Runtime Enforcement of Security Policies on Black Box Reactive Programs

Minh Ngo Fabio Massacci
University of Trento, Italy
ngo@disi.unitn.it,Fabio.Massacci@unitn.it

Dimiter Milushev Frank Piessens
iMinds-DistriNet, KU Leuven, Belgium
{Dimiter.Milushev,Frank.Piessens}@cs.kuleuven.be

Abstract

Security enforcement mechanisms like execution monitors are used to make sure that some untrusted program complies with a policy. Different enforcement mechanisms have different strengths and weaknesses and hence it is important to understand the qualities of various enforcement mechanisms.

This paper studies runtime enforcement mechanisms for reactive programs. We study the impact of two important constraints that many practical enforcement mechanisms satisfy: (1) the enforcement mechanism must handle each input/output event in finite time and on occurrence of the event (as opposed to for instance Ligatti's edit automata that have the power to buffer events for an arbitrary amount of time), and (2) the enforcement mechanism treats the untrusted program as a *black box*: it can monitor and/or edit the input/output events that the program exhibits on execution and it can explore alternative executions of the program by running additional copies of the program and providing these different inputs. It can *not* inspect the source or machine code of the untrusted program.

Such enforcement mechanisms are important in practice: they include for instance many execution monitors, virtual machine monitors, and secure multi-execution or shadow executions.

We establish upper and lower bounds for the class of policies that are enforceable by such black box mechanisms, and we propose a generic enforcement mechanism that works for a wide range of policies. We also show how our generic enforcement mechanism can be instantiated to enforce specific classes of policies, at the same time showing that many existing enforcement mechanisms are optimized instances of our construction.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection

General Terms Security

Keywords Runtime Enforcement; Hypersafety Policy; Black Box Mechanism; Reactive Program

POPL '15, January 15–17, 2015, Mumbai, India.

http://dx.doi.org/10.1145/2676726.2676978

1. Introduction

An *enforcement mechanism* is a mechanism to ensure that some untrusted program will comply with an independently specified security policy for that program. An interesting question that has received a considerable amount of attention over the past decade is: for what classes of policies do there exist *secure* and *precise* enforcement mechanisms? Roughly speaking, an enforcement mechanism is *secure* if it ensures that any program running under the enforcement mechanism complies with the policy. It is *precise* if it does not change the behavior of secure programs in any significant way, i.e. secure programs are not affected by the enforcement mechanism.

For purely static enforcement mechanisms (i.e. mechanisms that should accept or reject the program after some finite amount of analysis, possibly including verification of the code of the program, model checking of the program, or the execution of a finite number of test runs) it is known that they can enforce exactly the recursively decidable properties of programs (i.e. the class Π_0 of the arithmetic hierarchy) [15].

For dynamic (runtime) enforcement mechanisms, the picture is more complicated. At least in part this is the case because it is less clear what exactly runtime mechanisms are allowed to do. Can they only monitor the program and halt it on an observed policy violation? Or are they also allowed to change the program and/or its executions? And if so under what constraints? Obviously, the enforcement mechanism should not be allowed to make arbitrary changes if it is to be precise; it should preserve (in some sense) the behavior of secure programs.

For the important mechanism of *execution monitoring*, the mechanism can only observe executions of the program and terminate them (i.e. prevent any further actions) as soon as they are observed not to comply with the policy. The enforcement power of this mechanism was first studied by Schneider [25] and Schneider's results were later refined by Viswanathan et al. [28] and Hamlen et al. [15]. Roughly speaking, execution monitors can enforce the computable safety properties, but the details depend on the intervention power of the monitor (if there are certain events/actions that the monitor can not prevent from happening, such as the passing of time, this impacts the set of enforceable properties) [4, 15].

Edit automata are a generalization of execution monitors where the monitor can also suppress, insert or change actions that the program performs. They were shown to be strictly more powerful than execution monitors [19]. Edit automata can enforce the so-called *infinite renewal properties*, a class of properties that also includes some liveness properties. Yet, this is only possible at the expense of assuming that the monitor can buffer input/output events for an arbitrary amount of time. For reactive programs, it would be desirable if the enforcement mechanism handles each input/output event on occurrence of the event. Buffering an input/output event

will often inhibit further progress of the reactive program: events happen in response to earlier events.

The main objective of this paper is to come to a better understanding of the enforcement power of dynamic enforcement mechanisms for deterministic reactive programs that satisfy two constraints that are important in practice. First, we require the enforcement mechanism to handle each input event in finite time and before processing the next input event (no buffering). Second, the enforcement mechanism does not analyze the code or runtime state of the program: it treats the program as a *black box*. The enforcement mechanism can intercept input/output events and possibly modify them. It can also run multiple copies of the program as in secure multi-execution [11] or shadow executions [7].

In summary, this paper makes the following contributions:

- we define the class of black box enforceable policies, a class of policies that is enforceable by only considering the I/O behaviour of a program (and hence *without* any analysis of the source or binary code of the program).
- we establish upper and lower bounds for this class.
- we develop a generic black box enforcement mechanism, and prove that it is secure and precise.
- we show how this generic enforcement mechanism can be instantiated to interesting policies.

The remainder of this paper is structured as follows: in Sections 2 and 3 we define reactive programs and security policies. In Section 4 we define and study the notion of black box enforceable policies, and construct a generic black box enforcement mechanism. In Section 5 we show specific useful and interesting instances of the generic construction, and we discuss some of the implications of our result in Section 6. Finally Sections 7 and 8 discuss related work and conclude.

2. A Model of Reactive Programs and Observations

We model reactive programs as states in a labeled transition system with a total computable deterministic transition relation. Programs are black boxes in the sense that the only observation one can do on a program is feeding it input and observing the corresponding output. This section formalizes these concepts.

2.1 Programs

Let $\mathbf{I}$ (resp. $\mathbf{O}$) be enumerable sets of input (resp. output) values. The metavariable i (resp. o) ranges over $\mathbf{I}$ (resp $\mathbf{O}$).

A *stream of inputs* I (resp. *stream of outputs* O) is defined coinductively from $[]$, the empty stream and the observation $i : I$ (resp. $o : O$) of a value i and a tail stream I. The set of (finite and infinite) streams over $\mathbf{I}$ (resp. $\mathbf{O}$) is denoted $\mathbf{I}^\infty$ (resp. $\mathbf{O}^\infty$). The set of *finite* streams over $\mathbf{I}$ (resp. $\mathbf{O}$) is denoted $\mathbf{I}^\star$ (resp. $\mathbf{O}^\star$). We abuse notation and denote by $I : i$ the operation of appending value i to the finite stream I. For simplicity we will abbreviate $[i_1 : [\ldots [i_n : []] \ldots]]$ as $[i_1 : \ldots : i_n]$ for $n \geq 1$.

Definition 2.1. *A* labelled transition system *(LTS) is a tuple* $\langle \mathbf{C}, \mathbf{I}, \mathbf{O}, \rightarrow \rangle$*, where* $\mathbf{C}$ *is the set of states and* $\rightarrow \subseteq \mathbf{C} \times (\mathbf{I} \times \mathbf{O}) \times \mathbf{C}$ *is a transition relation. We denote a transition as* $C \xrightarrow{i|o} C'$*. An LTS is* input-total *if for all* $C \in \mathbf{C}$*, and* $i \in \mathbf{I}$*, there exist* $C' \in \mathbf{C}$ *and* $o \in \mathbf{O}$ *such that* $C \xrightarrow{i|o} C'$*. An LTS is* deterministic *if for all* $C \in \mathbf{C}$ *and for all* $i \in \mathbf{I}$*, if* $C \xrightarrow{i|o} C'$ *and* $C \xrightarrow{i|o'} C''$ *then* $o = o'$ *and* $C' = C''$*.*

For this paper, program states C are elements of the state set $\mathbf{C}$ of some input-total and deterministic LTS $\langle \mathbf{C}, \mathbf{I}, \mathbf{O}, \rightarrow \rangle$.

We assume throughout the paper that the function mapping any (C, i) to (o, C') such that $C \xrightarrow{i|o} C'$ holds is total and computable. The intuition is that if $C \xrightarrow{i|o} C'$ holds then program state C will process input i, produce output o in finite time, and transition to the new program state C'. We will often refer to states C as programs, rather than program states.

Each program C transduces an input stream I into an output stream O. This process coinductively defines a relation $C \xrightarrow{I|O} C'$ by the rules specified below.

$$\text{NIL}\ \frac{}{C \xrightarrow{[]|[]} C} \qquad \text{CONS}\ \frac{C \xrightarrow{i|o} C'' \qquad C'' \xrightarrow{I|O} C'}{C \xrightarrow{i:I|o:O} C'}$$

This relation defines a function on finite input streams that maps any (C, I) to some (O, C'). Since the function that processes an individual input is total and computable, also the function over streams is total and computable for finite input streams I. Therefore, we write $C(I) = O$ if there is some program C' such that $C \xrightarrow{I|O} C'$ and say that program C, on input I, produces output O.

Programs run for ever in this model (every program state is ready to accept more inputs), but termination of a program can be modeled by producing a special o_{end} output value ad infinitum for every input value.

Bohannon et al.'s model of reactive systems [6] allows divergence on a single input value. We impose the constraint that a program can not diverge on a single input value, because we want to impose that same constraint on enforcement mechanisms. The consequences of this choice are significant, and further discussed in Section 6.

Listing 1 shows an example program, specified as a function on an input value and on program state stored in a state variable.

Listing 1. A sum program

```
1 state var sum:Integer = 0
2 input(i:Integer) {
3   sum := sum + i;
4   output(sum)
5 }
```

This program maps for instance input stream $[1 : 2 : 3]$ on output stream $[1 : 3 : 6]$, and the infinite input stream $[1 : 1 : 1 : \ldots]$ on the infinite output stream $[1 : 2 : 3 : \ldots]$.

2.2 Finite observations on programs

A *primitive observation* is a pair (I, O) where $I \in \mathbf{I}^*$ and $O \in \mathbf{O}^*$ are finite and of equal length. A program C has the primitive observation (I, O) iff $C(I) = O$.

An *observation* M is a finite set of primitive observations, i.e. a partial mapping of finite input streams to finite output streams. A program C has the observation M (denoted by $M \subseteq C$) iff it has all the primitive observations in M.

The *prefix* of a primitive observation $(I : i, O : o)$ is (I, O). An observation M is *prefix-closed* if for each $(I : i, O : o) \in M$, it is also the case that $(I, O) \in M$. The *prefix closure* of M (denoted as $\overline{M}$) is the smallest set that includes M and is prefix-closed. It is straightforward to prove that if $M \subseteq C$, then $\overline{M} \subseteq C$.

We write $\text{inputs}(M)$ for the set $\{I \mid (I, O) \in M\}$. Given a finite set Is of finite input streams, we define

$$\text{map}(C, Is) \quad = \quad \{(I, O) \mid I \in Is \wedge C(I) = O\}.$$

It is easy to see that $M \subseteq C$ iff $\text{map}(C, \text{inputs}(M)) = M$.

We say an observation M is *possible* if there exists a program C that has the observation M. For instance, $\{([1], [1]), ([1 : 2], [1 :$

$3])\}$ is possible because it is an observation on the sum program above, but the observation $\{([1],[1]),([1:2],[2:3])\}$ is not possible: any program that has primitive observation $([1:2],[2:3])$ must also have the primitive observation $([1],[2])$ and hence by determinism of programs it can not have observation $([1],[1])$.

Proposition 2.1. *An observation is possible if and only if its prefix-closure is deterministic i.e.* $(I,O) \in \overline{M}$ *and* $(I,O') \in \overline{M}$ *implies that* $O = O'$.

In the sequel, we only consider possible observations. We write $\mathbf{OBS}$ for the set of all possible observations, and use the metavariable $M \in \mathbf{OBS}$ to range over them.

Definition 2.2. *Two programs* C *and* C' *are* observationally equivalent *(denoted by* $C \sim C'$*) iff they have the same primitive observations.*

It follows that, if two programs C and C' are *not* observationally equivalent, there is some finite input I such that $C(I) \neq C'(I)$.

3. Policies

A policy $\mathcal{P}$ can be defined very generally as the set of programs allowed by the policy. Membership of the policy is required to be compatible with observational equivalence: if $C \in \mathcal{P}$ then all programs observationally equivalent with C must also be in $\mathcal{P}$. Hence, one can also think of a policy as a set of sets of primitive observations: a program satisfies the policy iff the set of primitive observations of the program is an element of the policy [10].

For black box dynamic enforcement mechanisms such as the ones in this paper, we are interested in the subset of policies for which violation of the policy can be detected by means of a finite observation, and for which the test of violation is decidable.

3.1 Hypersafety policies

The policies for which violation can be detected by a finite observation were formalized by Clarkson and Schneider [10] as the hypersafety policies: if C is not in the policy then C has a finite observation M_{bad} that only programs disallowed by the policy have.

Definition 3.1. *A policy* $\mathcal{P}$ *is a* hypersafety policy *iff*

$$\forall C \notin \mathcal{P} \quad \Rightarrow \quad (\exists M_{bad} \in \mathbf{OBS}.\ M_{bad} \subseteq C \\ \wedge \big(\forall C'.\ M_{bad} \subseteq C' \Rightarrow C' \notin \mathcal{P}\big))$$

If there is a bound k on the cardinality of these bad observations, the policy is a k-safety policy.

Definition 3.2. *A policy* $\mathcal{P}$ *is a* k-safety policy *iff*

$$\forall C \notin \mathcal{P} \quad \Rightarrow \quad (\exists M_{bad} \in \mathbf{OBS}.\ M_{bad} \subseteq C \wedge |M_{bad}| \leq k \\ \wedge \big(\forall C'.\ M_{bad} \subseteq C' \Rightarrow C' \notin \mathcal{P}\big))$$

Hypersafety policies can be specified by defining a set $\mathcal{M}$ of bad or disallowed observations. The corresponding policy $\mathcal{P}$ is then defined as: $C \notin \mathcal{P}$ if and only if C has one of the specified bad observations $M_{bad} \in \mathcal{M}$. However, for a given hypersafety policy, $\mathcal{M}$ need not be unique. For example, one may choose $\mathcal{M}$ to be the set containing any one M_{bad} for every $C \notin \mathcal{P}$.

Example 1. *Suppose there are two distinguished output values* o_{send} *(the program is sending something over the network) and* o_{read} *(the program initiates a read action on the file system). The 1-safety policy* no-send-after-read *(NSAR) can be specified by the following set of bad observations:*

$$\{\{(I,O)\} | sendAfterRead(O)\}$$

where $sendAfterRead(O)$ *is a boolean function defined as:*

$$\begin{aligned} sendAfterRead([]) &= false \\ sendAfterRead(o_{read} : O) &= occurs(o_{send}, O) \\ sendAfterRead(\_ : O) &= sendAfterRead(O) \end{aligned}$$

where the boolean function $occurs(o, O)$ *is true if the value* o *occurs somewhere in finite stream* O.

NSAR is 1-safety since every specified bad observation is a singleton containing 1 primitive observation.

Example 2. *Noninterference (NI), the policy that low (L) outputs do not depend on high (H) inputs is a 2-safety policy, and can be specified by defining a set of disallowed observations as follows. Let* lvl *be a function that assigns H or L to input and output values. Given a finite stream I, let* $I|_L$ *be the resulting stream after filtering out the input values* i *with* $lvl(i) = H$ *(and similarly for* $O|_L$*).*

Then a program C *is* noninterferent *(or* $C \in \mathcal{P}_{NI}$*) iff*

$$\forall I, I' \in \mathbf{I}^* : I|_L = I'|_L \implies O|_L = O'|_L,$$

where $C(I) = O$ *and* $C(I') = O'$.

A set of bad observations that specifies $\mathcal{P}_{NI}$ *is:*

$$\{\{(I,O),(I',O')\} \mid I|_L = I'|_L \wedge\ O|_L \neq O'|_L\}.$$

A program C *that has an observation in this set is not NI. If* C *does not have any such observation, then it is NI.*

NI is a 2*-safety hyperproperty since the bad observations in the set above all have cardinality 2.*

Here is an example of a k-safety policy for arbitrary k [10].

Example 3. *Let* o_i *be the act of outputting the* i*-th share of a secret (for* $0 < i \leq k$*). The policy that no observation on the program will ever reveal all* k *shares can be specified by defining the following set of bad observations:*

$$\{\{(I_1,O_1),\ldots,(I_k,O_k)\} \mid occurs(o_1,O_1) \wedge \ldots \wedge occurs(o_k,O_k)\}$$

3.2 Testable hypersafety policies

To enforce a policy, it should be decidable whether an observation made on the program is allowed or not by the policy. For policies specified by a set of bad observations $\mathcal{M}$, this seems to imply we want membership of $\mathcal{M}$ to be decidable.

However, a hypersafety policy can be specified by different sets of bad observations, and membership can be decidable for some of these sets and not for others.

A canonical set of bad observations for a given policy is the *maximal* set.

Definition 3.3. *For a hypersafety policy* $\mathcal{P}$*, we define* $\mathcal{M}_{\mathcal{P}}$*, the* maximal *set of bad observations as:*

$$\mathcal{M}_{\mathcal{P}} = \{M_{bad} \mid \forall C : M_{bad} \subseteq C \implies C \notin \mathcal{P}\}$$

An observation M *is* allowed *by a policy* $\mathcal{P}$ *iff* $M \notin \mathcal{M}_{\mathcal{P}}$.

It is *maximal* in the sense that any other set $\mathcal{M}$ that specifies the same policy $\mathcal{P}$ is a subset of $\mathcal{M}_{\mathcal{P}}$.

For any hypersafety policy $\mathcal{P}$, the maximal set $\mathcal{M}_{\mathcal{P}}$ always exists. *Yet*, it is possible that $\mathcal{P}$ can be specified by a computable set of disallowed observations, even if $\mathcal{M}_{\mathcal{P}}$ is not computable.

Example 4. *Let* $\mathbf{O}$ *be the set of closed terms* t *of the untyped* λ*-calculus. Consider the policy that specifies that programs can only output valid reduction sequences (in some deterministic reduction relation): for any input, the next term on the output is a valid reduction of the previous term on the output stream. A program can therefore only emit a valid reduction of the term until it reaches normal form, and then no valid output is possible anymore. Recall from Section 2 that programs have to keep producing output (possibly stuttering on a specific output value to model termination).*

Hence, the only programs satisfying the policy are the programs that output sequences starting with a λ-term whose reduction does not terminate, and ad infinitum output further reductions of this term.

We define two predicates $isValidRS(O)$ which holds iff O is a valid reduction sequence *and $isValidNTRS(O)$ which holds iff O is a valid reduction sequence that ends in a term that is not normalizing (and hence can be further reduced indefinitely long).*

$$\begin{aligned} isValidRS([]) &= \textbf{\textit{true}} \\ isValidRS([t]) &= \textbf{\textit{true}} \\ isValidRS(t_1 : t_2 : O) &= t_1 \text{ λ-reduces to } t_2 \quad \wedge \\ &\quad\; isValidRS(t_2 : O) \\ isValidNTRS(O) &= isValidRS(O) \wedge \\ &\quad\; last(O) \text{ not normalizing} \end{aligned}$$

We can use the following set of bad observations:

$$\{M \mid \exists (I,O) \in \overline{M}, \neg isValidRS(O)\}$$

Membership in this set is computable and correctly captures the policy: any bad program will eventually have one of these bad observations. Yet, it is not maximal for the hypersafety policy it specifies: we can have observations (e.g. a program emitting a normalizing term), that are themselves not in the specified set of bad observations, but any program that has the observation is bound to eventually violate the policy. Once a program has output a normalizing term, it will eventually have to stop outputting valid reductions: it can reduce until it reaches normal form, but then the next element in the output cannot be obtained by a reduction of the former element, and the policy is violated.

The maximal set corresponding to this policy is:

$$\{M \mid \exists (I,O) \in \overline{M}, \neg isValidNTRS(O)\}$$

Any observation that is bound to eventually fail is in *this set: as soon as a program outputs a term that will always terminate (i.e. is normalizing), the program is rejected. However, membership in this maximal set is not computable (as this would require deciding termination of λ-terms).*

Definition 3.4. *A hypersafety policy $\mathcal{P}$ is* testable *iff membership in $\mathcal{M}_\mathcal{P}$ is decidable.*

For the construction of enforcement mechanisms, we limit our attention to testable hypersafety policies. Such policies can be specified by giving a total computable boolean function reject(M) that for an observation M returns **true** iff $M \in \mathcal{M}_\mathcal{P}$.

It is non-trivial to check whether a set of bad observations specified by a reject function is actually maximal. For example, the set of bad observations that we specified in Example 2 for NI is *not* maximal. It does not contain observations that have violated the policy in the past but where things have "re-adjusted" as the execution progressed, as shown in the following example:

Example 5. *Suppose that $\mathbf{I} = \mathbf{O} = \{0,1\}$ and that $lvl(0) = L$ whereas $lvl(1) = H$. Consider the following observation $\{([0:1:0],[0:1:0]),([1:0:0],[0:0:1])\}$. This observation is possible and it satisfies the simple test presented in Example 2 because the outputs are equivalent $[0:1:0]|_L = [0:0] = [1:0:0]|_L$. However, no program that satisfies the NI policy can generate this observation because it will have to first generate the observation $\{([0:1],[0:1]),([1:0],[0:0])\}$ which would violate the policy.*

Fortunately, the maximal set for NI is still decidable.

Example 6 (NI, a testable 2-safety hyperproperty)**.** *For the NI policy discussed in Example 2, a reject predicate can be constructed as follows:*

$$reject(M) = \begin{cases} \textbf{\textit{true}} & \exists M_{bad} = \{(I,O),(I',O')\}, \\ & M_{bad} \subseteq \overline{M} \text{ s.t. } I|_L = I'|_L \text{ and } O|_L \neq O'|_L, \\ \textbf{\textit{false}} & \text{otherwise.} \end{cases}$$

It is straightforward to check that this is a total computable function. We show that it specifies the maximal set of bad observations by contradiction.

*Suppose that there is an observation M such that reject$(M) =$ **false** and $M \in \mathcal{M}_\mathcal{P}$. By construction of the reject function we have that reject$(\overline{M}) =$ **false**. Since $\mathcal{M}_\mathcal{P}$ is maximal, $\overline{M} \in \mathcal{M}_\mathcal{P}$.*

By definition of $\mathcal{M}_\mathcal{P}$, all programs C such that $\overline{M} \subseteq C$ must not belong to the policy. Consider now one of such programs C_{good} such that for all $(I,O) \in \overline{M}$, $C_{good}(I) = O$ and otherwise it always outputs a value o with $lvl(o) = H$. This program satisfies the policy. Contradiction.

Example 7. *The policy in Example 4 is an example of a hypersafety policy that is not testable.*

3.3 Incrementally constructing observations allowed by a policy

An enforcement mechanism must not only decide whether or not an observation is rejected by the policy. It must also "correct" programs that turn out to have observations that are not allowed (for instance by terminating the program, or more generally by modifying the outputs of the program).

Given an observation M of the untrusted program that is still allowed so far, when we find for the next input i that the corresponding output will lead to a violation of the policy, we need to find another output that will *not* lead to a violation of the policy.

Definition 3.5. *Given a set of bad observations $\mathcal{M}$ that specifies a hypersafety policy, a function extend$_\mathcal{M}(M,I,i)$ is an* extension function *for $\mathcal{M}$ iff, for any observation $M \notin \mathcal{M}$, where $(I,O) \in M$, it returns an $o \in \mathbf{O}$ such that $M \cup \{(I:i, O:o)\} \notin \mathcal{M}$.*

Any extension function can be used for "correcting" outputs. Think of I as the input processed so far, i as the new input to be processed, and M as the set of observations made on the program so far (possibly including primitive observations on alternative input streams). If the new output o observed of the program makes $M \cup \{(I:i, O:o)\}$ a bad observation, then extend$_\mathcal{M}(M,I,i)$ can be used to replace this output.

One of the reasons why it is useful to work with the maximal set of bad observations to specify a policy is that for the maximal set, an extension function always exists.

Proposition 3.1. *For any hypersafety policy $\mathcal{P}$, there exists an extension function for the maximal set of bad observations $\mathcal{M}_\mathcal{P}$.*

Proof. If the policy $\mathcal{P}$ is empty then all observations are bad observations (in $\mathcal{M}_\mathcal{P}$) and therefore the precondition for the applicability of the extension function is false, and we are done.

If the policy is not empty, consider an observation M that is allowed by $\mathcal{M}_\mathcal{P}$. By definition of maximal set of bad observations there must be a program C_{good} such that $M \subseteq C_{good}$ and $C_{good} \in \mathcal{P}$ (if none existed M would have been in $\mathcal{M}_\mathcal{P}$).

Let I be an arbitrary input stream such that $(I,O) \in M$ and i an arbitrary input value. By definition of observation on a program, it must be $C_{good}(I) = O$. Since programs are input total, $C_{good}(I:i)$ has the form $O:o$ for some $o \in \mathbf{O}$. Pick this o as the return value for extend$_{\mathcal{M}_\mathcal{P}}(M,I,i)$.

Suppose now $M \cup \{(I:i, O:o)\} \in \mathcal{M}_\mathcal{P}$. Since $M \cup \{(I:i, O:o)\} \subseteq C_{good}$ by definition of maximal set of bad observation it should be $C_{good} \notin \mathcal{P}$. Contradiction. Therefore the set $M \cup \{(I:i, O:o)\}$ is also allowed by $\mathcal{M}_\mathcal{P}$. ∎

For a given hypersafety policy $\mathcal{P}$, we write $\text{extend}_{\mathcal{P}}(M, I, i)$ as an abbreviation for some function $\text{extend}_{\mathcal{M}_{\mathcal{P}}}(M, I, i)$ that is guaranteed to exist by the proposition above.

Interestingly, for testable hypersafety policies, there is always a *total computable* extension function.

Proposition 3.2. *Let $\mathcal{P}$ be a testable hypersafety policy, then there exists a total computable extension function for $\mathcal{M}_{\mathcal{P}}$.*

Proof. Let M be an observation, I be an arbitrary input stream in inputs(M) and i be an arbitrary input value. The total computable extension function is constructed as follows:

1. if the first argument M already belongs to $\mathcal{M}_{\mathcal{P}}$ then the function returns an arbitrary output value; (in this case the precondition for the extension function is not satisfied, and we can return any value)
2. otherwise the function enumerates all output values o and submits each observation $M \cup \{(I : i, O : o)\}$ to the total computable membership test for $\mathcal{M}_{\mathcal{P}}$ continuing until the reject function returns **false**.

Since (by Proposition 3.1) an o_{good} exists such that $M \cup \{(I : i, O : o_{good})\}$ is not in $\mathcal{M}_{\mathcal{P}}$ this procedure terminates. ■

The function constructed in the proof of this proposition is not very efficient. Many policies admit much more efficient ways of extending allowed observations.

Example 8. *For the NSAR policy, $\text{extend}_{\mathcal{P}_{NSAR}}(M, I, i)$ can be defined to always return o_{read}: Appending o_{read} to an allowed output stream is always allowed.*

Example 9. *For the NI policy, we can define $\text{extend}_{\mathcal{P}_{NI}}(M, I, i)$ as follows. Let o_H be an arbitrary output value with $lvl(o_H) = H$.*

1. *if $(I : i, O : o)$ is in $\overline{M}$, then return o,*
2. *else if $lvl(i) = H$, then return o_H,*
3. *else if $lvl(i) = L$:*
   (a) *if there exists $(I' : i, O' : o')$ in $\overline{M}$ s.t. $I'|_L = I|_L$ then return o'*
   (b) *otherwise, return o_H.*

We show that this is a correct extension function by contradiction.

*Suppose there exists an input stream I, an input value i, and an output stream O such that $(I, O) \in M$, reject(M) = **false**, and reject$(M \cup \{(I : i, O : o)\})$ = **true**, where the reject predicate is specified as in Example 6 and $o = \text{extend}_{\mathcal{P}_{NI}}(M, I, i)$ is the result of the above algorithm.*

*By construction of the reject function we also have reject$(\overline{M'})$ = reject(M') for all M'. Let $M_1 = M \cup \{(I : i, O : o)\}$, then by hypothesis and the properties of the reject predicate we have that reject$(\overline{M_1})$ = **true**. Further, since $(I, O) \in M$ we have that $\overline{M_1} = \overline{M} \cup \{(I : i, O : o)\}$.*

- *If $(I : i, O : o)$ is in $\overline{M}$, then $\overline{M_1} = \overline{M}$. Thus, reject$(\overline{M})$ = **true**. Contradiction.*
- *If $lvl(i) = H$, then $o = o_H$, hence $lvl(o) = H$. Since reject(M_1) = **true**, there exists (I_b, O_b) in $\overline{M}$ s.t $I_b|_L = I : i|_L$ and $O_b|_L \neq O : o|_L$. Because $lvl(i) = lvl(o) = H$, it follows that $I : i|_L = I|_L$ and $O : o|_L = O|_L$. But then $I_b|_L = I|_L$ and $O_b|_L \neq O|_L$. Thus, reject$(\overline{M})$ = **true**. Contradiction.*
- *If $lvl(i) = L$ and there exists $(I' : i, O' : o')$ in $\overline{M}$ s.t. $I'|_L = I|_L$, then o is o'. Since reject$(\overline{M_1})$ = **true**, there exists (I_b, O_b) in $\overline{M}$ s.t $I_b|_L = I : i|_L$, and $O_b|_L \neq O : o|_L$. But $I' : i|_L = I : i|_L$ and $O' : o'|_L = O : o|_L$, and since both (I_b, O_b) and $(I' : i, O' : o')$ are in $\overline{M}$ it follows that reject$(\overline{M})$ = **true**. Contradiction.*
- *If $lvl(i) = L$ and there is no $(I' : i, O' : o')$ in $\overline{M}$ s.t. $I'|_L = I|_L$, then $o = o_H$. Since reject$(\overline{M_1})$ = **true**, there must exist I_b in inputs$(\overline{M})$ s.t $I_b|_L = I : i|_L$. Let I'_b be the prefix of I_b that removes all elements with level H at the end of I_b. Then I'_b is in inputs$(\overline{M})$, and it must have the form $I''_b : i$ and it must have $I''_b|_L = I_L$. Contradiction.*

4. Black Box Enforcement Mechanisms

4.1 Definition

We model enforcement mechanisms as total computable functions from programs to programs: they turn a program (that possibly does not satisfy the policy) into a program that definitely complies with the policy (the *security* property). Moreover, they do not have any observable impact on untrusted programs that *do* happen to satisfy the policy (the *precision* property). Finally, they should be *black box*, i.e. only depend on the externally observable behaviour of their input program.

Definition 4.1. *A* black box enforcement mechanism *for a policy $\mathcal{P}$ is a total computable function $\mathsf{E}_{\mathcal{P}}$ from programs to programs, which satisfies the following properties:*

- *Security: for all programs C, $\mathsf{E}_{\mathcal{P}}(C) \in \mathcal{P}$*
- *Precision: if $C \in \mathcal{P}$, then $C \sim \mathsf{E}_{\mathcal{P}}(C)$*
- *Black box: if $C_1 \sim C_2$, then $\mathsf{E}_{\mathcal{P}}(C_1) \sim \mathsf{E}_{\mathcal{P}}(C_2)$*

A policy $\mathcal{P}$ is black box enforceable *iff there exists a secure and precise black box enforcement mechanism for $\mathcal{P}$.*

Because enforcement mechanisms produce programs in our program model, they have to respond to any input event in finite time with an appropriate output event without buffering inputs or outputs.

In this section, we derive upper and lower bounds on the set of black box enforceable policies.

4.2 Upper bounds on black box enforceable policies

First, non-suprisingly, any policy that is black box enforceable is a hypersafety policy.

Theorem 4.1. *If an enforcement mechanism $\mathsf{E}_{\mathcal{P}}$ exists for a policy $\mathcal{P}$, then $\mathcal{P}$ is a hypersafety policy.*

Proof. Suppose $C \notin \mathcal{P}$. We have to construct an M_{bad} such that (1) $M_{bad} \subseteq C$, and (2) $\forall C'.\ M_{bad} \subseteq C' \Rightarrow C' \notin \mathcal{P}$.

If $C \notin \mathcal{P}$, then, by security of the enforcement mechanism $\mathsf{E}_{\mathcal{P}}(C)$ is not observationally equivalent to C. Hence, there must be some finite input stream I such that $C(I) \neq \mathsf{E}_{\mathcal{P}}(C)(I)$. Take as M_{bad} the set of all primitive observations that $\mathsf{E}_{\mathcal{P}}$ has done on C during the execution of $\mathsf{E}_{\mathcal{P}}(C)(I)$, and add the primitive observation $(I, C(I))$. This set is necessarily finite as $\mathsf{E}_{\mathcal{P}}(C)(I)$ must process the finite input I in finite time.

It is clear that (1) holds, as M_{bad} is constructed only using primitive observations on C.

In order to prove (2), suppose a program C' has this same observation. Then $C'(I) = C(I)$, because M_{bad} includes $(I, C(I))$. But also $\mathsf{E}_{\mathcal{P}}(C')(I) = \mathsf{E}_{\mathcal{P}}(C)(I)$, as C' will respond exactly as C, to *all* the observations that $\mathsf{E}_{\mathcal{P}}$ does while processing the input stream I and $\mathsf{E}_{\mathcal{P}}$ is a deterministic program. Hence $C'(I) \neq \mathsf{E}_{\mathcal{P}}(C')(I)$. By hypothesis the enforcement mechanism is precise and therefore $C' \notin \mathcal{P}$. ■

Obviously, the converse is not true: there are many hypersafety policies that are not enforceable. For instance, the non-testable

policy in Example 4 is not enforceable: if the program outputs a first λ-term, the enforcement mechanism should output this term unmodified iff it is non-terminating. Obviously, it cannot decide this in finite time.

This upper bound can be substantially tightened. We say a hypersafety policy is *semi-testable* if membership in $\mathcal{M}_\mathcal{P}$ is co-recursively enumerable, i.e. we can enumerate all the observations that are allowed by the policy.

Theorem 4.2. *Let $\mathcal{P}$ be a hypersafety policy. If an enforcement mechanism $\mathsf{E}_\mathcal{P}$ exists for $\mathcal{P}$, then $\mathcal{P}$ is semi-testable.*

Proof. We have to provide an algorithm for the reject function that will return **false** for any observation M that is not in $\mathcal{M}_\mathcal{P}$ and will return **true** or diverge for any observation M that is in $\mathcal{M}_\mathcal{P}$.

Given an observation M, the algorithm enumerates all possible programs in some complete programming language for writing reactive programs. For each program, it applies the enforcement mechanism to inputs(M) and then tries to observe M. It returns **false** if it can observe M and otherwise continues with the next program.

If $M \in \mathcal{M}_\mathcal{P}$, then – by the security property of the enforcement mechanism – M will never be observed, and the algorithm diverges.

If $M \notin \mathcal{M}_\mathcal{P}$, then – by the maximality of $\mathcal{M}_\mathcal{P}$ there exists some program C_{good} that has M and satisfies $\mathcal{P}$. By precision, the enforcement mechanism applied to C_{good} will be observationally equivalent to C_{good} and hence will have observation M. Hence, if $M \notin \mathcal{M}_\mathcal{P}$, we will eventually enumerate C_{good}, and the algorithm for the reject function will terminate with return value **false**. ■

4.3 A lower bound on black box enforceable policies

For 1-safety policies it is obvious that testability (as defined in Definition 3.4) is a *sufficient* condition for enforceability. If membership of an observation in the maximal set of bad observations is decidable, then it is obviously possible to decide whether primitive observation $\{(I, O)\}$ is a bad observation. Therefore, one can simply use the reject function as a security automaton in the sense of Schneider [25]. At the point where the primitive observation corresponding to the current execution is about to violate the policy, one can use the extension function to correct the execution. Schneider considers only termination as a corrective measure (in the terminology of this paper, he assumes that outputting o_{end} is never disallowed by any policy, and hence one can use the constant function on o_{end} as an extension function). In cases where policies can talk about termination too [15], one can use another extension function. Only in the case where the policy is empty (and hence the maximal set of bad observations contains all observations), a testable 1-safety policy is not black box enforceable.

Surprisingly, testability is also a sufficient condition for black box enforceability for general hypersafety policies. We show that in this section by constructing a secure and precise enforcement mechanism for any testable hypersafety policy. We need to address two challenges: (1) testing a sufficient number of alternative executions, and (2) doing corrections in a consistent way.

4.3.1 Generating sufficient test inputs

For k-safety policies with $k > 1$, it is not obvious that testability is a sufficient condition for enforceability. For such policies, the enforcement mechanism should not only look at the input/output streams (I, O) of the current execution. For k-safety it should also make sure that other primitive observations that the policy defines to be incompatible with the current execution do not exist, and in general there will be infinitely many alternate input streams that can possibly lead to incompatible observations.

Example 10. *For the NI policy, for a given primitive observation (I, O), the set of all other primitive observations that are incompatible with (I, O) is:*

$$\{(I', O') \quad | \quad I|_L = I'|_L \wedge O'|_L \neq O|_L\}$$

This set contains an infinite number of input streams I', so the enforcement mechanism can not query the program C with all of them in finite time.

Fortunately, it is not necessary to check the behavior of the program on *all* input streams that might be potentially conflicting with the current input stream.

We introduce the notion of *test generator*, a function that computes a finite and sufficient set of alternative input streams to check.

Definition 4.2. *A* test generator *for a hypersafety policy $\mathcal{P}$ is a function $g : \mathbf{I}^\star \to$ finite $2^{\mathbf{I}^\star}$ s.t. for all $C \in \mathbf{C}$ and all $M_{bad} \in \mathcal{M}_\mathcal{P}$:*

$$(\forall I \in \mathit{inputs}(M_{bad}) : \mathit{map}(C, g(I) \cup \{I\}) \notin \mathcal{M}_\mathcal{P}) \Rightarrow M_{bad} \not\subseteq C$$

In other words, if a program C has a bad observation M_{bad}, then there is at least one input stream $I \in \text{inputs}(M_{bad})$ for which $g(I)$ is a sufficiently large set of input streams such that testing the program C on these input streams *in addition to* the actual input stream I will detect a policy violation.

Lemma 4.1. *If, for every I, $\mathit{map}(C, g(I) \cup \{I\}) \notin \mathcal{M}_\mathcal{P}$, then $C \in \mathcal{P}$.*

Proof. Suppose $C \notin \mathcal{P}$. By definition, there is a bad observation $M_{bad} \in \mathcal{M}_\mathcal{P}$ such that $M_{bad} \subseteq C$.

From the property of generators in Definition 4.2, it follows that there exists an $I \in \text{inputs}(M_{bad})$ such that $\text{map}(C, g(I) \cup I) \in \mathcal{M}_\mathcal{P}$. But this contradicts the condition of the lemma. ■

Example 11. *For 1-safety policies, $g(I) = \{\}$ is a test generator. Since a 1-safety policy defines a predicate on primitive observations, any M_{bad} contains at least one (I, O) that violates the predicate. Hence if $M_{bad} \subseteq C$, then at least one of the $\{(I, C(I))\}$ will be in $\mathcal{M}_\mathcal{P}$.*

Example 12. *For the NI policy from Example 2 (with reject specified in Example 6), $g(I) = \{I|_L\}$ is a test generator.*

Let $M_{bad} \in \mathcal{M}_{\mathcal{P}_{NI}}$. This means there must exist $(I, O), (I', O') \in M_{bad}$ with $I|_L = I'|_L$ and $O|_L \neq O'|_L$.

Now, suppose $M_{bad} \subseteq C$, i.e. $C(I) = O$ and $C(I') = O'$. We show that C has a bad observation either on inputs I and $I|_L$, or on inputs I' and $I'|_L$.

Consider the output O'' of C on $I|_L = I'|_L$. Since $O|_L \neq O'|_L$, we must have either that $O'' \neq O|_L$ or $O'' \neq O'|_L$.

- *If $O'' \neq O|_L$, then C has a bad observation on inputs I and $I|_L$.*
- *If $O'' \neq O'|_L$, then C has a bad observation on inputs I' and $I|_L$.*

This implies that g is a generator.

We can further reduce the size of the generator. Let us define g' as follows:

$$g'(I) \quad = \quad \textit{if } I|_L = I \textit{ then } \{\} \textit{ else } \{I|_L\}$$

Since for all I, $g(I) \cup I = g'(I) \cup I'$, it follows easily from Definition 4.2 that g' is a generator if g is a generator.

4.3.2 Consistently correcting executions

A second challenge that needs to be addressed is *how to correct executions*. While processing an input stream I, the enforcement mechanism will explore other input streams in order to check that

there are no bad observations. So the output for input stream I will be computed in different circumstances: while $\mathsf{E}_{\mathcal{P}}(C)$ is actually processing I, as well as while the mechanism's actual input is another stream I' and the stream I is only considered as a potential candidate jointly with I' for membership in a bad set. The enforcement mechanism should compute the same output for I in any of these circumstances.

Example 13. *Assume that* $\mathbf{I} = \mathbf{O} = \{0, 1\}$. $I \oplus I'$ *is defined for* I *and* I' *of equal length as a pair-wise xor of* I *and* I'*, where* $(0 \oplus 1) = (1 \oplus 0) = 1$ *and* $(0 \oplus 0) = (1 \oplus 1) = 0$*. Similarly, we define* $O \oplus O'$*. Let* ***1*** *(resp.* ***0****) denote a stream consisting of only* 1 *values (resp.* 0 *values).*

A program C *satisfies a policy* $\mathcal{P}_{xor}$ *iff*

$$\forall I, I' : I \oplus I' = \mathbf{1} \Rightarrow O \oplus O' = \mathbf{1}$$

The reject function that decides $\mathcal{M}_{\mathcal{P}_{xor}}$ *is as follows:*

$$reject(M) = \begin{cases} \textbf{\textit{true}} & \exists M_{bad} = \{(I, O), (I', O')\}, \\ & M_{bad} \subseteq \overline{M} \text{ s.t. } I \oplus I' = \mathbf{1} \text{ and } O \oplus O' \neq \mathbf{1}, \\ \textbf{\textit{false}} & \text{otherwise.} \end{cases}$$

We define $g(I) \triangleq \{I' \mid I \oplus I' = \mathbf{1}\}$*. Obviously, given an* I*, such* I' *is unique. It is easy to show that it actually is a test generator.*

Now consider a naive construction of an enforcement mechanism $\mathsf{E}_{\mathcal{P}_{xor}}$ *that on execution of a program* C *on input stream* I*, checks the behavior of* C *also on* $g(I) = \{I'\}$*. If the enforcement mechanism finds that* $C(I) \oplus C(I') \neq \mathbf{1}$*, it corrects the output for* $C(I)$ *to make it compliant with the policy. For instance, let* C *be the program that just outputs* 0 *for any input value (i.e.* $C(I) = \mathbf{0}$ *for any* I*). If* $\mathsf{E}_{\mathcal{P}_{xor}}(C)$ *is executed on* $[0]$*, the enforcement mechanism would see that* $C([0]) = [0]$ *and that* $C([1]) = [0]$*, and it would decide to correct the output for* $[0]$ *to* $[1]$*. It is easy to see that* $\mathsf{E}_{\mathcal{P}_{xor}}$ *is not a secure enforcement mechanism, because if* $\mathsf{E}_{\mathcal{P}_{xor}}(C)$ *is executed on* $[1]$*, it would see that* $C([1]) = [0]$ *and that* $C([0]) = [0]$*, and it would decide to correct the output for* $[1]$ *to* $[1]$*. Essentially* $\mathsf{E}_{\mathcal{P}_{xor}}(C)$ *will be a program that always outputs* 1 *on every input, and it violates the policy* $\mathcal{P}_{xor}$ *as badly as* C *does.*

This is an example of inconsistent corrections: on execution of $\mathsf{E}_{\mathcal{P}_{xor}}(C)$ *on* $[0]$*, we are considering the alternate execution* $[1]$*, but we are not taking into account that the alternate execution might as well be corrected if it were ever executed.*

Guaranteeing consistency of corrections is challenging. One idea is to use recursive invocations of the enforcement mechanism while checking alternative inputs.

Example 14. *For the example above, if* $\mathsf{E}_{\mathcal{P}_{xor}}(C)$ *is executed on* $[0]$*, the enforcement mechanism would see that* $C([0]) = [0]$ *and then it should not check this against* $C([1])$*, but against* $\mathsf{E}_{\mathcal{P}_{xor}}(C)([1])$*. Unfortunately, for the given generator, this would lead to divergence, as* $\mathsf{E}_{\mathcal{P}_{xor}}(C)([1])$ *will again recursively call* $\mathsf{E}_{\mathcal{P}_{xor}}(C)([0])$*.*

We address the issue of divergence by means of the notion of well-founded test generator: a generator is *well-founded*, if there exists a well-founded partial order $\sqsubset$ on the set of finite input streams, such that:

- $I \sqsubset I : i$
- $\forall I' \in g(I), I' \sqsubset I$

Now, we can recursively call the enforcement mechanism on alternative input streams generated by the generator, and this will make sure that corrections are done consistently.

Example 15. *Consider again the* $\mathcal{P}_{xor}$ *policy. We now propose the following generator:* $g(I : 0) \triangleq \emptyset$ *and* $g(I : 1) \triangleq \{I' : 0 | I : 1 \oplus I' : 0 = \mathbf{1}\}$.

This is a well-founded generator. The partial order $\sqsubset$ *can be defined as (the transitive closure of)* $I \sqsubset I : i$ *and* $I : 0 \sqsubset I : 1$*.*

Now consider again an enforcement mechanism $\mathsf{E}_{\mathcal{P}_{xor}}$ *that on execution of an untrusted program* C *on input stream* I*, checks the behaviour of* $\mathsf{E}_{\mathcal{P}_{xor}}(C)$ *also on* $g(I)$*. Now the enforcement mechanism will let any program* C *do its original output on* 0*s and it will correct the output on* 1*s so that the policy is satisfied.*

For instance, let C *again be the program that just outputs* 0 *for any input value. If* $\mathsf{E}_{\mathcal{P}_{xor}}(C)$ *is executed on* $[0]$*, the enforcement mechanism would output* $[0]$*. If* $\mathsf{E}_{\mathcal{P}_{xor}}(C)$ *is executed on* $[1]$*, our algorithm would see that* $C([1]) = [0]$ *and that* $\mathsf{E}_{\mathcal{P}_{xor}}(C)([0]) = [0]$*, and it would decide to correct the output for* $[1]$ *to* $[1]$*. Essentially* $\mathsf{E}_{\mathcal{P}_{xor}}(C)$ *will now be a program that echoes inputs on outputs, and hence it is a secure program. The recursive calls to* $\mathsf{E}_{\mathcal{P}_{xor}}(C)$ *always terminate thanks to the well-founded generator.*

Lemma 4.2. *Every non-empty testable hypersafety policy has a well-founded generator.*

Proof. Construct an enumeration of $\mathbf{I}^*$ (the set of finite streams of input values) that has the property that I is enumerated before $I : i$ for all i, I. The constructed enumeration defines a total order on $\mathbf{I}^*$. Say $I \sqsubset I'$ if I is enumerated before I'. Define the generator $g(I) = \{I' \mid I' \sqsubset I\}$. It is easy to check that this is a generator: for any M_{bad}, let I be the maximal element in the set $inputs(M_{bad})$. Then $inputs(M_{bad}) \subseteq g(I) \cup \{I\}$. Since C is deterministic and $inputs(M_{bad}) \subseteq g(I) \cup \{I\}$, if $M_{bad} \subseteq C$ then $M_{bad} \subseteq \text{map}(C, g(I) \cup \{I\})$, and hence $\text{map}(C, g(I) \cup \{I\}) \in \mathcal{M}_{\mathcal{P}}$ (from the property of maximal set of bad observations). ■

4.3.3 A generic enforcement mechanism

We now have all the necessary ingredients to construct a secure and precise enforcement mechanism for any testable hypersafety policy. This construction will be useful in two ways. We will use it in Section 5 to construct enforcement mechanisms for interesting policies by constructing efficient well-founded generators for these policies. We will also use it to show a lower bound on enforceable policies (Theorem 4.6).

Definition 4.3. *Let* $\mathcal{P}$ *be a testable hypersafety policy (specified as a total computable* $reject(M)$ *predicate), with a total computable well-founded generator* $g(I)$ *and a total computable extension function* $extend_{\mathcal{P}}(M, I, i)$*.*

We define a function $\mathsf{E}_{\mathcal{P}}$ *from programs to programs as follows. A state of* $\mathsf{E}_{\mathcal{P}}(C)$ *is a tuple* $\langle I, C, C', O\rangle$*, where* C *is the original program,* C' *is the current state of the program after input* I*, and* O *is a good output of the enforcement mechanism on* I *(or* $\mathsf{E}_{\mathcal{P}}(C)(I) = O$*). The initial state of* $\mathsf{E}_{\mathcal{P}}(C)$ *is a tuple* $\langle [], C, C, []\rangle$*. The transition relation of* $\mathsf{E}_{\mathcal{P}}(C)$ *is defined by the following rules:*

$$\text{OK}\ \frac{\begin{array}{c} M = map(\mathsf{E}_{\mathcal{P}}(C), g(I : i)) \\ C' \xrightarrow{i|o} C'' \quad reject(M \cup \{(I : i, O : o)\}) = \textbf{\textit{false}} \end{array}}{\langle I, C, C', O\rangle \xrightarrow{i|o} \langle I : i, C, C'', O : o\rangle}$$

$$\text{NOK}\ \frac{\begin{array}{c} M = map(\mathsf{E}_{\mathcal{P}}(C), g(I : i)) \\ reject(M \cup \{(I : i, O : o)\}) = \textbf{\textit{true}} \\ C' \xrightarrow{i|o} C'' \quad o' = extend_{\mathcal{P}}(M \cup \{(I, O)\}, I, i) \end{array}}{\langle I, C, C', O\rangle \xrightarrow{i|o'} \langle I : i, C, C''', O : o'\rangle}$$

The first rule says that, if the observation obtained by combining the recursive application of $\mathsf{E}_{\mathcal{P}}$ to $g(I : i)$ and the new observation that C is producing are allowed by the policy, then we just release the output of C.

The second rule says that, if the obtained observation is not allowed, we will correct the execution. We correct it by selecting a

new output using the consistent extension function extend$_\mathcal{P}$. The successor state for the monitored program can be an arbitrary program C''': as the program being monitored is definitely not compliant with the policy, we do no longer have to care about precision. C''' could for instance be a state that terminates (i.e. continuously emits the o_{end} output). While C''' can be chosen arbitrarily, the enforcement mechanism should choose it deterministically based only on its current state and input (in order to make $\mathsf{E}_\mathcal{P}(C)$ a deterministic program).

Notice that in the antecedent of our rules we do *not* check the output of the program C on the additional inputs from the generator, but the output of $\mathsf{E}_\mathcal{P}(C)$. This avoids inconsistency of corrections as we had in Example 13.

We have to show that the rules OK and NOK are a proper definition for a program.

Lemma 4.3. *For every state (I, C, C', O) of the enforcement mechanism, and for every input i, the transition relation is (1) total computable, and (2) deterministic.*

Proof. We show both properties by total induction on the well-founded order $\sqsubset$ on $\mathbf{I}^*$. So, suppose both properties (1) and (2) hold for all states (I_0, C_0, C_0', O_0) and input i_0 with $I_0 : i_0 \sqsubset I : i$.

The transition from (I, C, C', O) on input i is total computable, because (a) the transition relation on C is total computable, (b) the reject and extension functions are total computable, and (c) the computation of the map of $\mathsf{E}_\mathcal{P}(C)$ only needs a finite number of transitions on states (I_0, C_0, C_0', O_0) and inputs i_0 such that $I_0 : i_0 \sqsubset I : i$ (this follows from the fact that g is well-founded and hence all $I' \in g(I : i) \sqsubset I : i$). Hence by the induction hypothesis all these transitions are total computable.

The transition from (I, C, C', O) on input i is deterministic, because (a) the transition relation on C is deterministic, (b) the computation of the map of $\mathsf{E}_\mathcal{P}(C)$ only needs transitions on states (I_0, C_0, C_0', O_0) and inputs i_0 such that $I_0 : i_0 \sqsubset I : i$. Hence by the induction hypothesis all these transitions are deterministic. Now reject deterministically returns either true or false. For the false case, we are done. For the true case, since the extension function is indeed a function and its parameters are deterministically determined by the input state and input value, this function deterministically returns an o'. Finally, the state C''' in the output state is chosen arbitrarily but deterministically based on current state and input value. ■

Now that we have established that $\mathsf{E}_\mathcal{P}$ is a total computable function from programs to programs, we can prove its properties.

Theorem 4.3 (Security)**.** *Let $\mathcal{P}$ be a testable and non-empty hypersafety policy. Then $\mathsf{E}_\mathcal{P}(C) \in \mathcal{P}$ for any C.*

Proof. Let us say that a program C is I-level secure if it does not have any bad observation M_{bad} with for all $I' \in inputs(M_{bad})$, $I' \sqsubseteq I$.

We first show the following property: For all $I \in \mathbf{I}^*$, $\mathsf{E}_\mathcal{P}(C)$ is I-level secure. We prove this by complete induction on the well-founder order $\sqsubset$. So suppose the property holds for all inputs $I_1 \sqsubset I$. We prove it holds for I.

For the case where I is empty: Since the empty list is a minimal element under the $\sqsubset$ relation, we just have to show that the primitive observation $([], [])$ is not in $\mathcal{M}_\mathcal{P}$. This follows from the fact that $\mathcal{P}$ is non-empty: there is a program $C \in \mathcal{P}$, and every C has the observation $([], [])$.

For the case where I has the form $I_1 : i$, we show that map$(\mathsf{E}_\mathcal{P}(C), g(I_1 : i) \cup \{I_1 : i\}) \notin \mathcal{M}_\mathcal{P}$.

- For the subcase where the last output on this input was derived by the OK rule, this follows from the fact that reject$(M \cup \{(I_1 : i, O : o)\}) =$ **false** for $M =$ map$(\mathsf{E}_\mathcal{P}(C), g(I_1 : i))$.
- For the subcase where the last output on this input was derived by the NOK rule, we can use the induction hypothesis. All the input streams in $g(I_1 : i) \cup \{I_1\}$ are $\sqsubset I_1 : i$. Hence, the first argument to the extend function is an allowed observation.
  By the property of the extend function, we then also get for this subcase that map$(\mathsf{E}_\mathcal{P}(C), g(I_1 : i) \cup \{I_1 : i\}) \notin \mathcal{M}_\mathcal{P}$.

Now we can show that $\mathsf{E}_\mathcal{P}(C)$ is I-level secure. Suppose there is an M_{bad} with all elements of inputs$(M_{bad}) \sqsubseteq I$. Then we can easily see that for all $I' \in inputs(M_{bad})$ it holds that map$(\mathsf{E}_\mathcal{P}(C), g(I) \cup \{I\}) \notin \mathcal{M}_\mathcal{P}$. (For $I' \sqsubset I$ this follows from the induction hypothesis and the fact that g is well-founded, for $I' = I$ we have just shown it.) But then the definition of test generator tells us that $M_{bad} \notin \mathcal{M}_\mathcal{P}$.

Finally, using this fact that $\mathsf{E}_\mathcal{P}(C)$ is I-level secure for all I, we can apply Lemma 4.1 and we get that $\mathsf{E}_\mathcal{P}(C) \in \mathcal{P}$. ■

Theorem 4.4. $\mathsf{E}_\mathcal{P}(C)$ *is precise.*

Proof. We have to show that C and $\mathsf{E}_\mathcal{P}(C)$ have exactly the same primitive observations when $C \in \mathcal{P}$. We show this by complete induction on the well-founded partial order $\sqsubset$ on $\mathbf{I}^\star$.

Assume that C and $\mathsf{E}_\mathcal{P}(C)$ have the same primitive observations (I', O') for all $I' \sqsubset I : i$. We have to show that $\mathsf{E}_\mathcal{P}(C)(I : i) = C(I : i)$.

From the induction hypothesis, it follows that the derivation of the last step of $\mathsf{E}_\mathcal{P}(C)$ processing input $I : i$ was done by the OK rule: $\mathsf{E}_\mathcal{P}(C)$ is applied only on $I' \sqsubset I : i$, hence the induction hypothesis applies, and $\mathsf{E}_\mathcal{P}(C)$ has the same outputs as C on $g(I : i)$. Hence, map$(\mathsf{E}_\mathcal{P}(C), g(I : i) \cup \{(I : i, O : o)\})$ is actually an observation on C, and since $C \in \mathcal{P}$ it follows that map$(\mathsf{E}_\mathcal{P}(C), g(I : i)) \cup \{(I : i, O : o)\} \notin \mathcal{M}_\mathcal{P}$, and hence the call to reject must return false. As a consequence, the primitive observation of $\mathsf{E}_\mathcal{P}(C)$ on $I : i$ is the same as the primitive observation of C on $I : i$. ■

Theorem 4.5. $\mathsf{E}_\mathcal{P}(C)$ *is black box.*

Proof. By the same induction technique as in Lemma 4.3 one can show that map$(\mathsf{E}_\mathcal{P}(C_1), g(I : i))$ is equal to map$(\mathsf{E}_\mathcal{P}(C_2), g(I : i))$ if $C_1 \sim C_2$. ■

4.3.4 A lower bound

Finally, we can establish an interesting lower bound for the set of black box enforceable policies.

Theorem 4.6. *Every non-empty testable hypersafety policy is black box enforceable.*

Proof. For any non-empty testable hypersafety policy:

- the reject predicate is total computable by definition of testable.
- Proposition 3.2 gives us a total computable extension function.
- Lemma 4.2 gives us a well-founded generator.

Hence, we can construct an enforcement mechanism as in Definition 4.3, and Theorems 4.3, 4.4 and 4.5 tell us that this enforcement mechanism is secure, precise and black box. ■

Obviously, this construction is very inefficient, but from a theoretical point of view this lower bound on the class of black box enforceable policies is interesting. In the next section, we turn to constructing concrete enforcement mechanisms.

5. Instances

Given a hypersafety policy $\mathcal{P}$, the steps that need to be taken to enforce the policy using our generic enforcement mechanism are:

1. specify a total computable reject function that decides membership in $\mathcal{M}_{\mathcal{P}}$. Prove that the set of observations for which reject returns true is indeed maximal in the sense of definition 3.3.
2. specify a total computable test generator function, and prove it satisfies the property required of a test generator as stipulated in definition 4.2.
3. specify a total computable extension function, and prove that it has the property required of such a function as specified in definition 3.5. Since our enforcement mechanism only applies extend to observations M with inputs(M) of the form $g(I : i) \cup \{I\}$, it is sufficient to define extend for such input parameters.

In this section, we construct interesting instances and relate them to existing enforcement mechanisms. We also discuss some optimizations to improve the performance of the generic enforcement mechanism.

5.1 1-safety policies

The simplest instantiation of our mechanism is the case where $\mathcal{P}$ is a 1-safety policy.

1. The generator g maps every I on $\{\}$ (cfr Example 11).
2. The reject function can be simplified to just a boolean predicate on primitive observations. Maximality ensures that any primitive observation that is not rejected can be further extended in a way acceptable to the policy. Hence, reject is a *benevolent* detector in the sense of Hamlen et al. [15].
3. The extend function just takes a primitive observation (I, O) and a new input i and returns a valid output o.

Since g always returns the empty set, no recursive calls of the enforcement mechanism are needed: processing an input/output event just requires one call to reject and one call to extend.

Even this simplest instantiation admits interesting examples.

Example 16. *For the NSAR policy, reject was specified in Example 1, the generator always returns the empty set, and an extend function was defined in Example 8.*

Example 17. *Consider the policy that forbids any primitive observation of length bigger than 5. (Also counting the o_{end} output values) This policy is empty (*no *program complies), and hence not enforceable. The reject function for this policy is the constant function that returns true.*

Suppose the policy requires termination after at most 5 steps. This policy is enforceable. The reject function is:

$$reject((I, O)) \quad = \quad \exists o \neq o_{end} : occurs(o, (drop\ 5\ O))$$

The extend function always returns the termination event.

$$extend(I, O, i) = o_{end}$$

If a program fails to terminate before its 5th step, the enforcement mechanism will force it to terminate. For instance the sum program from Listing 1 will process the input stream $[1 : 1 : 1 : \ldots]$ *to the output stream* $[1 : 2 : 3 : 4 : 5 : o_{end} : o_{end} : o_{end} : \ldots]$ *when executed under this enforcement mechanism.*

5.2 Non-interference with two levels

Let pr be a total idempotent function from finite input streams to finite input streams. That is, for all $I \in \mathbf{I}^\star$, $pr(I) = pr(pr(I))$. We think of pr as a projection that removes confidential information from the input stream.

A program C is *non-interferent w.r.t. pr* iff

$$\forall I, I' \in \mathbf{I}^\star : pr(I) = pr(I') \implies O|_L = O'|_L,$$

where $C(I) = O$ and $C(I') = O'$.

The projection pr can be instantiated in many ways, and our enforcement mechanism can handle all these instantiations.

- $pr(I) = I|_{L,HD}$ where $I|_{L,HD}$ is the resulting stream after replacing the H input values in I with default values: this is a variation of NI where *content* of input events is secret but the *occurrence* of the input event is not. With the optimizations in Section 5.4 our generic enforcement mechanism reduces to standard secure multi-execution [11, 22].
- $pr(I) = I|_L$: models standard NI as in Example 2. Our generic enforcement mechanism defines a reactive variant of secure multi-execution as in [5, 30].
- pr can more generally project to values that depend on all previous values in the input stream. This can model for instance NI with stateful declassification policies [27]. Our generic enforcement mechanism even improves on the mechanism in [27], as it does not require declassify annotations for precision since in [27] a declassify operator is just a directive indicating that a particular value is computed by the release function.

Construction of the reject predicate and the test generator for this policy is similar to Example 6 and Example 12. For this more general case, the following definitions work:

$$\text{reject}(M) = \begin{cases} \textbf{true} & \exists M_{bad} = \{(I, O), (I', O')\}, \\ & M_{bad} \subseteq \overline{M} \text{ s.t. } pr(I) = pr(I') \\ & \quad \text{and } O|_L \neq O'|_L, \\ \textbf{false} & \text{otherwise.} \end{cases}$$

$$g(I) = \{pr(I)\} \setminus \{I\}$$

The well founded order for the generator is the smallest ordering such that for all I and i, $I \sqsubset I : i$ and if $pr(I) \neq I$ then $pr(I) \sqsubset I$. A sufficient condition is that pr(I) has smaller or equal length as I, and this is satisfied in all instances for pr mentioned above.

Since extend is only called for $M = \{(I, O), (I' : i', O' : o')\}$ where $pr(I : i) = I' : i'$, this function is as below where o_H is an arbitrary value such that $lvl(o_H) = H$.

$$\text{extend}_{\mathcal{P}_{NI1}}(M, I, i) = \begin{cases} o_H & \text{if } lvl(i) = H \text{ where } lvl(o_H) = H \\ o' & \text{if } lvl(i) = L. \end{cases}$$

5.3 Non-interference for multiple levels

It is relatively straigtforward to extend all the variants of NI above to multiple confidentiality levels. We illustrate this for standard NI. Let $\langle \mathcal{L}, \leq \rangle$ be a complete lattice of security levels with a top level ($\top$) and a bottom level ($\bot$), and let lvl be a function from $\mathbf{I} \cup \mathbf{O}$ to $\mathcal{L}$. A program C is *non-interferent* with respect to lvl iff $\forall I, I' \in \mathbf{I}^\star : I|_l = I'|_l \implies O|_l = O'|_l$, where $C(I) = O$ and $C(I') = O'$, and $I|_l$ filters out all i with $lvl(i) \not\leq l$ (and similarly for $O|_l$).

$$\text{reject}(M) = \begin{cases} \textbf{true} & \exists M_{bad} = \{(I, O), (I', O')\}, \\ & M_{bad} \subseteq \overline{M} \text{ s.t. } I|_l = I'|_l \\ & \quad \text{and } O|_l \neq O'|_l, \text{ for some } l \\ \textbf{false} & \text{otherwise.} \end{cases}$$

$$g(I) = \{I|_l \quad | \quad l \in \mathcal{L}\} \setminus \{I\}$$

$$\text{extend}_{\mathcal{P}_{NI}}(M, I, i) = \begin{cases} o_\top & \text{if } lvl(i) \neq \bot \text{ where } lvl(o_\top) = \top, \\ o_\bot & \text{if } lvl(i) = \bot. \end{cases}$$

(taking into account that extend is only called for $M = \{(I, O)\} \cup \{(I' : i', O' : o') \mid I' : i' \in g(I : i)\}$ and $(I_\perp : i_\perp, O_\perp : o_\perp)$ is in M, where $I_\perp : i_\perp = (I : i)|_\perp$)

5.4 Optimizations and extensions.

The construction in Definition 4.3 is designed to minimize the state of the enforcement mechanism (thus making the security and precision proofs relatively simple). However, it is very suboptimal in terms of execution time. The recursive calls to the enforcement mechanism will often recompute observations on $\mathsf{E}_\mathcal{P}(C)$ over and over many times.

A standard optimization for such cases is *memoization*: cache the results of calls to $\mathsf{E}_\mathcal{P}(C)$ in some data structure and reuse them from the cache instead of recomputing them.

A data structure that is particularly well suited for optimization of our enforcement mechanism is an array of entries $\langle I_j, C_j, O_j \rangle$, where for each j, it holds that $\langle I_j, C, C_j, O_j \rangle$ is the state reached by the enforcement mechanism after processing input stream I_j. This data structure (1) caches all the output streams that $\mathsf{E}_\mathcal{P}(C)$ produces on any of the I_j (or prefixes of I_j), and (2) it is efficient to update the data structure for growing I_j. Essentially, such a data structure maintains a set of alternative executions of C (taking into account corrections to the execution where necessary), in a way that is very close to secure multi-execution [11]. Proving the correctness of this optimization for an arbitrary policy and relating it to secure multi-execution is an interesting avenue for future work.

Finally, our generic construction also constructs secure and precise enforcement mechanism for many other policies. However, it is not always obvious that efficient test generators exist. For example, the k-safety policy from Example 3 is enforceable, but we have not been able to come up with an efficient generator for this policy. Another interesting line of future work is to look for efficient generators for other hypersafety policies.

6. Discussion

Many enforcement mechanisms used in practice make no, or very limited use of analysis of the code of the programs they are monitoring. Operating system kernels, virtual machine monitors, and application firewalls treat the programs they are guarding as black boxes. Hence it is interesting to understand what exactly can be enforced under this black box constraint.

Figure 1 summarizes our results on upper and lower bounds for the class of black box enforceable policies. The shaded region corresponds to the class of policies that are black box enforceable as defined in Definition 4.1. First, in Theorem 4.1 we have shown that any policy that is black box enforceable must be hypersafety. More importantly, in Theorem 4.2 we have shown that such policies must be semi-testable hypersafety policies. Based on this theorem, the policy in Example 4 is not black box enforceable. Finally, in Theorem 4.6 we have shown that every non-empty testable hypersafety policy is black box enforceable. The empty policy is not enforceable since there is no program in this policy and the mechanism is just a program.

An interesting question is how tight the bounds we establish for black box enforceable policies are. There is a gap between the sufficient condition, i.e. that the policy is testable hypersafety, and the necessary condition, i.e. that the policy is semi-testable hypersafety. Clearly, the upper bound is not tight: an example 1-safety policy that is semi-testable but not enforceable is the policy that requires programs to only output terminating λ-terms. The complement of the maximal set of bad observations is recursively enumerable, hence the policy is semi-testable. However an enforcement mechanism would have to decide termination. The question now is whether the subset marked with ??? on Figure 1 is empty. For the lower bound, we believe it might actually be tight, but we have not been able to prove this yet.

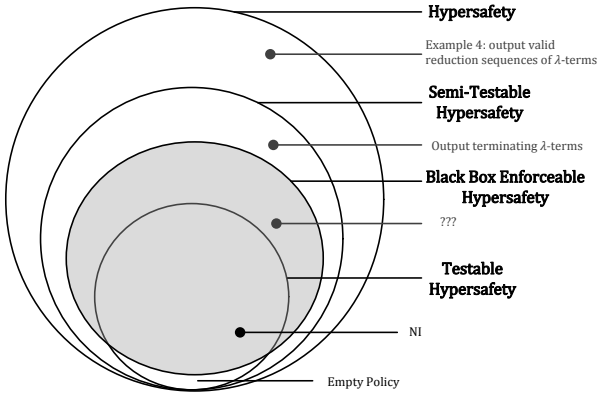

Figure 1. Lower and upper bounds of black box enforceable hypersafety policies.

One important aspect of our results is the fact that they hold for our specific model of reactive programs where programs can not diverge on a single input value. On the one hand, this is a positive aspect as many programs satisfy this constraint, and under this program model enforcement mechanisms can be guaranteed not to diverge while still being allowed to speculatively execute programs until the next observable output. Divergence of a program could perhaps be modeled by having the program react with a special output value o_{busy} to any input value while diverging, but this *does* impact the policies that are enforceable, as output values such as o_{busy} become observable, and hence policies have to take them into account. On the other hand, this constraint on programs makes enforcement easier, and some of our results will clearly not hold for programs that do diverge on some inputs. An important question for future work is to explore the impact of variations of the program model on our results.

Another interesting question to ask is: what is the relation between policies enforceable by program rewriting [15], and the black box enforceable policies (under our model of programs)? Clearly, the black box enforceable policies are also enforceable by rewriting. But rewriters seem to have additional power as they can inspect and transform the code of the program at will. However, we have not been able to find policies that are enforceable by rewriting but not black box enforceable. It seems that the power to edit input/output events and explore alternative runs is very close to the power to rewrite and inspect code – at least if one operates under the constraints that (1) the resulting enforcement mechanism must be secure and precise, and (2) programs must not diverge on single inputs.

Besides the bounds on enforceability, this article also introduces a general enforcement mechanism that can be used to build practical enforcement mechanisms. This was demonstrated in Section 5, where we have seen that several different flavors of enforcement mechanisms for NI are particular instances of our techniques with efficient test generators. We believe that the existence of efficient test generators is the key to practical enforceability. In fact, one can prove that the existence of an efficient enforcement mechanism implies the existence of an efficient test generator: one can take as $g(I)$ all the input streams I' that the efficient enforcement mechanism feeds to the untrusted program while processing input I. It is not too hard to see that this defines a test generator. Hence, we believe one of the interesting conclusions of this work is that the key to developing a black box enforcement mechanism for a hypersafety policy is to come up with an efficient test generator.

7. Related Work

7.1 Enforcement power of enforcement mechanisms

Erlingsson and Schneider [12, 25] initiated the investigation of dynamic enforcement of security policies via execution monitoring (EM). Schneider [25] demonstrated that for a security policy to be enforceable by EM, the policy has to be a safety property on execution traces. He also introduced a class of monitors called *security automata* that can be used as recognizers for safety properties. Such an automaton is capable of observing system actions, recognizing violations and terminating the observed system (called the *target*) when it is about to perform a violation. A target system is allowed to perform a computation step if and only if the respective security automaton can transition to some next state by performing the same step. Our proposed mechanisms can be seen as a generalization of Schneider's enforceable security policies, as they can be instantiated on nonempty 1-safety policies.

Schneider's work was refined by Viswanathan [28] who showed that a property has to be decidable in order to be enforceable. Formally, Viswanathan argued that the class of dynamically enforceable policies with EM is actually equivalent to the class of co-recursively enumerable ($coRE$) properties, also known as the class Π_1 of the arithmetic hierarchy. However, it was later shown by Hamlen et al. [15] that $coRE$ is actually an upper bound on the class of EM enforceable properties. The reason is that some policies in $coRE$ are not enforceable because the monitor might not have enough power to intervene. For instance, it is impossible to enforce a policy that forbids all the interventions available to the EM.

Fong [14] analyzed the impact of memory-limitations on the enforcement mechanisms. He introduced *shallow history automata* which only store information about the occurrence of past events and *not* about their order. Despite this limitation, many interesting properties were shown to remain enforceable. Talhi et al. [26] extended Fong's work by introducing *bounded history automata*, combining features of security and edit automata and having bounded memory. In addition, they proposed an algorithm that decides whether there exists a bounded history automaton for a policy in a language from a subclass of the regular languages (called the locally testable languages [21]). Falcone et al. [13] investigated the properties that are recognizable by different automata models with a finite set of control states and enforcement operations in terms of the safety-progress hierarchy [9]. In addition, they presented a systematic technique to synthesize monitors from an automaton recognizing some safety, guarantee, obligation or response properties. Hamlen et al. [15] defined the class of security policies enforceable by program rewriting (RW) and called it $RW\text{-}enforceable$. However, they did not provide a precise characterization of this class and even argued that a characterization in complexity theoretic terms might not exist. Finally, Chabot et al. [8] explored the problem of policy enforcement w.r.t. a trace universe. They showed that a truncation enforcement mechanism is more powerful if it only considers the possible traces and not all traces in the trace universe. They also presented an algorithm that takes as an input a finite state automaton (i.e. the policy) and a finite state transition system (i.e. the target system) and checks if the policy is enforceable. Additionally, if the policy is found to be enforceable, the algorithm returns a secured version of the target system.

Ligatti et al. [18, 19] introduced edit automata, capable of enforcing a class of non-safety (including purely liveness) policies called *infinite renewal* properties. They argued that infinite renewal properties subsume the computable safety properties. Edit automata are a black box enforcement mechanism, capable of inserting and deleting system actions as well as terminating a target system in the case of a policy violation. The authors also showed how to construct an enforcement mechanism that provably enforces any "reasonable" infinite renewal property [18]. Edit automata are similar to our approach in that they offer a black box mechanism capable of monitoring and/or editing the input/output behavior of programs. However, to enforce infinite renewal properties, they have to buffer program actions – something that is undesirable for reactive programs. Also, edit automata are not capable of exploring alternative executions.

Unlike edit automata, mandatory-result automata [20] have an interface to interact with a target system: they get requests from the system and send outputs back to the system but do not enforce policies that are more expressive than those investigated by Basin et al. [4]. Basin and his coauthors revisited Schneider's work and proposed to distinguish between actions that are under the control of the enforcement mechanism and actions that are merely observable (for instance the passing of time), i.e. the mechanism cannot prevent their execution. For this refinement of the problem, the authors presented necessary and sufficient conditions to characterize when a policy is EM enforceable, based on their generalized notion of safety. In addition, they studied the problem of whether a policy is enforceable for several specification languages. They also showed how to synthesize an enforcement mechanism for a given enforceable policy and gave complexity results.

Clarkson and Schneider [10] were the first to propose *hyperproperties* (sets of trace properties) as a model for security policies. They generalized safety and liveness to hypersafety and hyperliveness, and proposed a static verification approach for verifying compliance with a hypersafety property. The fact that hypersafety aligns so well with black box enforceability is additional evidence that their classification of security policies as hypersafety or hyperliveness is a useful taxonomy of policies.

7.2 Existing black box enforcement mechanisms for non-interference

Besides the general purpose black box enforcement mechanisms (such as the different variants of execution monitors and edit automata) discussed above, a considerable amount of research has been done on dynamic enforcement of non-interference, some of it assuming only black box access to programs. Le Guernic's PhD thesis [17] presents an extensive survey of the state of the art in dynamic information flow control circa 2007. The major more recent results are presented next.

There have been several proposals for information flow runtime monitors. Sabelfeld et al. proposed such monitors for DOM-like structures [24], dynamic code evaluation [1] and timeouts [23]. Austin and Flanagen [2] presented alternative techniques that have the potential of being more permissive. All these monitors are useful and relatively efficient, however a common problem is that they are not precise, i.e. they over-approximate the problem and as a result sometimes block secure executions.

A secure and precise dynamic enforcement technique is secure multi-execution (SME), developed independently by several researchers [11, 16, 29]. Khatiwala et al. [16] proposed a technique called *Data Sandboxing*, which partitions the program into two programs at source-code level and then uses system call interposition to control the output channels. In a follow-up article, Capizzi et al. [29] abandoned source-code partitioning; instead, they proposed to run two processes for the public and secret levels respectively and as a result provide strong confidentiality guarantees. Devriese and Piessens [11] independently proposed the related technique that they called SME. The overall idea of SME is to keep a program secure by separating computations at different security levels. To that end, the original program is executed multiple times, once for each security level and giving inputs and outputs special treatment; this technique guarantees security and precision.

Austin and Flanagan [3] developed a more efficient implementation of SME based on multi-faceted evaluation. Rafnsson and Sabelfeld [22] upgraded the SME technique to distinguish between presence and content of secret messages on a channel, to perform declassification and to make SME precise for observers having access to more than one level. Zanarini et al. [30] proposed combining SME with execution monitoring. The idea is at each step to compare the current execution with what SME would produce in order to detect (and report) actions that offend the policy. Similarly to Rafnsson and Sabelfeld [22], they use a scheduling strategy that preserves the interleaving of events from different security levels, thus increasing the precision of SME. Vanhoef et al. [27] proposed an SME-based technique for enforcing information flow policies with support for stateful declassification for event-driven programs. They used a projection function to specify what information about an event can be declassified and a stateful release function to specify aggregate information about secret events seen so far that can be declassified. It is clear that the mechanisms presented in this work are a generalization and at the same time go beyond a number of flavors of SME, for instance the vanilla one with 2-levels, the one with n-levels [11], and with stateful declassification [27].

8. Conclusion

We have studied security policy enforcement mechanisms for deterministic reactive programs that respond to input events in finite time with an observable output event. Our enforcement mechanisms are *black box*: the enforcement mechanism only needs to be able to (1) intercept and edit the input/output events of the untrusted program, and (2) run multiple isolated copies of the untrusted program in order to test the behaviour of the program on alternative inputs. The enforcement mechanism can *not* do (or more positively: does not *need* to do) analysis on the source or machine code of the program.

We have given a constructive proof that under these assumptions any testable hypersafety policy is enforceable: if it is decidable whether an observation can not be produced by any program in the policy, then the policy is black box enforceable by a generic enforcement mechanism. The construction in the proof is inefficient, but one can construct more efficient enforcement mechanisms by providing a reject function, an efficient test generator and an extension function as inputs to the general construction.

We have also shown an upper bound on black box enforceable policies: any enforceable policy is a semi-testable hypersafety policy, i.e. the observations that are possible for *some* program in the policy are enumerable.

While these lower and upper bounds are of theoretical interest, we have also shown that our generic enforcement mechanism is of practical interest by instantiating it to concrete enforcement mechanisms for a number of relevant policies.

Acknowledgments

The authors are grateful to Deepak Garg for proofreading this paper and suggesting numerous improvements. This research is partially funded by the Research Fund KU Leuven, by the Research Foundation - Flanders (FWO), by the Italian project TENACE PRIN (n. 20103P34XC), and by the project EU-IST-NOE-NESSOS.

References

[1] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *CSF*, pages 43–59, 2009.

[2] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *PLAS*, pages 3:1–3:12, 2010.

[3] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *POPL*, pages 165–178, 2012.

[4] D. Basin, V. Jugé, F. Klaedtke, and E. Zălinescu. Enforceable security policies revisited. *TISSEC*, 16(1):3:1–3:26, June 2013.

[5] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive noninterference for a browser model. In *5th International Conference on Network and System Security*, pages 97–104, 2011.

[6] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In *CCS*, pages 79–90, 2009.

[7] R. Capizzi, A. Longo, V. N. Venkatakrishnan, and A. P. Sistla. Preventing information leaks through shadow executions. In *ACSAC*, pages 322–331, 2008.

[8] H. Chabot, R. Khoury, and N. Tawbi. Extending the enforcement power of truncation monitors using static analysis. *Computers & Security*, 30(4):194–207, June 2011.

[9] E. Y. Chang, Z. Manna, and A. Pnueli. Characterization of temporal property classes. In *ICALP*, pages 474–486, 1992.

[10] M. R. Clarkson and F. B. Schneider. Hyperproperties. *JCS*, 18:1157–1210, September 2010.

[11] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *IEEE S&P*, pages 109–124, 2010.

[12] U. Erlingsson and F. B. Schneider. Sasi enforcement of security policies: A retrospective. In *1999 Workshop on New Security Paradigms*, pages 87–95, 2000.

[13] Y. Falcone, L. Mounier, J.-C. Fernandez, and J.-L. Richier. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods in System Design*, 38(3):223–262, 2011.

[14] P. W. L. Fong. Access control by tracking shallow execution history. In *IEEE S&P*, pages 43–55, 2004.

[15] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *TOPLAS*, 28(1):175–205, 2006.

[16] T. Khatiwala, R. Swaminathan, and V. N. Venkatakrishnan. Data sandboxing: A technique for enforcing confidentiality policies. In *ACSAC*, pages 223–234, 2006.

[17] G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007.

[18] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *Int. J. of Inf. Sec.*, 4(1–2):2–16, February 2005.

[19] J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of non-safety policies. *TISSEC*, 12(3):1–41, 2009.

[20] J. Ligatti and S. Reddy. A theory of runtime enforcement, with results. In *ESORICS*, pages 87–100, 2010.

[21] R. McNaughton and S. A. Papert. *Counter-Free Automata (M.I.T. Research Monograph No. 65)*. The MIT Press, 1971.

[22] W. Rafnsson and A. Sabelfeld. Secure multi-execution: fine-grained, declassification-aware, and transparent. In *CSF*, pages 33–48, 2013.

[23] A. Russo and A. Sabelfeld. Securing timeout instructions in web applications. In *CSF*, pages 92–106, 2009.

[24] A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In *ESORICS*, pages 86–103, 2009.

[25] F. Schneider. Enforceable security policies. *TISSEC*, 3(1):30–50, 2000.

[26] C. Talhi, N. Tawbi, and M. Debbabi. Execution monitoring enforcement under memory-limitation constraints. *Inf. Comput.*, 206(2-4):158–184, Feb. 2008.

[27] M. Vanhoef, W. De Groef, D. Devriese, F. Piessens, and T. Rezk. Stateful declassification policies for event-driven programs. In *CSF*, 2014.

[28] M. Viswanathan. *Foundations for the Run-time Analysis of Software Systems*. PhD thesis, University of Pennsylvania, 2000.

[29] A. R. Yumerefendi, B. Mickle, and L. P. Cox. Tightlip: Keeping applications from spilling the beans. In *4th USENIX Conference on Networked Systems Design & Implementation*, pages 12–12, 2007.

[30] D. Zanarini, M. Jaskelioff, and A. Russo. Precise enforcement of confidentiality for reactive systems. In *CSF*, pages 18–32, 2013.