

NOTE: Make sure to include your article's DOI as part of the bibstrip data; DOIs will be registered and become active shortly after publication in the ACM Digital Library. Once you have your camera ready copy ready, please send your source files and PDF to your event contact for processing.

☑ A. Assent to Assignment. I hereby represent and warrant that I am the sole owner (or authorized agent of the copyright owner(s)), with the exception of third party materials detailed in section III below. I have obtained permission for any third-party material included in the Work.

☐ B. Declaration for Government Work. I am an employee of the National Government of my country and my Government claims rights to this work, or it is not copyrightable (Government work is classified as Public Domain in U.S. only)

Are any of the co-authors, employees or contractors of a National Government? ○ Yes ◉ No

---

II. Permission For Conference Recording and Distribution

* Your Audio/Video Release is conditional upon you agreeing to the terms set out below.

I hereby grant permission for ACM to include my name, likeness, presentation and comments in any and all forms, for the Conference and/or Publication.

I further grant permission for ACM to record and/or transcribe and reproduce my presentation as part of the ACM Digital Library, and to distribute the same for sale in complete or partial form as part of an ACM product on CD-ROM, DVD, webcast, USB device, streaming video or any other media format now or hereafter known.

I understand that my presentation will not be sold separately as a stand-alone product without my direct consent. Accordingly, I give ACM the right to use my image, voice, pronouncements, likeness, and my name, and any biographical material submitted by me, in connection with the Conference and/or Publication, whether used in excerpts or in full, for distribution described above and for any associated advertising or exhibition.

Do you agree to the above Audio/Video Release? ◉ Yes ○ No

III. Auxiliary Material

Do you have any Auxiliary Materials? ○ Yes ◉ No

IV. Third Party Materials

In the event that any materials used in my presentation or Auxiliary Materials contain the work of third-party individuals or organizations (including copyrighted music or movie excerpts or anything not owned by me), I understand that it is my responsibility to secure any necessary permissions and/or licenses for print and/or digital publication, and cite or attach them below.

◉ We/I have not used third-party material.
○ We/I have used third-party materials and have necessary permissions.

V. Artistic Images

If your paper includes images that were created for any purpose other than this paper and to which you or your employer claim copyright, you must complete Part V and be sure to include a notice of copyright with each such image in the paper.

(●) We/I do not have any artistic images.
( ) We/I have any artistic images.

VI. Representations, Warranties and Covenants

The undersigned hereby represents, warrants and covenants as follows:

(a) Owner is the sole owner or authorized agent of Owner(s) of the Work;

(b) The undersigned is authorized to enter into this Agreement and grant the rights included in this license to ACM;

(c) The Work is original and does not infringe the rights of any third party; all permissions for use of third-party materials consistent in scope and duration with the rights granted to ACM have been obtained, copies of such permissions have been provided to ACM, and the Work as submitted to ACM clearly and accurately indicates the credit to the proprietors of any such third-party materials (including any applicable copyright notice), or will be revised to indicate such credit;

(d) The Work has not been published except for informal postings on non-peer reviewed servers, and Owner covenants to use best efforts to place ACM DOI pointers on any such prior postings;

(e) The Auxiliary Materials, if any, contain no malicious code, virus, trojan horse or other software routines or hardware components designed to permit unauthorized access or to disable, erase or otherwise harm any computer systems or software; and

(f) The Artistic Images, if any, are clearly and accurately noted as such (including any applicable copyright notice) in the Submitted Version.

☑ I agree to the Representations, Warranties and Covenants

Funding Agents

1. Cyber Florida award number(s): Seed-Grant-2017
2. National Science Foundation award number(s): CNS-1527144

DATE: **12/03/2018** sent to ligatti@usf.edu at **16:12:23**

IEEE COPYRIGHT AND CONSENT FORM

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE. Please read the form carefully and keep a copy for your files.

Through the Lens of Code Granularity: A Unified Approach to Security Policy Enforcement
Ms. Shamaria B Engram and Prof. Jay Ligatti
2020 IEEE Conference on Application, Information and Network Security (AINS)

COPYRIGHT TRANSFER

The undersigned hereby assigns to The Institute of Electrical and Electronics Engineers, Incorporated (the "IEEE") all rights under copyright that may exist in and to: (a) the Work, including any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work; and (b) any associated written or multimedia components or other enhancements accompanying the Work.

GENERAL TERMS

1. The undersigned represents that he/she has the power and authority to make and execute this form.
2. The undersigned agrees to indemnify and hold harmless the IEEE from any damage or expense that may arise in the event of a breach of any of the warranties set forth above.
3. The undersigned agrees that publication with IEEE is subject to the policies and procedures of the IEEE PSPB Operations Manual.
4. In the event the above work is not accepted and published by the IEEE or is withdrawn by the author(s) before acceptance by the IEEE, the foregoing copyright transfer shall be null and void. In this case, IEEE will retain a copy of the manuscript for internal administrative/record-keeping purposes.
5. For jointly authored Works, all joint authors should sign, or one of the authors should sign as authorized agent for the others.
6. The author hereby warrants that the Work and Presentation (collectively, the "Materials") are original and that he/she is the author of the Materials. To the extent the Materials incorporate text passages, figures, data or other material from the works of others, the author has obtained any necessary permissions. Where necessary, the author has obtained all third party permissions and consents to grant the license above and has provided copies of such permissions and consents to IEEE

You have indicated that you DO wish to have video/audio recordings made of your conference presentation under terms and conditions set forth in "Consent and Release."

CONSENT AND RELEASE

1. In the event the author makes a presentation based upon the Work at a conference hosted or sponsored in whole or in part by the IEEE, the author, in consideration for his/her participation in the conference, hereby grants the IEEE the unlimited, worldwide, irrevocable permission to use, distribute, publish, license, exhibit, record, digitize, broadcast, reproduce and archive, in any format or medium, whether now known or hereafter developed: (a) his/her presentation and comments at the conference; (b) any written materials or multimedia files used in connection with his/her presentation; and (c) any recorded interviews of him/her (collectively, the "Presentation"). The permission granted includes the transcription and reproduction of the Presentation for inclusion in products sold or distributed by IEEE and live or recorded broadcast of the Presentation during or after the conference.
2. In connection with the permission granted in Section 1, the author hereby grants IEEE the unlimited, worldwide, irrevocable right to use his/her name, picture, likeness, voice and biographical information as part of the advertisement, distribution and sale of products incorporating the Work or Presentation, and releases IEEE from any claim based on right of privacy or publicity.

BY TYPING IN YOUR FULL NAME BELOW AND CLICKING THE SUBMIT BUTTON, YOU CERTIFY THAT SUCH ACTION CONSTITUTES YOUR ELECTRONIC SIGNATURE TO THIS FORM IN ACCORDANCE WITH UNITED STATES LAW, WHICH AUTHORIZES ELECTRONIC SIGNATURE BY AUTHENTICATED REQUEST FROM A USER OVER THE INTERNET AS A VALID SUBSTITUTE FOR A WRITTEN SIGNATURE.

Shamaria B. Engram	20-10-2020
Signature	**Date (dd-mm-yyyy)**

Information for Authors

AUTHOR RESPONSIBILITIES

The IEEE distributes its technical publications throughout the world and wants to ensure that the material submitted to its publications is properly available to the readership of those publications. Authors must ensure that their Work meets the requirements as stated in section 8.2.1 of the IEEE PSPB Operations Manual, including provisions covering originality, authorship, author responsibilities and author misconduct. More information on IEEE's publishing policies may be found at http://www.ieee.org/publications_standards/publications/rights/authorrightsresponsibilities.html Authors are advised especially of IEEE PSPB Operations Manual section 8.2.1.B12: "It is the responsibility of the authors, not the IEEE, to determine whether disclosure of their material requires the prior consent of other parties and, if so, to obtain it." Authors are also advised of IEEE PSPB Operations Manual section 8.1.1B: "Statements and opinions given in work published by the IEEE are the expression of the authors."

RETAINED RIGHTS/TERMS AND CONDITIONS

- Authors/employers retain all proprietary rights in any process, procedure, or article of manufacture described in the Work.
- Authors/employers may reproduce or authorize others to reproduce the Work, material extracted verbatim from the Work, or derivative works for the author's personal use or for company use, provided that the source and the IEEE copyright notice are indicated, the copies are not used in any way that implies IEEE endorsement of a product or service of any employer, and the copies themselves are not offered for sale.
- Although authors are permitted to re-use all or portions of the Work in other works, this does not include granting third-party requests for reprinting, republishing, or other types of re-use.The IEEE Intellectual Property Rights office must handle all such third-party requests.
- Authors whose work was performed under a grant from a government funding agency are free to fulfill any deposit mandates from that funding agency.

AUTHOR ONLINE USE

- **Personal Servers**. Authors and/or their employers shall have the right to post the accepted version of IEEE-copyrighted articles on their own personal servers or the servers of their institutions or employers without permission from IEEE, provided that the posted version includes a prominently displayed IEEE copyright notice and, when published, a full citation to the original IEEE publication, including a link to the article abstract in IEEE Xplore. Authors shall not post the final, published versions of their papers.
- **Classroom or Internal Training Use.** An author is expressly permitted to post any portion of the accepted version of his/her own IEEE-copyrighted articles on the author's personal web site or the servers of the author's institution or company in connection with the author's teaching, training, or work responsibilities, provided that the appropriate copyright, credit, and reuse notices appear prominently with the posted material. Examples of permitted uses are lecture materials, course packs, e-reserves, conference presentations, or in-house training courses.
- **Electronic Preprints.** Before submitting an article to an IEEE publication, authors frequently post their manuscripts to their own web site, their employer's site, or to another server that invites constructive comment from colleagues. Upon submission of an article to IEEE, an author is required to transfer copyright in the article to IEEE, and the author must update any previously posted version of the article with a prominently displayed IEEE copyright notice. Upon publication of an article by the IEEE, the author must replace any previously posted electronic versions of the article with either (1) the full citation to the