Stream-Monitoring Automata

Hernan Palombo
hpalombo@usf.edu
University of South Florida

Egor Dolzhenko
egor.dolzhenko@gmail.com
Illumina

Jay Ligatti
ligatti@usf.edu
University of South Florida

Hao Zheng
haozheng@usf.edu
University of South Florida

ABSTRACT

Over the past nearly twenty years, numerous formal models of enforcement and runtime monitors have been investigated. This paper takes the lessons learned from earlier models and proffers a new general model of runtime enforcement that is more suitable for modeling security mechanisms that operate over infinite event streams. The new model, called *Stream-Monitoring Automata (SMAs)*, enables the constraints and analyses of interest in previous models to be encoded, and overcomes several shortcomings of existing models with respect to expressiveness. SMAs capture the practical abilities of mechanisms to monitor infinite event streams, execute even in the absence of event inputs, enforce non-safety policies, and operate an enforcement model in which extraneous constraints such as transparency and uncontrollable events may be specified as meta-policies.

KEYWORDS

Enforceability Theory; Monitoring; Runtime Enforcement; Security Automata; Models of Enforcement

1 INTRODUCTION

Runtime enforcement mechanisms or *monitors* oversee untrusted systems behavior, and act on their inputs and outputs to ensure they obey desired policies. Due to the pervasiveness of security threats and attacks, security monitors have become ubiquitous. A mismatch between the security goals and the actual policies being enforced can be catastrophic [16, 19, 25]. Thus, rigorous models of monitors and analysis of the policies that they can enforce are of significant importance.

In recent years, there has been a notable shift towards a model of computation in which programs are producers and consumers of infinite data streams [2, 27]. Some example embodiments include web applications, cloud services, operating systems, and embedded microcontrollers in self-powered devices. The new paradigm's popularity is in part due to the rise of cloud infrastructure providers that offer low-cost storage and processing resources on demand and the reduced production costs of computing devices and their consequent proliferation, which are leading to the exponential increase in the amounts of digital data constantly flowing worldwide [1].

Although many models of runtime enforcement have been proposed, they sometimes omit important details or make assumptions which limit their applicability within the new computational paradigm. For example, many earlier enforcement models assume that monitors operate on possibly finite input sequences [8, 10, 15, 17, 21, 31, 32], that they are only enabled on input events [10, 13, 21, 28, 29, 31], or that they are always able to stop the underlying targets being monitored [10, 13, 21, 31]. Often, these and other constraints limit the models' enforcement power, e.g. to only enforce a subset of safety properties [15, 24, 28, 31, 32], or allow them to enforce exceptional policies in sometimes impractical scenarios [21] (as discussed in [13]). Remarkably, these limitations make many previous frameworks unsuitable for modeling monitors of black-box targets that process infinite event streams, which are now widespread.

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

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

- Runtime mechanisms are modeled as Stream-Monitoring Automata (SMAs), transition systems that mediate events in infinite input streams. A distinctive feature of SMAs is their ability to affect the targets being monitored even in the absence of input events. Another benefit is their simple operational semantics, which are defined with one rule—and thus are amenable for formal analysis.
- The properties that SMAs can soundly enforce are characterized as *game* properties, which are explained by making an analogy between property enforcement and 2-player infinite games.

ICSCA'20, February 2020, Langkawi, Malaysia

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

– Relationships established with safety and liveness properties reveal that SMAs can enforce some non-safety properties that were previously considered unenforceable. Conversely, SMAs cannot enforce some safety properties that were previously considered enforceable. Specifically, SMAs cannot enforce some properties requiring that the monitor receives certain input events.
– The model's extensibility is shown by encoding different definitions of enforcement, e.g. transparency and uncontrollable events [8] as meta-policies, i.e. policies about policies. The main advantage of meta-policies is that special constraints about how traces must be transformed can be specified, avoiding the need to hardcode them into the model.

2 BACKGROUND AND NOTATION

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

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

A key difference between SMAs and previous models is the explicit characterization of nil events. Nil events constitute a void result from a monitor's proactive sampling of inputs. In other words, a nil event is an empty or blank observation. In all previous security automata models of which we are aware of, monitors are reactive, i.e. the monitors' computations are blocked when the monitored target provides no inputs to the monitor [8, 10, 13, 15, 21, 24, 28, 29, 31, 32]. In contrast, SMAs are proactive, sampling inputs and possibly making transitions even on nil events.

A stream is an infinite sequence of events $e_0e_1e_2\ldots$ such that $\forall i \in \mathbb{N} : e_i \in E$. The set of streams is denoted by E^ω. Similarly, the set of finite sequences of events is denoted by E^*. For any set S, we write S_f to mean the set is finite. Metavariable i (o) ranges over E and is used to denote an event that is input to (output from) the monitor. The wildcard symbol _ is used to signify any event in E that is not bound to a variable.

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

3 STREAM-MONITORING AUTOMATA

This section defines SMAs and their operational semantics.

Definition 3.1. A Stream-Monitoring Automaton (SMA) M is a tuple (E, Q, q_0, δ), where E is a recursively enumerable set of events such that $\epsilon \in E$, Q is a recursively enumerable set of automaton states, $q_0 \in Q$ is an initial state, and $\delta : Q \times E \to Q \times E$ is a computable and total transition function.

An important feature that distinguishes SMAs from other automata-based transducers (e.g. Mealy machines [26]) is that SMAs may have a countably infinite set of states. It is common for security automata frameworks to allow states to be infinite [13, 21, 31]. Yet, some previous works have considered the effects of limiting the number of states in security automata [15, 32].

Example 3.2 (Filter-and-sanitize SMA). Consider an SMA M_1 that monitors a web server receiving queries from clients. M_1 filters queries from bad sources according to the computable function $allow : E \to \{0, 1\}$, and sanitizes the contents of the queries using the computable function $sanitize : E \to E$. M_1's transition function is defined by $\delta(q, i) = (q, sanitize(i))$ if $allow(i)$; otherwise $\delta(q, i) = (q, \epsilon)$.

Example 3.3 (Log SMA). Consider an SMA M_2 that logs all activity observed by issuing a store operation $st(f) \in E$ where f is a file generated using function $log : E^* \to \{f \mid f \text{ is a binary file}\}$. M_2's set of states is $Q = \{v \mid v \in E^*\}$, and its initial state is the empty sequence. Let $|v|$ be the length of sequence v and k a user-defined parameter that specifies the maximum log size. M_2's transition function is defined by $\delta(v, i) = (vi, \epsilon)$ if $|vi| < k$; otherwise $\delta(v, i) = (i, st(log(v)))$.

Operational Semantics

An *exchange* is a pair of events $\langle i, o\rangle$, consisting of an input event $i \in E$ (i.e. the event input to the SMA) and an output event $o \in E$ (i.e. the event output from the SMA). The set of exchanges over the input and output events is denoted by $\mathcal{E}$, i.e. $\mathcal{E} = E \times E$. The operational semantics of SMAs is defined with a labeled single-step judgment whose form is $q \xrightarrow{\langle i,o\rangle}_M q'$, shown in Figure 1, where q, $q' \in Q$ are configurations (i.e. states) of an SMA. This judgment indicates that the SMA M takes a single step from configuration q to q' while producing the exchange $\langle i, o\rangle$. Event i is an input event as observed by the monitor, and there are no explicit assumptions on it. In other words, a target can produce any $i \neq \epsilon$ at any time-step of the target's execution, or $i = \epsilon$. By default, SMAs can step on nil input events and output nil events. More restrictive SMAs can be specified with meta-policies (Section 6).

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

Definition 3.4. Given an SMA $M = (E, Q, q_0, \delta)$, the *language* of M is

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

The operational semantics of SMAs have three noteworthy features.

– Because SMAs' transition functions are total and input events may be nil (i.e. ϵ), SMAs can output events even when the monitored target is inactive. Formally, this means that for any state q, $\delta(q, \epsilon)$ is defined. SMAs that produce

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

Figure 1: SMAs' operational semantics. Symbol i (o) denotes an input (output) event, and q, q' are states of SMA M.

outputs in the absence of inputs model monitors that initiate communications, which are common in practice (e.g. monitors may publish information to other nodes on a network), yet it was not considered in previous frameworks [8, 10, 13, 15, 21, 28, 29, 31, 32].
- Because SMAs read inputs and produce outputs incessantly (since their transition function is computable), SMAs do not have the power to stop traces. This model of traces is reasonable in the context of stream-generating black-box targets where enforcement mechanisms may not be able to stop the executions of the targets being monitored and may always continue to receive inputs. In contrast, previous models assumed that monitors had the power to stop the targets being monitored [10, 13, 21, 31].
- As SMA traces include monitor inputs (as similarly done in [13, 24, 28]), SMAs are useful to reason about security policies that specify constraints between monitor inputs and outputs. Because many previous models did not include monitor inputs on traces [8, 10, 15, 21, 29, 31, 32], requirements about how monitors transformed certain inputs, e.g. filtering policies, were often left out of previous analyses.

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

4 SMA-BASED ENFORCEMENT

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

4.1 Properties

This paper adopts standard definitions of policies and properties [11, 21, 31]. A *policy* is a predicate over sets of executions. Some policies are properties.

Definition 4.1. A policy P is a property if there exists a decidable predicate p over $\mathcal{E}^\omega$ such that $\forall X \subseteq \mathcal{E}^\omega : (P(X) \iff \forall x \in X : p(x))$.

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

4.2 Enforcement

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

Definition 4.2. An SMA M soundly enforces a property P if $\mathcal{L}(M) \subseteq P$.

Definition 4.2 is significantly simpler and more flexible than previous definitions of enforcement because it does not hardcode specific requirements or constraints (e.g. transparency, uncontrollable events), which can be encoded as meta-policies (Section 6).

Example 4.3 (Filter-and-sanitize Property). SMA M_1 soundly enforces a *filter-and-sanitize* property which asserts that only sanitized queries from allowed sources are output, i.e.

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

Example 4.4 (Log Property). SMA M_2 soundly enforces a *log* property which asserts that input events are eventually logged, i.e.

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

4.3 Enforceable Game Properties

The set of properties that are soundly enforceable by SMAs can be characterized by new definitions that draw connections to game theory. Intuitively, property enforcement can be interpreted as a two-player game, where the first player is a target and the second player is a monitor, played in a sequence of rounds (traces). Each round has an infinite sequence of hands (exchanges) that are the concatenation of player one's hand (target events that are input to the monitor) and player two's hand (monitor outputs). A property describes the "winning rules" for the monitor. That is, a property specifies all possible rounds (traces) which dictate that player two wins the game (produces satisfying traces). A property contains a winning-strategy if a plan exists for a monitor to win no matter how the target plays. Precisely, let $i, o \in E$ denote the input and output of an exchange $\langle i, o\rangle$ in an execution $x \in \mathcal{E}^\omega$. A property P contains a winning strategy if $\forall i_0 : \exists o_0 : \forall i_1 : \exists o_1 : \ldots : p(\langle i_0, o_0\rangle\langle i_1, o_1\rangle\ldots)$. Properties with winning strategies can be enforced if there exists a function that can compute a valid output at every exchange.

Definition 4.5. A property P is a game property if there exists a computable function $f : E^* \to E$ such that $\forall i_0, i_1, \ldots \in E : p(\langle i_0, f(i_0)\rangle\langle i_1, f(i_0 i_1)\rangle\ldots)$.

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

Proposition 4.6. $\forall P, P' \subseteq \mathcal{E}^\omega : (P \subseteq P') \wedge P \in GP \implies P' \in GP$.

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

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

Theorem 4.7. *A property P is soundly enforceable by an SMA iff P is a game property.*

Proof. Let M be an SMA that soundly enforces some property P, i.e. $\mathcal{L}(M) \subseteq P$. Because M's transition function is total and computable, it is defined for all possible inputs and will always output some event (possibly ϵ). Let $f : E^* \to E$ be a function that takes any finite sequence of inputs and runs M with those inputs to compute a valid output. Because the traces that form by starting from the empty sequence and concatenating every pair $\langle i_0, f(i_0)\rangle, \langle i_1, f(i_0 i_1)\rangle \ldots$ ad infinitum satisfy P (because $\mathcal{L}(M) \subseteq P$), $\mathcal{L}(M)$ is a game property. Further, because $\mathcal{L}(M) \subseteq P$, by Proposition 4.6, P is a game property, i.e. $P \in GP$.

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

□

5 A HIERARCHY OF PROPERTIES WITH EXAMPLES

Relationships can be established among game and traditional formulations of safety and liveness properties. Recall that safety properties, as first defined by Lamport [20] and later by Alpern and Schneider [31], proscribe "bad things". Let $\prec$ be the strict prefix operator, i.e. $\forall x \in \mathcal{E}^* : \forall y \in \mathcal{E}^\omega : x \prec y \iff \exists x' \in \mathcal{E}^\omega : xx' = y$. The $\succ$ operator is defined symmetrically. A property P is safety if

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

On the other hand, liveness properties prescribe "good things", i.e. a property P is liveness [4] if

$$\forall x \in \mathcal{E}^* : \exists y \succ x : p(y).$$

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

Figure 2 shows a hierarchy of properties that includes safety, liveness, and game properties, and examples in each class. As expected, SMAs can enforce a myriad of safety properties that are enforceable in other models, including those for integrity and access-control (e.g. classical *no-send-after-read* [31] and *authenticated-login* [21] properties), and filtering properties that specify requirements between monitor inputs and outputs (P_1 in Example 4.3) [13].

Further, SMAs can soundly enforce some additional properties.

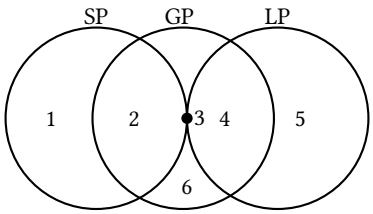
- SMAs can enforce liveness properties that specify requirements about a monitor's eventual obligation or sequence of obligations, e.g. the *log* property P_2 in Example 4.4, the *eventual-monitor-activity* property $\mathcal{E}^\omega \setminus \langle \_, \epsilon\rangle^\omega$, or the *eventual-monitor-inactivity* property $\ldots\langle \_, \epsilon\rangle^\omega$. Note that, in many cases, there may be multiple ways to enforce non-safety properties. For example, for *eventual-obligation* properties such as $\ldots\langle \_, o\rangle\ldots$ (where o is some arbitrary output event), a monitor may choose to output o right away or after some number of exchanges. The alternatives should be evaluated in context, and the choice may be intentionally left to the implementer. On the other hand, this flexibility can be removed by incorporating specific requirements as meta-policies that, e.g., require obligations to be output within a certain number of exchanges.
- SMAs can enforce some non-safety, non-liveness properties that are the conjunction of some safety and liveness. For example, consider a garbage collector in a programming language's runtime system. Garbage collection demands that all objects that are not referenced by other objects are eventually collected (liveness requirement), and that objects that are referenced by other objects are not collected (safety requirement). Let $collect(r) \in E$ be a memory collection event that effectively frees memory reference r in a set of references R, and $collectable : E^* \to 2^R_f$ a computable function that takes an input sequence and computes a finite set of references that are out of scope and can be marked for collection. The *garbage-collection* property is

$$\begin{aligned}\{&\langle i_0, o_0\rangle\langle i_1, o_1\rangle\ldots \mid \\ &(\forall n \in \mathbb{N} : o_n = collect(r) \implies \\ &\quad \exists m \le n : r \in collectable(i_m \ldots i_n)) \wedge \\ &(\forall m, n \in \mathbb{N} : \forall r \in collectable(i_m \ldots i_n) : \\ &\quad \exists p \ge n : o_p = collect(r))\}.\end{aligned}$$

Lastly, some properties with specific input constraints are not enforceable, regardless of being safety, liveness, or neither. Broadly speaking, safety properties that only allow some inputs to appear on traces (e.g. the *always-nil-inputs* property $\langle \epsilon, \_\rangle^\omega$) cannot be enforced because monitors have no control over the inputs received. These properties are enforceable in previous models that rely on the assumption that monitors are always able to stop target executions (e.g. [31]), but that assumption is impractical in the context of monitors of black-box targets operating on streams of events. Further, liveness properties that require particular monitor inputs cannot be enforced either because those inputs may not ever come. For example, SMAs cannot guarantee that targets will eventually cease all activity (i.e. ϵ monitor inputs ad infinitum) or conversely always be active. Thus, these *target-activity* properties are not enforceable by SMAs.

6 ADDITIONAL CATEGORIES OF POLICIES

Models of runtime monitors often need to accommodate extraneous enforcement constraints to be applicable in practical scenarios, e.g. monitors may be required to enforce properties transparently, i.e. by outputting valid inputs verbatim, or deal with uncontrollable events that cannot be modified. SMAs adopt a definition of enforcement that is simpler than those considered by other models and generalize extraneous constraints as meta-policies, i.e. policies about policies. Traditionally, these constraints have been hard-coded into the definitions of enforcement, limiting the models' applicability (e.g. [29]). Because meta-policies can specify arbitrary constraints about input and output events, they are more fine-grained than previous encodings. Some commonly studied constraints are discussed next.



Example Properties

1.	always-nil-inputs	$\langle \epsilon, \_ \rangle^\omega$
2.	filter-and-sanitize	P_1 in Example 4.3
3.	allow-all	$\mathcal{E}^\omega$
4.	log	P_2 in Example 4.4
	eventual-monitor-activity	$\mathcal{E}^\omega \setminus \langle \_, \epsilon \rangle^\omega$
5.	target-inactivity	$\ldots\langle \epsilon, \_ \rangle^\omega$
6.	garbage-collection	$\{\langle i_0, o_0 \rangle \langle i_1, o_1 \rangle \ldots \mid$
		$(\forall n : o_n = collect(r) \Rightarrow \exists m \leq n : r \in collectable(i_m \ldots i_n)) \wedge$
		$(\forall m, n : \forall r \in collectable(i_m \ldots i_n) : \exists p \geq n : o_p = collect(r))\}$

Figure 2: A hierarchy of properties with examples. SMAs can enforce properties in GP. Variables $m, n, p \in \mathbb{N}$.

6.1 Transparency Policies

First, consider transparency [22], which requires that valid sequences input to a monitor not be modified. A transparency requirement may be specified to constrain the enforcement power of a monitor so that the input events from the target are output verbatim if they do not violate the property under consideration.

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

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

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

Corollary 6.1. *Let TP be the set of properties that are enforceable by SMAs and satisfy the transparency requirement. Then, $TP = GP \cap \{P \mid P \subseteq \mathcal{E}_t^\omega\}$.*

Proof. ($\subseteq$) Let M be an SMA that enforces a property P that satisfies the transparency requirement for the set of input sequences V, i.e. $\mathcal{L}(M) \subseteq P$ and $\forall x \in P : x \in \mathcal{E}_t^\omega$. Theorem 4.7 states that enforceable properties are game, so $P \in GP$. Because all the traces in P are in $\mathcal{E}_t^\omega$, $P \subseteq \mathcal{E}_t^\omega$. Thus, $P \in GP \cap \{P \mid P \subseteq \mathcal{E}_t^\omega\}$.

($\supseteq$) Let P be a property in $GP \cap \{P \mid P \subseteq \mathcal{E}_t^\omega\}$. Theorem 4.7 shows how to construct an SMA M that enforces any property in GP. Moreover, because $P \subseteq \mathcal{E}_t^\omega$, all the traces in P satisfy the transparency requirement, i.e. $\forall x \in P : x \in \mathcal{E}_t^\omega$. Therefore, $P \in TP$. □

6.2 Uncontrollable-Events Policies

Uncontrollable or *only observable* events cannot be mediated by monitors [8]; a typical example are clock ticks. On the other hand, events that can be inserted or suppressed by monitors are called *controllable*. Let (E_u, E_c) be a partition of events where E_u, E_c are sets of uncontrollable and controllable events respectively. The uncontrollable events constraint requires that events in E_u that are input to the monitor are not be suppressed, and that events in E_u are not output if they are not in the input. Let $\mathcal{E}_u^\omega$ be the set of all traces constructed as the sequences of exchanges that satisfy the uncontrollable events constraint, i.e. $\mathcal{E}_u = \{\langle i, o \rangle \mid i \in E_u \vee o \in E_u \implies i = o\}$. Properties about traces with uncontrollable events that are enforceable by SMAs are exactly the game properties that satisfy the uncontrollable events constraint.

Corollary 6.2. *Let UP be the set of properties about traces with uncontrollable events that are enforceable by SMAs. Then, $UP = GP \cap \{P \mid P \subseteq \mathcal{E}_u^\omega\}$.*

Proof. ($\subseteq$) Let M be an SMA that enforces a property $P \in UP$ that satisfies the uncontrollable events constraint, i.e. $\mathcal{L}(M) \subseteq P$ and $\forall x \in P : x \in \mathcal{E}_u^\omega$. By Theorem 4.7, $P \in GP$. Because all the traces in P are in $\mathcal{E}_u^\omega$, $P \subseteq \mathcal{E}_u^\omega$. Thus, $P \in GP \cap \{P \mid P \subseteq \mathcal{E}_u^\omega\}$.

($\supseteq$) Let P be a property in $GP \cap \{P \mid P \subseteq \mathcal{E}_u^\omega\}$. By Theorem 4.7, P is enforceable. Since $P \subseteq \mathcal{E}_u^\omega$, all the traces in P satisfy the uncontrollable events constraint, i.e. $\forall x \in P : x \in \mathcal{E}_u^\omega$. Thus, $P \in UP$.

□

6.3 Partially Controllable-Events Policies

Events are called partially controllable when they can be inserted but not suppressed, or suppressed but not inserted [17]. For example, a monitor may be able to drop but not inject events encrypted

with a key that is unknown to the monitor. To encode partially controllable events, let (E_u, E_d, E_i, E_c) be a partition of E such that:

- E_u contains uncontrollable events,
- E_d contains events that can be suppressed but not inserted,
- E_i contains events that can be inserted but not suppressed,
- E_c contains events that can be fully controlled (i.e. suppressed *and* inserted).

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

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

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

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

Corollary 6.3. *Let PCP be the set of properties about traces with partially controllable events that are enforceable by SMAs. Then,* $PCP = GP \cap \{P \mid P \subseteq \mathcal{E}^{\omega}_{pc}\}$.

Proof. ($\subseteq$) Let M be an SMA that enforces a property $P \in PCP$ that satisfies the partially-controllable events constraint, i.e. $\mathcal{L}(M) \subseteq P$ and $\forall x \in P : x \in \mathcal{E}^{\omega}_{pc}$. By Theorem 4.7, $P \in GP$. Because all the traces in P are in $\mathcal{E}^{\omega}_{pc}$, $P \subseteq \mathcal{E}^{\omega}_{pc}$. Thus, $P \in GP \cap \{P \mid P \subseteq \mathcal{E}^{\omega}_{pc}\}$.

($\supseteq$) Let P be a property in $GP \cap \{P \mid P \subseteq \mathcal{E}^{\omega}_{pc}\}$. By Theorem 4.7, P is enforceable. Since $P \subseteq \mathcal{E}^{\omega}_{pc}$, all the traces in P satisfy the partially-controllable events constraint, i.e. $\forall x \in P : x \in \mathcal{E}^{\omega}_{pc}$. Thus, $P \in PCP$.

□

7 RELATED WORKS

Many models of runtime enforcement have been proposed. This section reviews related works with a focus on models of runtime monitors for black-box targets that include enforceability results.

Schneider introduced security automata [31], an early model of monitors that can only enforce a subset of safety properties by truncating executions. Ligatti et al. introduced edit automata [21], a more powerful model that can insert or suppress events, and enforce some non-safety properties. Bielova and Massacci [10] introduced a fine-grained classification of edit automata and analyzed their enforcement capabilities. When comparing SMAs to security and edit automata, SMAs are closer to edit automata because they can not only truncate but also transform executions. Still, edit automata differ from SMAs in that the former are reactive, i.e. they are only enabled on target events. Further, edit automata only include monitor outputs on traces, which limits them to specification of target-centric policies [13]. In contrast, traces in the SMA model also include monitor inputs so arbitrary relationships between monitor inputs and outputs can be expressed.

Other models have also included monitor inputs on traces. Ngo et al. [28] modeled mechanisms on black-box reactive programs, which cannot delay outputs hence are limited to safety enforcement. Mallios et al [24] proposed a runtime enforcement framework based on I/O automata [23], yet their monitors are also limited to enforce safety properties.

Some works have studied the enforcement capabilities of monitors under extraneous constraints. Basin et al. [8] proposed a distinction between system actions that are controllable by an enforcement mechanism and those that are only observable. Khoury et al. [17] further distinguished sets of events that can only be partially controlled, i.e. events that can only be inserted or suppressed. In the SMA model, these extraneous constraints can be encoded as meta-policies.

A few models [13, 18] have attempted to generalize extraneous constraints and incorporate them into the definitions of policies. Khoury and Tawbi [18] analyzed monitors that preserve some equivalence relation between its inputs and outputs, give examples of meaningful relations and identify the policies that are enforceable with their use. Dolzhenko et al. [13] demonstrated that constraints such as transparency and uncontrollable events define subsets of the properties that are enforceable by MRAs. Yet, MRAs definition of precise enforcement imposes an additional completeness requirement, i.e. to enforce a property a monitor is required to produce *all* the traces in that property. In the MRA model, traces may be finite and enforceable properties are prefix-closed, so the completeness requirement is useful to rule out definitions of MRAs that could artificially enforce any property by having transition functions that were undefined for all inputs. In other words, completeness guarantees that MRAs will produce at least some trace. In contrast, because SMAs read inputs from infinite streams and always produce some output, SMAs will always produce some trace and the completeness requirement is no longer needed. Further, completeness in the sense of MRAs is a rare and often impractical requirement for the type of monitors that SMAs model. Thus, SMAs generalize MRAs' definition of enforcement by only requiring soundness, which is a widely accepted and more reasonable requirement in practice.

Others have studied monitors with memory-limitations. Fong [15] showed that security automata with a shallow-history can enforce many practical properties. Talhi et al. [32] studied bounded-history security and edit automata and showed they can enforce some locally testable properties [9]. Renard et al. [30] extended work on uncontrollable events by studying enforcement of timed properties, and showed how to synthesize runtime enforcement mechanisms at two levels of abstraction to facilitate their design and implementation. Bounded-memory SMAs and enforcement of timed-properties are subjects of future work.

Some works in runtime verification have studied the monitorability of different specification languages. Viswanathan and Kim [33] proposed a class of monitorable languages that coincides with the class Π^0_1 of the Arithmetic hierarchy. Diekert et al. [12] discussed monitor constructions for deterministic ω-regular languages. Aceto et al. [3] investigated the monitorability of a variant of the modal μ-calculus that included silent actions (which are different from nil events because they represent targets' computational steps that are not fully visible by the monitor). In contrast to SMAs, these works focus on monitors that can only recognize violations, i.e. they cannot insert corrective actions on traces.

Finally, some authors have used stream-processing automata models to study other problems. Alur and Cerny [5] proposed algorithms for checking functional equivalence and verification of pre/post conditions for a class of finite-state transducers. Alur et al. [7] investigated the expressiveness of cost register automata [6] and identified a subclass that generalizes weighted automata [14]. Apart from being limited to finite state machines, none of these models establish connections with game, safety, or liveness properties, nor discuss generalizations of enforcement definitions.

8 DISCUSSION AND FUTURE WORK

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

The SMA model overcomes several limitations of previous works with respect to expressiveness and enables interesting analyses considering different enforcement scenarios. SMAs model practical abilities of security mechanisms monitoring streams that were not previously considered, e.g. their ability to act even in the absence of input events and enforce some non-safety properties.

SMAs adopt a simple enforcement model in terms of soundness, a widely accepted requirement across the security and formal methods communities. Interesting connections are drawn between runtime enforcement and game theory, and the exact set of enforceable properties is characterized by a new definition of game properties. The model's extensibility is demonstrated with adaptable meta-policies that embody extraneous enforcement constraints.

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

REFERENCES

[1] 2018. Serverless Architecture Market Size, Share. Industry Report, 2018-2025. https://www.grandviewresearch.com/industry-analysis/serverless-architecture-market.

[2] 2018. Serverless. The Serverless Application Framework powered by AWS Lambda, API Gateway, and more. https://serverless.com/.

[3] Luca Aceto, Antonis Achilleos, Adrian Francalanza, and Anna Ingólfsdóttir. 2018. Monitoring for silent actions. In *37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

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

[5] Rajeev Alur and Pavol Černỳ. 2011. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 599–610.

[6] Rajeev Alur, Loris DAntoni, Jyotirmoy Deshmukh, Mukund Raghothaman, and Yifei Yuan. 2013. Regular functions and cost register automata. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE, 13–22.

[7] Rajeev Alur, Dana Fisman, Konstantinos Mamouras, Mukund Raghothaman, and Caleb Stanford. 2018. Streamable regular transductions. *arXiv preprint arXiv:1807.03865* (2018).

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

[9] Daniele Beauquier and J-E Pin. 1991. Languages and scanners. *Theoretical Computer Science* 84, 1 (1991), 3–21.

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

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

[12] Volker Diekert, Anca Muscholl, and Igor Walukiewicz. 2015. A Note on Monitors and Büchi automata. In *International Colloquium on Theoretical Aspects of Computing*. Springer, 39–57.

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

[14] Manfred Droste, Werner Kuich, and Heiko Vogler. 2009. *Handbook of weighted automata*. Springer Science & Business Media.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[33] Mahesh Viswanathan and Moonzoo Kim. 2004. Foundations for the run-time monitoring of reactive systems–fundamentals of the mac language. In *International Colloquium on Theoretical Aspects of Computing*. Springer, 543–556.