The Specification and Compilation of Obligation Policies for Program Monitoring

Cheng Xu Philip W. L. Fong
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
{ cxu, pwlfong }@ucalgary.ca

ABSTRACT

An extensible software system must protect its resources from being abused by untrusted software extensions. The access control policies of extensible software systems are traditionally enforced by some form of reference monitors. Recent study of access control policies advocates the use of obligation policies, which impose behavioural constraints to the future actions of the accessor after the access is granted. It is argued that obligation policies provide continuous protection to the system.

Not all obligation policies can be enforced by reference monitors. We argue that humans have long recognized the unenforceability of naively formulated obligation policies, and have devised standard policy idioms to cope with the issue. We therefore developed tool support to assist a policy developer in using such policy idioms. First, we designed a policy language to capture the idiomatic elements of obligation policies, in such a way that the elements are modular and composeable. Second, we designed a type system for capturing patterns of policy composition that preserve enforceability, such that well-typed policies are enforceable. Third, we designed a compilation algorithm that compiles well-typed policies into reference monitors. Such a framework helps policy developers articulate obligation policies and refine them into enforceable ones.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access Controls; F.4.1 [**Mathematical Logic**]: Temporal Logic

General Terms

Security, Language, Theory

Keywords

Enforceability, guarantee property, obligation policy, policy compilation, policy language, reference monitor, safety property, type system

ASIACCS '12, May 2–4, 2012, Seoul, Korea.

1. INTRODUCTION

Complex software systems are usually structured as an ***extensible system***, which is composed of a core system and a set of third-party extensions. The extensions augment the functionality of the core system. For example, a mobile phone platform, such as Android, can be extended by installing mobile applications; word processors, like Microsoft Word™, support macro programming; and Internet web browsers, like Firefox, can be extended with plugins and applets. A threat to extensible systems is the lack of assurance about the extensions' behaviour. A malicious extension can compromise the core system because the two share the same set of resources.

One way to address the security challenge above is to deploy a ***reference monitor*** [4] to ensure that the behaviour of the extensions comply with pre-defined security policies. A reference monitor intercepts actions of the extensions and takes remedial measures on actions that may cause damage to the integrity of the system (e.g., by suppressing the offending actions, or terminating the offending extensions). A modern implementation of this enforcement mechanism is ***Inlined Reference Monitor (IRM)*** [13], in which, the ***target program*** (extension) is modified, through a trusted program rewriter, to include the functionality of a reference monitor.

Previous work [20, 12] noted that some security properties that need to be imposed on untrusted programs are ***obligation policies*** [8, 23]. While a traditional access control policy either grants or denies access, an obligation policy grants access "with strings attached": access is granted with the condition that future actions of the accessor conform to some behavioural constraints. For example, consider the *Log-After-Access-Denial* policy presented in [12]: when a user attempts to access a document for which he lacks the required credentials, every subsequent attempt to access documents by that user must be logged.

Unfortunately, not all obligation policies are enforceable by a reference monitor. Schneider [37] showed that enforceable properties must be safety properties. Yet, Dougherty *et al.* [12] pointed out that obligation policies relevant to software security are safety and guarantee properties as well as their boolean combinations [10].

Devising strategies to cope with unenforceability has been an important theme of obligation policy research. Irwin *et al.* [23, 34] proposed an alternative security goal — accountability. Assuming that other subjects behave properly (in a technical sense), if a subject has enough permissions to carry out the obligations imposed on her, then she is accountable

for the unfulfillment of the obligation. That is, although the obligation policy cannot be enforced, at least one can reasonably hold a subject accountable for failure. Dougherty *et al.* [12] attempted to approximate an unenforceable obligation policies by a safety properties (for detecting failure) and a guarantee property (for detecting fulfillment). Nevertheless, some execution traces may still be left as unclassified by this approximation.

We argue that humans have long noticed the unenforceability of naively formulated obligation policies, and have devised standard policy idioms to cope with the issue. Specifically, notions such as deadlines, rewards and penalties are created to turn an otherwise unenforceable policy into an enforceable one. For example, the obligation policy "*return book eventually*" may not be enforceable, but its variant, "*return book in 3 weeks, or else a penalty of* $1 *per day will be imposed*", is a perfectly enforceable policy. For us, the research opportunity is therefore to assist policy developers in adopting such standard policy idioms, and ensuring the usage of the idioms results in obligation policies that are enforceable.

The contribution of this work is the design of tool support for helping policy developers to compose enforceable obligation policies out of idiomatic policy elements. We envision the workflow of an obligation policy developer to involve three stages: specification, implementability check, and implementation (Fig. 1). Our specific contributions can be classified according to the stages.

1. The ***specification*** stage involves the translation of an informal obligation policy (in natural language) to a formal one. In Sec. 3, we propose a policy language for specifying obligation policies. Inspired by [20], our language is a temporal logic, and thus has a formal semantics. What distinguishes our policy language is that the language captures standard idioms of obligation policies (e.g., time window, reward/penalty) in a modular and composeable manner (as opposed to a monolithic syntax).

2. The policy developer then performs an ***implementability check***, to determine if the obligation policy can be enforced by a reference monitor. The result of the implementability check will be fed back to the policy developer, who can either go back to the specification stage, or move forward to the implementation stage. Sec. 4 presents a type system for determining if an obligation policy written in our policy language is either a safety or guarantee property [10]. The type system captures patterns of policy composition that preserve enforceability. We demonstrate through a case study that the type system can provide feedback to the policy developer so that she can refine the obligation policy into an enforceable one.

3. The ***implementation*** stage involves the construction of an enforcement mechanism for an enforceable obligation policy. Sec. 5 reports a compilation algorithm that we developed for compiling well-typed obligation formulae to their corresponding enforcement mechanisms. The compilation algorithm is directed by the type system. In a sense, the compilation algorithm justifies the soundness of our type system: for each pattern of policy composition that is identified by the type system to be preserving enforceability, there is a corresponding construction of an enforcement mechanism for the composite policy out of the enforcement mechanisms of the component policies.

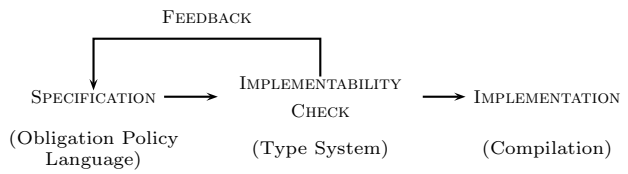

Figure 1: Workflow for obligation policy development

2. PRELIMINARIES

This section introduces notions that will be used throughout this paper.

Basic notations.

Given a set S, $[S]^k$ denotes the set of all k-element subsets of S, and $[S]^{<\omega}$ denotes the set of all finite subsets of S. The notation 2^S denotes the powerset of S (i.e. the set of all subset of S).

Program model.

Recall that our motivation is the monitoring of untrusted programs to ensure compliance to obligation policies. We describe here the program model that will be adopted throughout this paper.

At run time, a program performs actions that belong to a set Σ. A program execution can be characterized by a ***trace***, which is either a finite *or* infinite sequence of actions. Intuitively, a finite trace represents either a terminated execution or a partial execution; an infinite trace represents a non-terminating execution. (This representation is similar to the program models of [37, 27].) We denote by Σ^* the set of finite traces over Σ, and by Σ^ω the set of infinite traces over Σ. Then, $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ is the set of all traces over Σ. The empty sequence is denoted by ϵ. Given $u \in \Sigma^*$ and $v \in \Sigma^\infty$, we write $u \cdot v$ (or simply uv) to denote the concatenation of u and v. If $w = u \cdot v$, we say u is a ***prefix*** of w, v is a ***suffix*** of w, and we write $u \preceq w$ and $v \sqsubseteq w$. Specifically, if $u \neq w$, then we write $u \prec w$; if $v \neq w$, then we write $v \sqsubset w$. We also write w/v to denote u, and $w - u$ to denote v.

Properties.

A ***property*** Φ is a set of traces (i.e., $\Phi \subseteq \Sigma^\infty$). If $w \in \Phi$ then w ***satisfies*** Φ. Otherwise, w ***violates*** Φ. In [12] obligation policies are taken to be of two kinds: (a) obligation policies that can be enforced by reference monitors are safety properties, and (b) obligation policies for which fulfillment can be detected by reference monitors are guarantee properties. Since our program model involves finite traces, we adapt the definition of safety and guarantee property for our program model.

Property Φ is a ***safety property***[1] iff:

$$\forall w \in \Sigma^\infty : (w \notin \Phi \Rightarrow \exists u \preceq w : (u \notin \Phi \wedge \forall v \in \Sigma^\infty : uv \notin \Phi))$$

Property Φ is a ***guarantee property*** iff:

$$\forall w \in \Sigma^\infty : (w \in \Phi \Rightarrow \exists u \preceq w : (u \in \Phi \wedge \forall v \in \Sigma^\infty : uv \in \Phi))$$

A safety property Φ is a ***k-bounded safety property*** iff:

$$\forall u \in \Sigma^* : (|u| \geq k \wedge u \in \Phi \Rightarrow \forall v \in \Sigma^\infty : uv \in \Phi)$$

Intuitively, a violation of a k-bounded safety property can only occur at or before the kth step. A reference monitor can safely give up detecting violation after failing to find one in the first k steps. A guarantee property Φ is a ***k-bounded guarantee property*** iff:

$$\forall u \in \Sigma^* : (|u| \geq k \wedge u \notin \Phi \Rightarrow \forall v \in \Sigma^\infty : uv \notin \Phi)$$

A fulfilment of a k-bounded guarantee property can only occur at or before the kth step. A reference monitor can safely give up detecting fulfillment after failing to find one in the first k steps[2].

Action assertion language.

We assume that an ***action assertion language*** has been provided for classifying actions. Given the set Σ of actions, an action assertion language $\mathcal{L}_\Sigma$ is a pair $\langle \mathcal{P}, \Vdash \rangle$, where $\mathcal{P}$ is a countable set of ***action propositions***, and $\Vdash \subseteq \Sigma \times \mathcal{P}$ is a binary relation specifying which proposition is satisfied by which action. ***Action formulae*** (ψ) denote boolean combinations of action propositions. The syntax of an action formula is given below:

$$\psi ::= \texttt{true} \mid \texttt{false} \mid p \mid \texttt{!}\,\psi \mid \psi\,\texttt{\&\&}\,\psi \mid \psi\,\texttt{||}\,\psi$$

where p is an action proposition. Let $\Phi(\mathcal{L}_\Sigma)$ be the set of action formulae defined over $\mathcal{L}_\Sigma$. The satisfaction relation ($\Vdash$) can be extended to action formulae in a standard way. That is, we write $a \Vdash \psi$ whenever action a satisfies action formula ψ. For example, $a \Vdash \psi_1 \,\texttt{\&\&}\, \psi_2$ whenever both $a \Vdash \psi_1$ and $a \Vdash \psi_2$. The other cases are analogous.

3. AN OBLIGATION POLICY LANGUAGE

By examining the structure of obligation policies found in the literature [30, 17, 23, 20, 11], we identified the four common components of obligation policies: (1) ***Trigger*** defines the applicability of an obligation policy. Obligations are imposed whenever the trigger condition is satisfied. (2) ***Obligation*** is the core of an obligation policy. It specifies the actions that are obliged (or prohibited) to be performed by the target program. (3) ***Temporal constraint*** defines the time window in which the obligation is imposed. (4) ***Penalty (Reward)*** is the access control implication if the target program violates (or fulfills) the obligation. Except for the obligation component, every component is optional. Consequently, rather than designing a monolithic syntax for a policy clause, we instead design a reusable construct for each component, and use a temporal logic to provide means to mix and match the components.

Our policy language is a temporal logic, in which a formula is evaluated against a trace. Consequently, formulae in the policy language are called ***trace formulae*** (ϕ) to distinguish them from action formulae (ψ). We capture the standard components of obligation policies by various temporal operators (or their derived forms) in the language. This allows the policy developer to compose expressive obligation policies in a flexible manner.

3.1 Syntax and Semantics

Syntax.

The syntax of trace formulae (ϕ) is given below.

$$\phi ::= \top \mid \psi \mid \neg\phi \mid \phi \wedge \phi \mid \mathsf{Eventually}\ \phi \mid \mathsf{Before}^+\ \phi : \phi \mid \mathsf{After}^+\ \phi : \phi \mid \mathsf{Ignoring}\ \psi : \phi$$

Semantics.

Given a trace $w \in \Sigma^\infty$ and a trace formula ϕ, we write $w \models \phi$ to assert that w ***satisfies*** ϕ:

1. $w \models \top$. (That is, $\top$ is satisfied by any trace.)
2. $w \models \psi$ iff there exists an action $a \preceq w$ such that $a \Vdash \psi$. (Intuitively, the *trace* formula ψ is satisfied by a non-empty trace with its first action satisfying the *action* formula ψ. Note that ψ specifies a guarantee property.)
3. $w \models \neg\phi$ iff $w \not\models \phi$.
4. $w \models \phi_1 \wedge \phi_2$ iff both $w \models \phi_1$ and $w \models \phi_2$.
5. $w \models \mathsf{Eventually}\ \phi$ iff there exists suffix $v \sqsubseteq w$ such that $v \models \phi$. (That is, $\mathsf{Eventually}\ \phi$ holds whenever ϕ holds for some suffix of w. $\mathsf{Eventually}$ is therefore semantically equivalent to the temporal operator $\mathcal{F}$ in LTL [33].)
6. $w \models \mathsf{Before}^+\ \phi_1 : \phi_2$ iff either (a) there exists prefix $u \preceq w$ such that (i) $u \models \phi_1$, (ii) for every proper prefix $u' \prec u$, $u' \not\models \phi_1$, and (iii) $u \models \phi_2$, or (b) for every prefix $u \preceq w$, $u \not\models \phi_1$, and $w \models \phi_2$. (The meaning of $\mathsf{Before}^+\ \phi_1 : \phi_2$ is illustrated in Fig. 2. Suppose t is the first instant when ϕ_1 is fulfilled by a prefix $a_1 a_2 \ldots a_t$ of w. Then ϕ_2 is imposed on that prefix. If ϕ_1 is never fulfilled by any prefix of w, then ϕ_2 is imposed over w itself. The intended usage is that ϕ_1 shall be a guarantee property and ϕ_2 shall be a safety property. When used this way, $\mathsf{Before}^+\ \phi_1 : \phi_2$ is a safety property.)
7. $w \models \mathsf{After}^+\ \phi_1 : \phi_2$ iff either (a) there exists prefix $u \preceq w$ such that (i) $u \models \phi_1$, (ii) for every proper prefix $u' \prec u$, $u' \not\models \phi_1$, and (iii) $w - u \models \phi_2$, or (b) for every prefix $u \preceq w$, $u \not\models \phi_1$. (Fig. 2 illustrates

[1]Note that the definition of safety and guarantee properties in formal verification literature concerns only infinite traces [2, 10]. Our definition of safety property for both finite and infinite traces closely follows that of [27]. Our adaptation of guarantee properties to account for both finite and infinite traces is obtained in a similar fashion.

[2]In formal verification literature [2, 10], in which complete program executions are modelled as infinite traces, there is no difference between a k-bounded safety property and a k-bounded guarantee property. Yet, in our model, in which partial executions and terminated executions are modelled as finite traces, the two notions refer to two distinct families of properties.

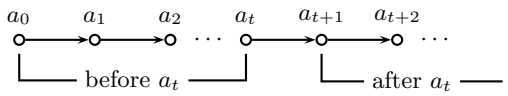

Figure 2: Illustration of "before" and "after" a certain time instant t.

the meaning of $\mathsf{After}^+\ \phi_1 : \phi_2$. Suppose t is the first instant when ϕ_1 is fulfilled by a prefix $a_1 a_2 \ldots a_t$ of w. Then ϕ_2 is imposed on the suffix $a_{t+1}a_{t+2}\ldots$. If ϕ_1 is never fulfilled by any prefix of w, then there is no further obligations. The intended usage is that ϕ_1 shall be a guarantee property and ϕ_2 shall be a safety property. When used this way, $\mathsf{After}^+\ \phi_1 : \phi_2$ is a safety property.)

8. $w \models \mathsf{Ignoring}\ \psi : \phi_2$ iff $w|_\psi \models \phi_2$, where $w|_\psi$ is the subsequence of w obtained by removing from w all actions a for which $a \Vdash \psi$.

Derived forms.

The following derived forms are defined to capture further idiomatic elements in obligation policies.

$$\bot \stackrel{\text{def}}{=} \neg\top \qquad \phi_1 \vee \phi_2 \stackrel{\text{def}}{=} \neg(\neg\phi_1 \wedge \neg\phi_2)$$

$$\mathsf{Always}\ \phi \stackrel{\text{def}}{=} \neg\mathsf{Eventually}\ \neg\phi$$

$$\mathsf{Before}^-\ \phi_1 : \phi_2 \stackrel{\text{def}}{=} \neg\mathsf{Before}^+\ \phi_1 : \neg\phi_2$$

$$\mathsf{After}^-\ \phi_1 : \phi_2 \stackrel{\text{def}}{=} \neg\mathsf{After}^+\ \phi_1 : \neg\phi_2$$

$$\langle \mathtt{1} \rangle \stackrel{\text{def}}{=} \mathtt{true} \qquad \langle \mathtt{k} \rangle \stackrel{\text{def}}{=} \mathsf{After}^-\ \langle \mathtt{1} \rangle : \langle \mathtt{k}-\mathtt{1} \rangle$$

$$[\psi] \stackrel{\text{def}}{=} \neg(!\ \psi)$$

$$\mathsf{Whenever}\ \phi_1 : \phi_2 \stackrel{\text{def}}{=} \mathsf{Always}\ \mathsf{After}^+\ \phi_1 : \phi_2$$

$$\mathsf{Fulfilling}^k\ \phi_1\,?\,\phi_2 : \phi_3 \stackrel{\text{def}}{=}$$
$$(\mathsf{After}^+\ (\mathsf{Before}^-\ \langle \mathtt{k} \rangle : \phi_1) : \phi_2) \wedge$$
$$((\mathsf{Before}^+\ \phi_1 : \neg\langle \mathtt{k}+\mathtt{1} \rangle) \vee\ \mathsf{After}^+\ \langle \mathtt{k} \rangle : \phi_3)$$

Some of the derived forms are merely standard duals ($\bot$, $\vee$, Always). $\mathsf{Before}^-\ \phi_1 : \phi_2$ is intended to be a guarantee property, imposing both guarantee properties ϕ_1 and ϕ_2, and requiring ϕ_2 to be discharged no later than ϕ_1. $\mathsf{After}^-\ \phi_1 : \phi_2$ is intended to be a guarantee property, imposing guarantee property ϕ_2 after guarantee property ϕ_1 is discharged. (The superscripts + and − indicate the intended usage of the four Before and After constructs: the + versions are intended to form safety properties, and the − versions are intended to form guarantee properties.) $\langle \mathtt{1} \rangle$ is a guarantee property fulfilled by a trace of length one or more; $\langle \mathtt{k} \rangle$ extends the lower bound to k. $[\psi]$ is a safety property violated by a non-empty trace for which the first action violates the action formula ψ. ($[\psi]$ is therefore the safety counterpart of the guarantee formula ψ.) $\mathsf{Whenever}\ \phi_1 : \phi_2$ imposes ϕ_2 whenever ϕ_1 is triggered. $\mathsf{Fulfilling}^k\ \phi_1\,?\,\phi_2 : \phi_3$ asserts the following: if a guarantee property ϕ_1 is discharged within k steps, then impose ϕ_2 after fulfillment, otherwise impose ϕ_3 after k steps.

Convention.

To reduce unnecessary brackets, the following binding priorities are adopted: unary operators ($\neg$, Eventually, Always) bind most tightly; next in order comes binary boolean operators ($\wedge$, $\vee$); then comes binary temporal operators (Before, After, Whenever, Ignoring).

Expressing Obligation Policies.

A generic template for a typical obligation policy can be encoded in our policy language as follows:

$$\mathsf{Whenever}\ \phi_{trig} : \mathsf{Fulfilling}^k\ (\mathsf{Before}^-\ \phi_{tc}\ :\ \phi_{obl})\,?\,\phi_r : \phi_p$$

The clause ϕ_{trig}, ϕ_{tc}, ϕ_{obl}, ϕ_r and ϕ_p correspond to the trigger, the temporal constraint, the obligation, the reward and penalty component of an obligation policy respectively. The formulae ϕ_{trig} and ϕ_{obl} are intended to be guarantee properties. The formula ϕ_{tc} is intended to specify a guarantee property that is fulfilled when the deadline of the obligation is reached. The formulae ϕ_r and ϕ_p are intended to be safety properties, with ϕ_r granting a more liberal level of access than ϕ_r.

3.2 Case Studies: Obligation Policies in Software Systems

We illustrate the use of our language by the following examples.

Policy 1 (File check-out and return [12]). *In a code versioning tool, a developer who checks out a file is obliged to check the file back in.*

This policy can be expressed as Formula (1) in Fig. 3. The action propositions p_{cout} and p_{cin} are satisfied respectively by file-check-out and file-check-in operations respectively. Formula (1) demands that, once the file is checked out (p_{cout}), no other check-out (p_{cout}) is allowed before the file is checked-in (p_{cin}) again.

Policy 2 (Log after access denial [12]). *When a user attempts to access a document for which he lacks the required credentials, every subsequent attempts to access documents by that user must be logged.*

This policy can be captured by Formula (2) in Fig. 3. The action propositions p_{unauth_acc}, p_{acc} and p_{log} are satisfied respectively by actions that access documents without credentials, actions that access documents (with or without credentials), and actions that log the subsequent document-access attempt of the user. Formula (2) states that: after the first unauthorized access attempt to a document (p_{unauth_acc}), any subsequent document-access operations (p_{acc}) are disallowed unless a log operation (p_{log}) has been performed, and meanwhile, there is no two consecutive document-access operations (p_{acc}) without a log operation (p_{log}) between them.

Policy 3 (Secured connect after read [1]). *An application can only open a file for reading. Once the file has been accessed, the application has to obtain approval from the user each time a connection is to be opened.*

Formula (3) in Fig. 3 encodes this policy. The action propositions p_{write}, p_{read}, p_{acq_appr} and p_{conn} are satisfied respectively by program actions that access a file for write, actions that access a file for read, actions that acquire user approval

$$\text{Whenever Eventually } p_{cout} \text{ : (Eventually } p_{cin} \wedge \text{ Before}^{+} \text{ Eventually } p_{cin} \text{ : Always } \neg p_{cout}) \quad (1)$$

$$\text{After}^{+} \text{ Eventually } p_{unauth\_acc} \text{ : ((Before}^{+} \text{ Eventually } p_{log} \text{ : Always } \neg p_{acc}) \wedge$$
$$\text{Whenever Eventually } p_{acc} \text{ : Before}^{+} \text{ Eventually } p_{log} \text{ : Always } \neg p_{acc}) \quad (2)$$

$$\text{Always } \neg p_{write} \wedge \text{ After}^{+} \text{ Eventually } p_{read} \text{ : (Before}^{+} \text{ Eventually } p_{acq\_appr} \text{ : Always } \neg p_{conn}) \wedge$$
$$\text{(Whenever Eventually } p_{conn} \text{ : Before}^{+} \text{ Eventually } p_{acq\_appr} \text{ : Always } \neg p_{conn}) \quad (3)$$

$$\text{(Whenever Eventually } p_{unsec\_conn} \text{ : Before}^{+} \text{ Eventually } p_{close\_conn} \text{ : Always } \neg p_{acc\_pim}) \wedge$$
$$\text{(After}^{+} \text{ Eventually } p_{acc\_pim} \text{ : Always } \neg p_{unsec\_conn}) \quad (4)$$

$$\text{Before}^{+} \text{ Eventually } p_{appr} \text{ : Always } \neg p_{critical\_acc} \quad (5)$$

$$\text{Whenever Eventually } p_{critical\_acc} \text{ : Before}^{+} \text{ Eventually } p_{appr} \text{ : Always } \neg p_{critical\_acc} \quad (6)$$

$$\text{Before}^{+} \text{ Eventually } p_{acc} \text{ : Always } \neg p_{appr} \quad (7)$$

$$\text{Whenever Eventually } p_{acc} \text{ :}$$
$$\text{Fulfilling}^{1} p_{log} \text{ ? (After}^{+} \text{ Eventually } p_{appr} \text{ : Before}^{+} \text{ Eventually } p_{critical\_acc} \text{ : Always } \neg p_{appr}) \text{ : (Always } \neg p_{appr}) \quad (8)$$

Figure 3: Examples of policy formulas.

to connect to Internet, and actions that make Internet connection. Thus, Formula (3) states that: (a) writing to a file (p_{write}) is not allowed; (b) after a file has been read (p_{read}), the application is not allowed to connect to Internet (p_{conn}) before it has acquired approval from the user (p_{acq_appr}), and meanwhile, there is no two consecutive connect-to-Internet operations (p_{conn}) without an acquire-user-approval operation (p_{acq_appr}) in between.

POLICY 4 (SECURED PIM ACCESS [12]). *An application must not access the Personal Information Manager (PIM) while unsecured connections are open, and it must only open secure connections after the PIM is accessed.*

Formula (4) in Fig. 3 captures this policy. The action propositions p_{unsec_conn}, p_{close_conn} and p_{acc_pim} are satisfied respectively by program actions that make a HTTP connection[3], actions that close the opened HTTP connection, and actions that access the PIM respectively. Formula (4) states that: (a) once an HTTP connection has been opened (p_{unsec_conn}) , then before the connection is closed (p_{close_conn}), the application is not permitted to access the PIM (p_{acc_pim}); (b) after the PIM has been accessed (p_{acc_pim}), HTTP connections are not permitted (p_{unsec_conn}).

POLICY 5 (ACCESS TO CRITICAL DOCUMENTS). *An application must explicitly request permission from the user before every access to a critical document. In addition, such a request is not even allowed until the application can demonstrate its trustworthiness by dutifully logging every document access (including non-critical documents). If this logging obligation is at any time violated, then the application will forever loss its right to accessing critical documents.*

Critical documents are sensitive resources. Their accesses are not available to an application until the application can demonstrate that it fulfills a certain obligation. Here, the obligation is imposed in a continuous manner: the application has to honor the obligation repeatedly. Once the application violates the obligation, the application will be penalized with deprival of rights to access sensitive resources.

Suppose the action propositions p_{appr}, $p_{critical\_acc}$, p_{acc} and p_{log} are satisfied respectively by program actions for acquiring user approval to access critical documents, actions for accessing critical documents, actions for accessing either critical or non-critical documents, and actions for logging a document-access operation. The above policy can be specified as the conjunction of four formulas:

$$\phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4$$

The definition of the four conjuncts are given below.

1. Conjunct ϕ_1 is specified as Formula (5) in Fig. 3, which requires the following: without user approval (p_{appr}), access to critical documents ($p_{critical\_acc}$) is not permitted.

2. Conjunct ϕ_2 is specified as Formula (6) in Fig. 3, which requires the following: there is no two consecutive $p_{critical\_acc}$ operations without a p_{appr} operation in between.

3. Conjunct ϕ_3 is specified as Formula (7) in Fig. 3. Before the application can demonstrate that it fulfills the log-following-access obligation (i.e., it has performed a document-access operation (p_{acc}) at least once), it is not allowed to acquire user approval to access the critical documents (p_{appr}).

4. Conjunct ϕ_4 is specified as Formula (8) in Fig. 3. If the target application fulfills the obligation of logging (p_{log}) immediately after a generic document access operation (p_{acc}), then, as a ***reward***, the application unlocks the privilege to acquire user approval for accessing critical document (p_{appr}). Such a privilege is

[3]In contrast to the HTTPS connection, the HTTP connection are considered insecure.

granted in only a constrained way: i.e., when it acquires user approval to access critical documents (p_{appr}), it has to perform the critical-document-access operation ($p_{critical\_acc}$). If the aforementioned obligation is violated, the application is not allowed to acquire user approval to access critical documents (p_{appr}).

4. RECOGNIZING ENFORCEABLE AND MONITORABLE POLICIES

Type System.

Not all policies specified in our policy language are enforceable. This section introduces a type system that can be used for recognizing if a given obligation policy (written in our policy language) is enforceable. Specifically, Fig. 4 specifies the inference rules that define the typing assertion "$\phi : T$", where T is the type of ϕ. If T is of the form k-***enf***, where $k \in \mathbf{N} \cup \{\omega\}$, then ϕ is recognized to be a non-empty safety property: i.e., the empty trace satisfies the property, and thus the property is not empty. The implication is that ϕ is ***enforceable*** by a reference monitor. If $k \in \mathbb{N}$ then ϕ is a k-bounded safety property, meaning that the reference monitor can stop monitoring if no violation is detected in the first k steps. If T is of the form k-***mon***, where $k \in \mathbf{N} \cup \{\omega\}$, then ϕ is recognized to be a non-universal guarantee property: i.e., there is at least one trace, the empty trace, that does not satisfy the property. The implication is that the discharging of ϕ can be detected by a reference monitor. Such a property is said to be ***monitorable***. If $k \in \mathbb{N}$ then ϕ is a k-bounded guarantee property, meaning that the reference monitor can safely stop monitoring if the property is not discharged in the first k steps. The soundness of the type system is ensured by Theorem 12. In summary, the type system captures patterns of policy composition that yields policies that are enforceable or monitorable.

Incremental Refinement of Obligation Policies.

This type system can guide a policy developer to incrementally refine an obligation policy into one that is enforceable by a reference monitor. Specifically, the type system identifies components of the policy that are causing problem. The following case study demonstrates this use.

Suppose a policy developer is to write a library loan policy, which states that every book checked out from the library has to be returned. The goal is to formulate the policy in such a way that it is enforceable. Let the action propositions p_{cout} and p_{ret} refer to the check-out and return operations respectively. Suppose further that each time point corresponds to a day. We start with a naive formulation that captures the main intention of the policy.

POLICY 6. *Every book checked out from the library has to be retured.*

$$\mathsf{Whenever}\ \mathsf{Eventually}\ p_{cout}\ :\ \mathsf{Eventually}\ p_{ret}$$

Note that Whenever is merely a shorthand for Always After^{+}. By TE-AF, the formula is not well-typed (Fig. 5), meaning that the formula is not enforceable. TE-AF also identifies the subformula (Eventually p_{ret}) to be the cause (i.e., that subformula is monitorable but not enforceable).

One standard way to turn a monitorable policy into an enforceable policy is by the use of reward and penalty. Such a policy structure is captured in the Fulfilling construct, which, according to TE-FU, takes a monitorable policy as an operand, and produces an enforceable policy. So the policy developer refines the policy as follows.

POLICY 7. *Every book checked out from the library has to be retured, otherwise the book user will be penalized for not allowing to borrow any books from the library.*

$$\mathsf{Whenever}\ \mathsf{Eventually}\ p_{cout}\ :\ \mathsf{Fulfilling}^{k}\ \mathsf{Eventually}\ p_{ret}\ ?\ \top\ :\ \mathsf{Always}\ \neg p_{cout}$$

Note that the formula is still not well typed because TE-FU expects (Eventually p_{ret}) to be a k-bounded guarantee property for some finite k (Fig. 6). That means a deadline k must be imposed on book returning.

POLICY 8. *Every book checked out from the library has to be retured within 30 days, otherwise the book user will be penalized for not allowing to borrow any book.*

$$\mathsf{Whenever}\ \mathsf{Eventually}\ p_{cout}\ :\ \mathsf{Fulfilling}^{30}\ (\mathsf{Before}^{-}\ \langle 30\rangle : \mathsf{Eventually}\ p_{ret})\ ?\ \top\ :\ \mathsf{Always}\ \neg p_{cout}$$

The policy developer now sets a deadline for the Eventually clause, making the obligation a bounded guarantee property. The type system recognizes this formula to have type ω-***enf***, meaning that the formula is enforceable (Fig. 7).

Discussions.

A reasonable implementation of the type checker can *infer* both the bound in the Fulfilling construct and the signs ($+$ or $-$) of the Before and After constructs. So the policy developer can in theory omit these details.

5. POLICY COMPILATION

This section reports an algorithm for compiling obligation policies to their enforcement or monitoring mechanisms. The algorithm translates well-typed policies to their corresponding enforcement, and thus serves as a soundness proof for the type system.

We begin with a representation of enforcement and monitoring mechanisms called obligation monitors. This representation is the "assembly language" of our compilation.

5.1 Obligation Monitors

An ***Obligation Monitor (OM)*** can be used for representing either an enforcement mechanism for a safety property, or a monitoring mechanism for a guarantee property. Partly inspired by [41], it is designed to facilitate the inlining of monitoring logic, and is fully capable of expressing deontic concepts such as rights, prohibitions, obligations and dispensations [24].

DEFINITION 9. *An OM $\mathcal{M}$ is a quadruple of the form $\langle \mathcal{L}_\Sigma, \mathcal{O}, \iota, S_0 \rangle$.*

1. *$\mathcal{L}_\Sigma$ is an* **action assertion language** *over the set of actions Σ.*

2. *$\mathcal{O}$ is a countable set of* **obligation identifiers**.

$$\frac{}{\psi : 1\text{-}\boldsymbol{mon}} \text{ (TM-Ax)} \qquad \frac{}{[\psi] : 1\text{-}\boldsymbol{enf}} \text{ (TE-Ax)}$$

$$\frac{\phi : k\text{-}\boldsymbol{enf}}{\neg\phi : k\text{-}\boldsymbol{mon}} \text{ (TM-Ne)} \qquad \frac{\phi : k\text{-}\boldsymbol{mon}}{\neg\phi : k\text{-}\boldsymbol{enf}} \text{ (TE-Ne)}$$

$$\frac{\phi : k\text{-}\boldsymbol{mon}}{\mathsf{Eventually}\ \phi : \omega\text{-}\boldsymbol{mon}} \text{ (TM-Ev)} \qquad \frac{\phi : k\text{-}\boldsymbol{enf}}{\mathsf{Always}\ \phi : \omega\text{-}\boldsymbol{enf}} \text{ (TE-Al)}$$

$$\frac{\phi_1 : k_1\text{-}\boldsymbol{mon} \qquad \phi_2 : k_2\text{-}\boldsymbol{mon} \qquad k = \boldsymbol{max}\,(k_1,\ k_2)}{\phi_1 \vee \phi_2 : k\text{-}\boldsymbol{mon}} \text{ (TM-Or)} \qquad \frac{\phi_1 : k_1\text{-}\boldsymbol{enf} \qquad \phi_2 : k_2\text{-}\boldsymbol{enf} \qquad k = \boldsymbol{max}\,(k_1,\ k_2)}{\phi_1 \vee \phi_2 : k\text{-}\boldsymbol{enf}} \text{ (TE-Or)}$$

$$\frac{\phi_1 : k_1\text{-}\boldsymbol{mon} \qquad \phi_2 : k_2\text{-}\boldsymbol{mon} \qquad k = \boldsymbol{max}\,(k_1,\ k_2)}{\phi_1 \wedge \phi_2 : k\text{-}\boldsymbol{mon}} \text{ (TM-An)} \qquad \frac{\phi_1 : k_1\text{-}\boldsymbol{enf} \qquad \phi_2 : k_2\text{-}\boldsymbol{enf} \qquad k = \boldsymbol{max}\,(k_1,\ k_2)}{\phi_1 \wedge \phi_2 : k\text{-}\boldsymbol{enf}} \text{ (TE-An)}$$

$$\frac{\phi_1 : k_1\text{-}\boldsymbol{mon} \qquad \phi_2 : k_2\text{-}\boldsymbol{mon}}{\mathsf{Before}^{-}\ \phi_1 : \phi_2 : k_2\text{-}\boldsymbol{mon}} \text{ (TM-Be)} \qquad \frac{\phi_1 : k_1\text{-}\boldsymbol{mon} \qquad \phi_2 : k_2\text{-}\boldsymbol{enf}}{\mathsf{Before}^{+}\ \phi_1 : \phi_2 : k_2\text{-}\boldsymbol{enf}} \text{ (TE-Be)}$$

$$\frac{\phi_1 : k_1\text{-}\boldsymbol{mon} \qquad \phi_2 : k_2\text{-}\boldsymbol{mon} \qquad k = k_1 + k_2}{\mathsf{After}^{-}\ \phi_1 : \phi_2 : k\text{-}\boldsymbol{mon}} \text{ (TM-Af)} \qquad \frac{\phi_1 : k_1\text{-}\boldsymbol{mon} \qquad \phi_2 : k_2\text{-}\boldsymbol{enf} \qquad k = k_1 + k_2}{\mathsf{After}^{+}\ \phi_1 : \phi_2 : k\text{-}\boldsymbol{enf}} \text{ (TE-Af)}$$

$$\frac{\phi : k\text{-}\boldsymbol{mon}}{\mathsf{Ignoring}\ \psi : \phi : \omega\text{-}\boldsymbol{mon}} \text{ (TM-Ig)} \qquad \frac{\phi : k\text{-}\boldsymbol{enf}}{\mathsf{Ignoring}\ \psi : \phi : \omega\text{-}\boldsymbol{enf}} \text{ (TE-Ig)}$$

$$\frac{k \in \mathbf{N}}{\langle \mathbf{k} \rangle : k\text{-}\boldsymbol{mon}} \text{ (TM-Co)} \qquad \frac{\phi_1 : k_1\text{-}\boldsymbol{mon} \qquad k_1 \in \mathbf{N} \qquad \phi_2 : k_2\text{-}\boldsymbol{enf} \qquad \phi_3 : k_3\text{-}\boldsymbol{enf} \qquad k = k_1 + \boldsymbol{max}\,(k_2,\ k_3)}{\mathsf{Fulfilling}^{k_1}\ \phi_1\ ?\ \phi_2 : \phi_3 : k\text{-}\boldsymbol{enf}} \text{ (TE-Fu)}$$

Figure 4: Inference rules for the type system

$$\dfrac{\dfrac{\dfrac{p_{cout} : 1\text{-}\boldsymbol{mon}}{\mathsf{Eventually}\ p_{cout} : \omega\text{-}\boldsymbol{mon}}\ \text{(TM-Ev)} \qquad \dfrac{p_{ret} : 1\text{-}\boldsymbol{mon}}{\mathsf{Eventually}\ p_{ret} : \omega\text{-}\boldsymbol{mon}}\ \text{(TM-Ev)}}{\mathsf{After}^{+}\ \mathsf{Eventually}\ p_{cout} : \mathsf{Eventually}\ p_{ret}}\ \text{(Err:TE-Af)}}{\mathsf{Always\ After}^{+}\ \mathsf{Eventually}\ p_{cout} : \mathsf{Eventually}\ p_{ret}}\ \text{(TE-Al)}$$

Figure 5: Derivation tree for Policy 6. (Err:TE-Af) indicates that an error occurs when applying rule TE-Af.

$$\dfrac{\dfrac{\dfrac{p_{cout} : 1\text{-}\boldsymbol{mon}}{\mathsf{Eventually}\ p_{cout} : \omega\text{-}\boldsymbol{mon}}\ \text{(TM-Ev)} \qquad \dfrac{\dfrac{p_{ret} : 1\text{-}\boldsymbol{mon}}{\mathsf{Eventually}\ p_{ret} : \omega\text{-}\boldsymbol{mon}}\ \text{(TM-Ev)} \qquad k_1 = \omega \qquad \dfrac{\dfrac{p_{ret} : 1\text{-}\boldsymbol{mon}}{\neg p_{ret} : 1\text{-}\boldsymbol{enf}}\ \text{(TE-No)}}{\mathsf{Always}\ \neg p_{cout} : \omega\text{-}\boldsymbol{enf}}\ \text{(TE-Al)}}{\mathsf{Fulfilling}^{\omega}\ \mathsf{Eventually}\ p_{ret}\ ?\ \top : \mathsf{Always}\ \neg p_{cout}}\ \text{(Err: TE-Fu)}}{\mathsf{After}^{+}\ \mathsf{Eventually}\ p_{cout} : \mathsf{Fulfilling}^{\omega}\ \mathsf{Eventually}\ p_{ret}\ ?\ \top : \mathsf{Always}\ \neg p_{cout}}\ \text{(TE-Af)}}{\mathsf{Always\ After}^{+}\ \mathsf{Eventually}\ p_{cout} : \mathsf{Fulfilling}^{\omega}\ \mathsf{Eventually}\ p_{ret}\ ?\ \top : \mathsf{Always}\ \neg p_{cout}}\ \text{(TE-Al)}$$

Figure 6: Derivation tree for Policy 7. (Err:TE-Fu) indicates that an error occurs when applying rule TE-Fu.

$$\dfrac{\dfrac{\dfrac{\dfrac{p_{cout} : 1\text{-}\boldsymbol{mon}}{\mathsf{Eventually}\ p_{cout} : 1\text{-}\boldsymbol{mon}}\ \text{(TM-Ev)} \qquad \dfrac{\dfrac{30 \in \mathbb{N}}{\langle 30 \rangle : 30\text{-}\boldsymbol{mon}}\ \text{(TM-Co)} \qquad \dfrac{p_{ret} : 1\text{-}\boldsymbol{mon}}{\mathsf{Eventually}\ p_{ret} : \omega\text{-}\boldsymbol{mon}}\ \text{(TM-Ev)}}{\mathsf{Before}^{-}\ \langle 30 \rangle : \mathsf{Eventually}\ p_{ret} : 30\text{-}\boldsymbol{mon}}\ \text{(TM-Be)} \qquad k_1 = 30 \qquad \dfrac{\dfrac{p_{ret} : 1\text{-}\boldsymbol{mon}}{\neg p_{ret} : 1\text{-}\boldsymbol{enf}}\ \text{(TE-No)}}{\mathsf{Always}\ \neg p_{cout} : \omega\text{-}\boldsymbol{enf}}\ \text{(TE-Al)}}{(\mathsf{Fulfilling}^{30}\ \mathsf{Before}^{-}\ \langle 30 \rangle : \mathsf{Eventually}\ p_{ret}\ ?\ \top : \mathsf{Always}\ \neg p_{cout}) : 30\text{-}\boldsymbol{enf}}\ \text{(TE-Fu)}}{(\mathsf{After}^{+}\ \mathsf{Eventually}\ p_{cout} : \mathsf{Fulfilling}^{30}\ \mathsf{Before}^{-}\ \langle 30 \rangle : \mathsf{Eventually}\ p_{ret}\ ?\ \top : \mathsf{Always}\ \neg p_{cout}) : \omega\text{-}\boldsymbol{enf}}\ \text{(TE-Af)}}{(\mathsf{Always\ After}^{+}\ \mathsf{Eventually}\ p_{cout} : \mathsf{Fulfilling}^{30}\ \mathsf{Before}^{-}\ \langle 30 \rangle : \mathsf{Eventually}\ p_{ret}\ ?\ \top : \mathsf{Always}\ \neg p_{cout}) : \omega\text{-}\boldsymbol{enf}}\ \text{(TE-Al)}$$

Figure 7: Derivation tree for Policy 8

3. $\iota : \mathcal{O} \to \mathsf{obl}_{\mathcal{M}}$ *assigns to each obligation identifier an* **obligation** *from* $\mathsf{obl}_{\mathcal{M}}$*, where* $\mathsf{obl}_{\mathcal{M}} = \Phi(\mathcal{L}_\Sigma) \uplus (\Phi(\mathcal{L}_\Sigma) \times [\mathcal{O}]^{<\omega} \times 2^{\mathcal{O}})$.

4. $S_0 \in \mathsf{states}_{\mathcal{M}}$ *is the* **initial state**.

An OM is a state machine. The set $\mathsf{states}_{\mathcal{M}}$ of states is defined as $[\mathcal{O}]^{<\omega}$: i.e., a state is a finite set of obligation identifiers. The initial state S_0 is one of such states.

The obligation identifiers in a given state encode the transitions allowed for that state. Specifically, the function ι interprets an obligation identifier as an obligation. There are two kinds of obligations. First, an obligation of the form $\psi \in \Phi(\mathcal{L}_\Sigma)$ is called a ***simple condition***. Intuitively, an action $a \in \Sigma$ is allowed to be performed at a state S only if a satisfies every simple condition identified in S. Second, an obligation of the form $\langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \rangle \in \Phi(\mathcal{L}_\Sigma) \times [\mathcal{O}]^{<\omega} \times 2^{\mathcal{O}}$ is called a ***trigger rule***, where:

- ψ is the ***trigger condition***, specifying the actions that would render the trigger rule applicable to the monitor state,
- $\mathcal{O}_{add}$ identifies a finite set of obligation identifiers to be added to the monitor state when this trigger rule is applied, and,
- $\mathcal{O}_{del}$ identifies a set of obligation identifiers to be deleted from the monitor state when this trigger rule is applied.

Intuitively, if an action $a \in \Sigma$ is allowed to be performed at a monitor state S, then the trigger rules identified in S will be tested for applicability, and those applicable trigger rules will specify how S will be changed. We formalize state transitions in the following.

Transition.

Given a monitor state $S \in \mathsf{states}_{\mathcal{M}}$, we define $\mathsf{cond}_{\mathcal{M}}(S) = \{o \in S \mid \iota(o) \in \Phi(\mathcal{L}_\Sigma)\}$ as the set of obligation identifiers in S that refer to simple conditions. Similarly, $\mathsf{rule}_{\mathcal{M}}(S) = \{o \in S \mid \iota(o) \in \Phi(\mathcal{L}_\Sigma) \times [\mathcal{O}]^{<\omega} \times 2^{\mathcal{O}}\}$ is the set of obligation identifiers in S that refer to trigger rules. We define $\mathsf{trig}_{\mathcal{M}}(S, a) = \{o \in \mathsf{rule}_{\mathcal{M}}(S) \mid \iota(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \rangle \wedge (a \Vdash \psi)\}$ to be the set of obligation identifiers for those trigger rules that are "triggered" by a at state S (i.e., the trigger condition is satisfied). Moreover, we define $\mathsf{add}_{\mathcal{M}}(S, a)$ and $\mathsf{del}_{\mathcal{M}}(S, a)$ as the sets of obligation identifiers that will be added and deleted after applying all the rules that are fired by a at the monitor state S. Formally,

$$\mathsf{add}_{\mathcal{M}}(S, a) \stackrel{\text{def}}{=} \bigcup \{\, \mathcal{O}_{add} \mid o \in \mathsf{trig}_{\mathcal{M}}(S, a), \iota(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \rangle \,\}$$

$$\mathsf{del}_{\mathcal{M}}(S, a) \stackrel{\text{def}}{=} \bigcup \{\, \mathcal{O}_{del} \mid o \in \mathsf{trig}_{\mathcal{M}}(S, a), \iota(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \rangle \,\}$$

Given an action $a \in \Sigma$, a transition $S_1 \xrightarrow{a}_{\mathcal{M}} S_2$ can be made iff both of the following hold: (1) for every $o \in \mathsf{cond}_{\mathcal{M}}(S_1)$, we have $a \Vdash \iota(o)$; (2) $S_2 = (S_1 \cup \mathsf{add}_{\mathcal{M}}(S_1, a)) \backslash \mathsf{del}_{\mathcal{M}}(S_1, a)$. One can extend the one-step transition relation to a multi-step transition relation in a standard way: we write $S \xrightarrow{\epsilon}_{\mathcal{M}} S$, and write $S_1 \xrightarrow{a \cdot w}_{\mathcal{M}} S_2$ whenever $S_1 \xrightarrow{a}_{\mathcal{M}} S_3$ and $S_3 \xrightarrow{w}_{\mathcal{M}} S_2$ for some $S_3 \in \mathsf{states}_{\mathcal{M}}$.

From OMs to Properties.

We say that the OM $\mathcal{M}$ ***accepts*** the sequence $w \in \Sigma^\infty$ iff for any $u \preceq w$, $S_0 \xrightarrow{u}_{\mathcal{M}} S$ for some $S \in \mathsf{states}_{\mathcal{M}}$; and it ***recognizes*** the sequence $w \in \Sigma^\infty$ iff there exists $u \preceq w$, $S_0 \xrightarrow{u}_{\mathcal{M}} \emptyset$. Let $\mathcal{L}_{acc}(\mathcal{M})$ be the set of action sequences accepted by $\mathcal{M}$. An OM $\mathcal{M}$ ***enforces*** a property $\mathcal{L}$ iff $\mathcal{L} = \mathcal{L}_{acc}(\mathcal{M})$. Let $\mathcal{L}_{rec}(\mathcal{M})$ be the set of action sequences recognized by $\mathcal{M}$. An OM $\mathcal{M}$ ***monitors*** a property $\mathcal{L}$ iff $\mathcal{L} = \mathcal{L}_{rec}(\mathcal{M})$. Note that $\mathcal{L}_{acc}(\mathcal{M})$ is a non-empty safety property (non-empty because $\epsilon \in \mathcal{L}_{acc}(\mathcal{M})$), and $\mathcal{L}_{rec}(\mathcal{M})$ is a guarantee property.

Special OMs for Guarantee Properties.

An OM that contains only trigger rules (but no simple conditions) is called a ***pure obligation monitor***. It turns out that simple conditions are not needed for monitoring guarantee properties.

PROPOSITION 10. *If a guarantee property $\mathcal{L}$ is monitored by an OM, then $\mathcal{L}$ is also monitored by a pure OM.*

Our compilation algorithm will thus generate only pure OMs when the policy formula represents a guarantee property.

The pure OMs generated by our compilation algorithm have further structures.

PROPOSITION 11. *Suppose $\mathcal{L}$ is a non-universal guarantee property (i.e., non-universal meaning $\epsilon \notin \mathcal{L}$ and thus $\mathcal{L} \neq \Sigma^\infty$). If $\mathcal{L}$ is monitored by a pure OM, then $\mathcal{L}$ is also monitored by a pure OM $\mathcal{M} = \langle \mathcal{L}_\Sigma, \mathcal{O}, \iota, S_0 \rangle$ for which there is a set $\mathcal{O}_f \subseteq \mathcal{O}$ of trigger rule identifiers such that:*

$$\forall w \in \Sigma^*.\, \forall S_1 \in \mathsf{states}_{\mathcal{M}}.\, \Big\{ (S_0 \xrightarrow{w}_{\mathcal{M}} S_1 \wedge S_1 \neq \emptyset) \Rightarrow \forall a \in \Sigma.\, \forall S_2 \in \mathsf{states}_{\mathcal{M}}.\, \Big[(S_1 \xrightarrow{a}_{\mathcal{M}} S_2 \wedge S_2 = \emptyset) \Leftrightarrow (\mathsf{trig}_{\mathcal{M}}(S_1, a) \cap \mathcal{O}_f \neq \emptyset) \Big] \Big\}$$

$\mathcal{O}_f$ is the set of identifiers for the **final trigger rules** *of $\mathcal{M}$.*

Final trigger rules have a role analogous to "final states". They are triggered *if and only if* the next state is an empty state (i.e., the guarantee property has been fulfilled). The proposition above says that every guarantee property (except for Σ^∞) can be monitored by a pure OM with a set of final trigger rules. Our compilation algorithm compiles a guarantee property into a pure OM with final trigger rules.

5.2 Compilation Algorithm

We devised a compilation algorithm, `compile`(ϕ), that takes obligation policy ϕ as argument, and returns a tuple $\langle \mathcal{T}, \mathcal{B}, \mathcal{M}, \mathcal{O}_f \rangle$, where:

- $\mathcal{T} \in \{\boldsymbol{enf}, \boldsymbol{mon}\}$ is the type of ϕ assigned by the type system;
- $\mathcal{B} \in \mathbb{N} \cup \{\omega\}$ is the bound of ϕ assigned by the type system;
- $\mathcal{M} = \langle \mathcal{L}_\Sigma, \mathcal{O}, \iota, S_0 \rangle$ is an OM, which enforces ϕ if $\mathcal{T} = \boldsymbol{enf}$, or monitors ϕ if $\mathcal{T} = \boldsymbol{mon}$; in the latter case, $\mathcal{M}$ is a pure OM;
- $\mathcal{O}_f \subseteq \mathcal{O}$ is the set of final trigger rules for $\mathcal{M}$ if $\mathcal{T} = \boldsymbol{mon}$. $\mathcal{O}_f = \emptyset$ if $\mathcal{T} = \boldsymbol{enf}$.

The complete specification of the compilation algorithm is given in the companion technical report [40]. Here, we present the compilation of three example cases.

Case ψ.

A monitor for trace formula ψ examines only the first action of the input trace. If the first action satisfies ψ, then the monitor transitions to an empty state. Otherwise it moves to a non-empty state and stays there forever. To generate this monitor, we define $\texttt{compile}(\psi) = \langle \boldsymbol{mon}, 1, \mathcal{M}, \{o_1\}\rangle$, where $\mathcal{M} = \langle \mathcal{L}_\Sigma, \{o_1, o_2, o_r\}, \iota, \{o_1, o_2\}\rangle$ such that:

$$\begin{aligned}\iota(o_1) &= \langle \psi, \emptyset, \{o_1, o_2\}\rangle\\ \iota(o_2) &= \langle !\,\psi, \{o_r\}, \{o_1, o_2\}\rangle\\ \iota(o_r) &= \langle \texttt{true}, \emptyset, \emptyset\rangle\end{aligned}$$

At the initial state, if $a \Vdash \psi$ for the input action a, then o_1 is fired. This removes both o_1 and o_2 from the state, and puts $\mathcal{M}$ into an empty state. Otherwise, $a \not\Vdash \psi$, and thus o_2 is fired. This removes o_1 and o_2 from the state, while adding back o_r. As a result, $\mathcal{M}$ is trapped at a non-empty state.

Case $\mathsf{Before}^+\ \phi' : \phi''$.

The compilation of formula $\mathsf{Before}^+\ \phi' : \phi''$ depends on the recursive compilation of its subformulae ϕ' and ϕ''. Suppose $\texttt{compile}(\phi') = \langle \boldsymbol{mon}, k', \mathcal{M}', \mathcal{O}'_f\rangle$ and $\texttt{compile}(\phi'') = \langle \boldsymbol{enf}, k'', \mathcal{M}'', \emptyset\rangle$ where $\mathcal{M}' = \langle \mathcal{L}_\Sigma, \mathcal{O}', \iota', S'_0\rangle$ and $\mathcal{M}'' = \langle \mathcal{L}_\Sigma, \mathcal{O}'', \iota'', S''_0\rangle$.

Semantically, $\mathsf{Before}^+\ \phi' : \phi''$ enforces ϕ'' before ϕ' is fulfilled. So the intuitive compilation involves the running of $\mathcal{M}'$ and $\mathcal{M}''$ in parallel. As soon as $\mathcal{M}'$ detects the fulfillment of ϕ', $\mathcal{M}'$ terminates $\mathcal{M}''$ by "erasing" the state information of the latter. We define $\texttt{compile}(\mathsf{Before}^+\ \phi' : \phi'') = \langle \boldsymbol{enf}, k'', \mathcal{M}, \emptyset\rangle$ where $\mathcal{M} = \langle \mathcal{L}_\Sigma, \mathcal{O}' \uplus \mathcal{O}'', \iota, S'_0 \uplus S''_0\rangle$ such that:

$$\iota(o) = \begin{cases} \iota'(o) & \text{if } o \in \mathcal{O}'\backslash\mathcal{O}'_f\\ \iota''(o) & \text{if } o \in \mathcal{O}''\\ \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del} \uplus \mathcal{O}''\rangle & \text{if } o \in \mathcal{O}'_f \wedge\\ & \quad \iota'(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del}\rangle\end{cases}$$

First, $\mathcal{M}$ starts at initial state $S'_0 \uplus S''_0$ (the initial states of $\mathcal{M}'$ and $\mathcal{M}''$), so that the simulation of both $\mathcal{M}'$ and $\mathcal{M}''$ begins simultaneously. Suppose a trace fulfills ϕ', but does not violate ϕ'' up to the point of fulfillment of ϕ'. The trace will fires some trigger rule $o' \in \mathcal{O}_f$, which removes $\mathcal{O}''$ from the state, and transitions $\mathcal{M}'$ to an empty state. As a result, $\mathcal{M}''$ stops enforcing ϕ''.

Case $\mathsf{Eventually}\ \phi'$.

Suppose $\texttt{compile}(\phi') = \langle \boldsymbol{mon}, k', \mathcal{M}', \mathcal{O}'_f\rangle$ is the compilation for ϕ', where k' is a finite bound, and $\mathcal{M}' = \langle \mathcal{L}_\Sigma, \mathcal{O}', \iota', S'_0\rangle$.

Semantically, $\mathsf{Eventually}\ \phi'$ looks for the fulfillment of ϕ' in every suffix the target trace. If any of the suffice satisfies ϕ', then $\mathsf{Eventually}\ \phi'$ is satisfied. A naive compilation will initiate a fresh instance of $\mathcal{M}'$ at every step, and run all the instances of $\mathcal{M}'$ concurrently, until one of them detects a fulfillment. Yet, k' is a finite bound, and thus each instance need to be executed only for k' steps. To conserve space, rather than generating an infinite number of concurrently executed instances, existing instances that have executed for longer than k' steps are restarted, so only a finite number of instances are needed for detecting fulfillment.

Let $\mathcal{M}'_1, \mathcal{M}'_2, \cdots, \mathcal{M}'_{k'+1}$ be $k'+1$ copies of $\mathcal{M}'$, such that for each $i \in \{1, 2, \cdots, k'+1\}$, $\mathcal{M}'_i = \langle \mathcal{L}_\Sigma, \mathcal{O}'_i, \iota'_i, S'^i_0\rangle$, and $\mathcal{O}'_i \cap \mathcal{O}'_j = \emptyset$ if $i \neq j$.

Then the compilation for $\mathsf{Eventually}\ \phi'$ is given below:

$$\texttt{compile}(\mathsf{Eventually}\ \phi') = \langle \boldsymbol{mon}, \omega, \mathcal{M}, \mathcal{O}'^1_f \uplus \cdots \mathcal{O}'^{k'+1}_f\rangle$$

where

$$\begin{aligned}\mathcal{M} &= \langle \mathcal{L}_\Sigma, \mathcal{O}'_1 \uplus \cdots \uplus \mathcal{O}'_{k+1} \uplus \mathcal{O}^*, \iota, S'^1_0 \uplus \{o^*_1\}\rangle\\ \mathcal{O}^* &= \{o^*_i \mid i = 1, 2, \cdots, k'+1\}\end{aligned}$$

Here we assume that the o^*_i's are fresh obligation identifiers such that $\mathcal{O}^* \cap \mathcal{O}'_i = \emptyset$. The interpretation function ι is defined as follows:

$$\iota(o) = \begin{cases} \iota'_i(o) & \\ \qquad \text{if } o \in \mathcal{O}'_i\backslash\mathcal{O}'^i_f & \\ \langle \psi, \mathcal{O}_{add}, \mathcal{O}'_1 \uplus \mathcal{O}'_2 \uplus \cdots \uplus \mathcal{O}'_{k'+1} \uplus \mathcal{O}^*\rangle & \\ \qquad \text{if } o \in \mathcal{O}'^i_f \wedge \iota'_i(o) = \langle \psi, \mathcal{O}_{add}, \mathcal{O}_{del}\rangle & \\ \langle \texttt{true}, S'^{i+1}_0 \uplus \{o^*_{i+1}\}, (\mathcal{O}'_{i+1}\backslash S'^{i+1}_0) \uplus \{o^*_i\}\rangle & \\ \qquad \text{if } o \in \mathcal{O}^* \wedge o = o^*_i \wedge 1 \leq i \leq k' & \\ \langle \texttt{true}, S'^1_0 \uplus \{o^*_1\}, (\mathcal{O}'_1\backslash S'^1_0) \uplus \{o^*_{k'+1}\}\rangle & \\ \qquad \text{if } o \in \mathcal{O}^* \wedge o = o^*_{k'+1} & \end{cases}$$

The trigger rule $o^*_i \in \mathcal{O}^*$ is always fired. It terminates the old copy of $\mathcal{M}'_j$ where $j = (i \bmod (k'+1)) + 1$, and starts a new $\mathcal{M}'_j$.

Presented here are but three cases of the compilation algorithm. Consult the companion technical report for the full algorithm [40].

5.3 Compilation Correctness and Type Soundness

Compilation correctness is guaranteed by the next theorem.

THEOREM 12 (CORRECTNESS). *Given any well-typed obligation policy ϕ, suppose* $\texttt{compile}(\phi) = \langle \mathcal{T}, \mathcal{B}, \mathcal{M}, \mathcal{O}_f\rangle$. *Then both of the following hold:*

1. *if $\phi : k$-**enf**, then*
   (a) *$\mathcal{T} = \boldsymbol{enf}$, $\mathcal{B} = k$, and $\phi = \mathcal{L}_{acc}(\mathcal{M})$;*
   (b) *if $k \in \mathbb{N}$, then $\mathcal{L}_{acc}(\mathcal{M})$ is a k-bounded safety property.*
2. *if $\phi : k$-**mon**, then*
   (a) *$\mathcal{T} = \boldsymbol{mon}$, $\mathcal{B} = k$, $\phi = \mathcal{L}_{rec}(\mathcal{M})$, and $\mathcal{M}$ is a pure OM with final trigger rules $\mathcal{O}_f$;*
   (b) *if $k \in \mathbb{N}$, then $\mathcal{L}_{rec}(\mathcal{M})$ is a k-bounded guarantee property.*

A proof of this theorem can be found in the companion technical report [40]. The proof proceeds by structural induction on the derivation tree of the type system. Recall from Sec. 5.1 that $\mathcal{L}_{acc}(\mathcal{M})$ is a safety property, and $\mathcal{L}_{rec}(\mathcal{M})$ is a guarantee property. So Theorem 12 also ensures the soundness of our type system as well.

Summary.

We identified the interface of a compilation algorithm that takes well-typed policy ϕ as input and generates an enforcer or monitor for ϕ in the form of an OM. We illustrated how compilation works via three example cases. We stated the correctness theorem for the compilation algorithm. The complete compilation algorithm and the correctness proof are given in the companion technical report [40].

6. IMPLEMENTATION

The type checker and the compilation algorithm have been fully implemented in 1347 lines of Java code (excluding comments), in approximately 80 man-hours. The compiler takes the abstract syntax tree of a formula as input and returns an OM for enforcing or monitoring the formula. The type checker, which realizes the typing rules in Sect. 4, is integrated into the compiler.

7. RELATED WORK

This section examines some related works with respect to the three stages of the workflow shown in Fig. 1.

Specification.

A number of policy languages have been designed for specifying obligation policies. Some of them [5, 19, 11] provide only syntactic elements for expressing obligations, leaving the semantics open to interpretation. Hilty *et al.* [20] proposed the policy language OSL for expressing obligation policies. They carefully defined both the syntax and semantics for the language, which is based on LTL. Policy enforcement [21] is achieved by compiling OSL to ODRL [22], and directly adopting the enforcement mechanisms of digital rights management (DRM). Compared to OSL, our language not only captures the idiomatic components of obligation policies in the framework of temporal logic, but also employs a type system to articulate composition patterns that preserve implementability.

Kagal *et al.* [24] proposed the language Rei based on deontic logic [29]. We describe in the companion technical report [40] that the simple conditions and trigger rules of OM are sufficient for capturing the four deontic elements in Rei.

Other protection models that model obligations or the triggering of permission changes include TRBAC [7], UCON_{ABC} [32] and P-RBAC [31], among which UCON_{ABC} has been formalized via Lamport's temporal logic of actions.

Implementability check.

Schneider pioneered the study of the enforceability of security properties [37]. The discussion was based on the standard safety-liveness classification of properties [2]. He shows that reference monitors can only enforce safety properties. More recently, Ligatti *et al.* [27, 26, 25] showed that it is possible to enforce at runtime more than safety properties, using edit automata. Yet, the insertion and suppression operations of the edit automata model is sometimes beyond practicality when actually implemented.

A syntactical classification for LTL formulae based on the safety-liveness hierarchy has been presented by Sistla [38]. In comparison, our type system provides incremental feedback to policy developers, helping them to identify policy components that require refinement.

The conflicts between obligations and permissions were studied in [24, 31]. Irwin *et al.* [23] suggested an approach to detect the accountability of obligation policies (i.e. whether a user has sufficient resources or privileges to perform an obligation). Later, Pontual *et al.* [34] studied techniques that can be deployed in a reference monitor to restore accountability. Our work could be seen as a static policy analysis for ensuring that the policy in question can indeed be enforced by a reference monitor.

Implementation.

Dougherty *et al.* [12] defined a rich model of obligations. They also identified that obligation policies contain both safety and guarantee properties. The approach they proposed to enforce non-safety obligations is to use $\mathcal{S}$*-safety closure* [12] to approximate obligations. In contrast, our focus in this paper are policies that are enforceable.

Falcone *et al.* [14, 15, 16] presented a unified view of runtime verification and enforcement of properties in the safety-progress classification. They proposed a systematic approach to produce an enforcement monitor based on the Streett automaton [39]. Their monitor model however relies on the *store* and *dump* operations [14], which faces the same dilemma as the edit automata.

Ribeiro *et al.* [35] suggested enforcing obligation-based security policies using transactional rollback [9, 36]. Gama and Ferreira [17] designed a platform called Heimdall for enforcing obligation-based policies. As these approaches require program actions to be remediable, they are only applicable to certain domains.

8. CONCLUSION AND FUTURE WORK

Conclusion.

In this paper, we argue that humans have long recognized the unenforceability of naively formulated obligation policies, and have devised standard policy idioms for expressing enforceable obligation policies. The contribution of this work is the provision of tool support for helping the policy developer to take advantage of these idioms. First, we designed a policy language, in the form of a temporal logic, for composing obligation policies out of idiomatic policy elements. Second, we formulated a type system for capturing the patterns of policy composition that preserve enforceability. Such a type system can be used by the policy developer to obtain feedback regarding the enforceability of the policy she is working on, and to identify the policy component that is causing the policy not to be enforceable. Third, a compilation algorithm is developed to translate well-typed policies (i.e., enforceable policies) into their corresponding enforcement mechanisms. The compilation algorithm is therefore a justification of soundness for the type system.

Future Work.

This work can be extended in a number of ways. First, we plan to explore the enforcement of obligation policies in a smartphone platform. Our plan is to develop a framework for injecting obligation monitors into Android applications. Second, we plan to explore the automatic optimization of obligation monitors, designing algorithms that will reduce the complexity of an OM generated from our compiler. Third, we are also interested in studying the

theoretical properties of our policy language, especially its relative expressiveness compared to other specification languages such as LTL [28], TPTL [3], MFOTL [6], or the policy language of [18]. Lastly, while we have shown that the type system is sound, whether it is complete remains an open problem. Specifically, it is not known whether there is a formula ϕ for which (a) ϕ specifies a non-empty safety property (resp. non-universal guarantee property), but (b) ϕ can not be typed as k-***enf*** (resp. k-***mon***).

Acknowledgment

This work is supported in part by an NSERC Discovery Grant and by ISSNet — an NSERC Strategic Research Network. We also benefit from discussions with Ida Siahaan.

9. REFERENCES

[1] Irem Aktug and Katsiaryna Nalluka. ConSpec - a formal language for policy specification. In *Proceedings of the 1st International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM'07)*, volume 197 of *Electronic Notes in Theoretical Computer Science*, pages 45–58, Dresden, Germany, February 2007.

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

[3] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–204, January 1994.

[4] James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, U.S. Air Force Electronic Systems Division, Deputy for Command and Management Systems, HQ Electronic Systems Divisions (AFSC), October 1972.

[5] Paul Ashley, Satoshi Hada, Günter Karjoth ad Calvin Powers, and Matthias Schunter. Enterprise privacy authorization language (EPAL 1.2). `http://www.zurich.ibm.com/security/enterprise-privacy/epal`, 2003.

[6] David Basin, Felix Klaedtke, and Samuel Müller. Monitoring security policies with metric first-order temporal logic. In *Proceedings of the 15th ACM Symposium on Access Control Models and Technologies (SACMAT'10)*, pages 23–34, Pittsburgh, PA, USA, June 2010.

[7] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. TRBAC: A temporal role-based access control model. *ACM Transactions on Information and System Security*, 4(3):191–223, 2001.

[8] Claudio Bettini, Sushil Jajodia, X. Sean Wang, and Duminada. Provisions and obligations in policy management and security applications. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, Hong Kong, China, August 2002.

[9] Arnar Birgisson, Mohan Dhawan, Úlfar Erlingsson, Vinod Ganapathy, and Liviu Lftode. Enforcing authorization policies using transactional memory introspection. In *Proceedings of the 15th ACM Conference on Computer and Communication Security (CCS'08)*, pages 223–234, Alexandria, Virginia, USA, October 2008.

[10] Edward Chang, Zohar Manna, and Amir Pnueli. The safety-progress classification. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specifications*, NATO Advanced Science Institutes Series, pages 143–202. Springer, 1991.

[11] Nicodemos Damianou, Daranker Dulay, Emil Lupu, and Morris Sloman. The Ponder policy specification language. In *Proceedings of the 2001 IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'01)*, pages 18–38, London, UK, January 2001.

[12] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Obligations and their interaction with programs. In *Proceedings of the 12th European Symposium on Research in Computer Security (ESORICS'07)*, volume 4734 of *Lecture Notes in Computer Science*, pages 375–389, Dresden, Germany, September 2007. Springer.

[13] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy (S&P'00)*, pages 246–255, Berkeley, California, USA, May 2000.

[14] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Synthesizing enforcement monitors wrt. the safety-progress classification of properties. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS'08)*, volume 5352 of *Lecture Notes in Computer Science*, pages 41–55, Hyderabad, India, December 2008. Springer.

[15] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Enforcement monitoring wrt. the safety-progress classification of properties. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC'09)*, pages 593–600, Honolulu, Hawaii, USA, March 2009.

[16] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime verification of safety-progress properties. In *Proceedings of the 9th International Workshop on Runtime Verification (RV'2009)*, volume 5779 of *Lecture Notes in Computer Science*, pages 40–59, Grenoble, France, June 2009. Springer.

[17] Pedro Gama and Paulo Ferreira. Obligation policies: an enforcement platform. In *Proceedings of the 2005 IEEE Internation Workshop on Policies for Distributed Systems and Networks (POLICY'05)*, pages 203–212, Stockholm, Sweden, June 2005.

[18] Deepak Garg, Limin Jia, and Anupam Datta. Policy auditing over incomplete logs: Theory, implementation and applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, Chicago, Illinois, USA, October 2011.

[19] Simon Godik and Tim Moses. eXtensible access control makeup language (XACML). Technical report, OASIS, May 2002.

[20] Manuel Hilty, Alexander Pretschner, David Basin, Christian Schaefer, and Thomas Walter. A policy language for distributed usage control. In *Proceedings of the 12th European Symposium on Research in Computer Security (ESORICS'07)*, volume 4734 of *Lecture Notes in Computer Science*, pages 531–546,

Dresden, Germany, September 2007. Springer.

[21] Manuel Hilty, Alexander Pretschner, Thomas Walter, and Christian Schaefer. Enforcement for usage control - an overview of control mechanisms. Technical report, DoCoMo Euro-labo, October 2006.

[22] Renato Iannella. Open digital rights language (ODRL) version 1.1. `http://odrl.net/1.1/ODRL-11.pdf`, August 2002.

[23] Keith Irwin, Ting Yu, and William H. Winsborough. On the modeling and analysis of obligation. In *Proceedings of the 13th ACM Conference on Computer and Communication Security (CCS'06)*, pages 134–143, Alexandria, Virginia, USA, October 2006.

[24] Lalana Kagal, Tim Finin, and Anupam Joshi. A policy language for a pervasive computing environment. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'03)*, pages 63–74, Lake Como, Italy, June 2003.

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

[26] Jay Ligatti, Lujo Bauer, and David Walker. Enforcing non-safety security properties with program monitors. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS'05)*, volume 3679 of *Lecture Notes in Computer Science*, pages 355–373, Milan, Italy, September 2005. Springer.

[27] Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of non-safety policies. *ACM Transactions on Information and Systems Security*, 12(3), 2009.

[28] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1992.

[29] John-Jules Ch. Meyer and Roel Wieringa. Deontic logic: a concise overview. In *Deontic Logic In Computer Science*, pages 3–16. Wiley, 1993.

[30] Naftaly H. Minsky and Abe D. Lockman. Ensuring integrity by adding obligations to privilege. In *Proceedings of the 8th International Conference on Software Engineering (ICSE'85)*, pages 92–102, London, England, 1985.

[31] Qun Ni, Elisa Bertino, and Jorge Lobo. An obligation model bridging access control policies and privacy policies. In *Proceedings of the 13th ACM Symposium on Access Control and Technologies (SACMAT'08)*, pages 133–142, Estes Park, Colorado, USA, June 2008.

[32] Jaehong Park and Ravi Sandhu. The UCON_{ABC} usage control model. *ACM Transactions on Information and System Security*, 7(1):128–174, 2004.

[33] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–57, New York, USA, September 1977.

[34] Murillo Pontual, Omar Chowdhury, William H. Winsborough, Ting Yu, and Keith Irwin. On the management of user obligations. In *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies (SACMAT'11)*, pages 175–184, Innsbruck, Austria, June 2011.

[35] Carlos Ribeiro, Andre Zuquete, and Paulo Ferreira. Enforcing obligation with security monitors. In *Third Internatonal Conference on Information and Communications Security (ICICS'01)*, pages 172–176, Xi'an, China, January 2001.

[36] Algis Rudys and Dan S. Wallach. Transactional rollback for language-based systems. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN'02)*, pages 439–448, Washington, DC, USA, June 2002.

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

[38] A. Prasad Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–511, 1993.

[39] Robert S. Streett. Propositional dynamic logic of looping and converse is elementarily decidable. *Information and Control*, 54(1-2):121–141, 1982.

[40] Cheng Xu and Philip W. L. Fong. The specification and compilation of obligation policies for program monitoring. Technical Report 2011-996-08, Department of Computer Science, University of Calgary, Alberta, Canada, March 2012.

[41] Fei Yan and Philip W. L. Fong. Efficient IRM enforcement of history-based access control policies. In *Proceedings of the 4th ACM Symposium on Information, Computer and Communications Security (ASIACCS'09)*, pages 35–46, Sydney, Australia, March 2009.