PoCo: A Language for Specifying Obligation-Based Policy Compositions

Danielle Ferguson*, Yan Albright*, Daniel Lomsak, Tyler Hanks, Kevin Orr, Jay Ligatti
Department of Computer Science and Engineering
University of South Florida

ABSTRACT

Existing security-policy-specification languages allow users to specify obligations, but challenges remain in the composition of complex obligations, including effective approaches for resolving conflicts between policies and obligations and allowing policies to react to other obligations. This paper presents PoCo, a policy-specification language and enforcement system for the principled composition of atomic-obligation policies. PoCo enables policies to interact meaningfully with other policies' obligations, thus preventing unexpected and insecure behaviors that can arise from partially executed obligations or obligations that execute actions in violation of other policies.

KEYWORDS

Information security, policy composition, policy specification, obligations

1 INTRODUCTION

Security-policy composition is a classic problem in software security, due to conflicts that arise when policies have competing requirements. To date, policy composition does not have a complete solution; many languages are domain specific, and the general-purpose solutions may compose obligations in undesirable ways, such as allowing obligations to execute even when they violate the constraints of other policies.

As software becomes more complex, the quantity and severity of security vulnerabilities increases [1]. Managing policies that mitigate these vulnerabilities becomes challenging as the complexity increases; enforcement may devolve into a patchwork of security mechanisms affecting each other in unexpected or hard-to-understand ways, or policies may expand to become complex, monolithic specifications that conflate crosscutting concerns. As the complexity of policies increases, so does the likelihood of errors within the policies.

Following standard software-engineering practices, it is simpler to maintain modules of related functionality, where each security concern can be addressed in isolation, and then build more complex policies as compositions of the modules. When policies are simple enough, as with classic safety properties [2], composition is trivial because the only decision made is whether to permit or deny a given action; such decisions can be composed with boolean operators. However, these simple policies are insufficiently expressive in practice because they do not allow policies to propose alternative or additional actions to be executed. These actions, referred to as obligations [3], complicate the process of composing policies.

Obligations enable policies that are impossible with safety properties. For example, a policy that grants or denies fund transfers may also include an obligation to log such requests for auditing, or a policy to prevent unintended file deletion may include an obligation to prompt the user for confirmation before rendering a decision on a file deletion request.

The challenge of handling conflicts in obligation-based policies is well known (e.g., [4–6]), but neglecting to do so could lead to unexpected behavior or security vulnerabilities. Consider policies P_a and P_b that respectively disallow file downloads and window pop-ups. P_a also defines an obligation to pop up a warning when a user attempts a file download, which violates P_b. A policy P that composes P_a and P_b using *conjunction* (i.e., enforcing both P_a and P_b) should disallow all downloads and pop-ups. However, without validating P_a's obligation with P_b's constraints, P would allow pop-ups.

Beyond these direct policy conflicts, some policies also require the ability to react to other policies' obligations. For example, a policy limiting the number of files open needs access to an accurate count of currently open files—including those opened and closed by other policies' obligations. If this open-file-limiting policy cannot observe and react to actions performed by other policies' obligations, it cannot be enforced.

Contributions. This paper presents PoCo (short for Policy Composition), a new policy-specification language and enforcement system that:

- Allows for principled (i.e., with provable guarantees) composition of complex atomic obligations
- Supports pre-, post-, and ongoing-obligations
- Allows policies and their obligations to be effectful and specified in a Turing-complete language
- Uses static analysis to enable conflict resolution between policies and other policies' obligations
- Allows policies to control and react to obligations
- Supports custom policy-composition operators

As far as we are aware, PoCo is the first system to provide support for atomic obligations, including conflict resolution and allowing policies to react to obligations.

Organization. The remainder of this paper is organized as follows: Section 2 discusses goals for the PoCo enforcement

*Joint first author

Conference'17, July 2017, Washington, DC, USA

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

system, Section 3 system design, Section 4 policy structures, Section 5 policy composition, Section 6 the system implementation, Section 7 related work, and Section 8 conclusions.

2 GOALS

For obligation-based policies to be expressive and composable, an ideal enforcement system would support *1*) pre-, post-, and ongoing obligations, *2*) atomic obligations, *3*) obligations with side effects, *4*) Turing-complete policy specification, *5*) conflict resolution between policies and obligations, *6*) complete mediation of obligations, and *7*) custom composition operators.

Obligation-Type Support. Obligations can be partitioned into categories based on time of execution: *pre-*, *post-*, and *ongoing-* [7–9]. A *pre*-obligation is fulfilled before a decision about an event is enforced. For example, in the file-deletion policy, the confirmation obligation must be enforced before deciding to permit or deny the deletion because the decision depends on the result. A *post*-obligation is fulfilled after the decision is enforced, as in a policy that logs failed transactions. An *ongoing*-obligation is performed asynchronously, as in a policy that monitors system resource usage. If any type is absent, there is a class of policies that cannot be enforced.

Atomic Obligations. An *atomic* obligation requires that either all or none of the obligation's actions are executed. Atomicity can be extended to include the decision to permit or deny an event after the obligation executes. For many policies, atomicity is necessary for correctness. In the file-deletion policy, if the confirmation obligation is executed, the decision entered by the user must also be followed. Otherwise, the policy may incorrectly deny a deletion that the user confirmed or permit a file deletion that the user canceled.

Obligations with Side Effects. Related work (e.g., [5]) requires obligations to be side-effect free, making some policies unenforceable. For example, any obligation that prompts the user or makes a remote call causes side effect(s) that cannot be undone; any mechanism that relies on rolling back obligations may be unable to manage such effectful obligations correctly.

Complete Mediation of Obligations. Policies sometimes need to react to obligations. For example, the open-file-limiting policy needs access to the number of open files. Excluding files opened or closed during obligation execution may cause the policy to have an inaccurate count, leading to incorrect enforcement. The ability to monitor all events, including those executed by policy obligations, is called *complete mediation* [10].

Turing Completeness. Turing-complete languages ensure expressiveness, at the cost of non-guaranteed termination (discussed in Section 3.6). Tools, like PoCo, that aim for general-purpose policy specification, prioritize expressiveness and leave it to the policy author to limit non-termination.

Conflict Resolution. Several types of conflicts exist in policy enforcement; policies may disagree on a permit/deny decision, an obligation may be disallowed by another policy, or multiple policies may have obligations for the same event. Disagreement between policies on a permit/deny decision is simplest and can be resolved with Boolean algebra. However, when an obligation violates the rules of another policy, or the ordering of obligations is important, the resulting behavior can be inconsistent with the policies' behavior in isolation; it is important to handle these conflicts in a predictable manner.

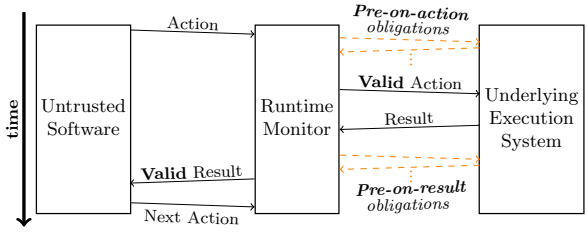

Figure 1: Obligations are either pre-on-action or pre-on-result

Custom Composition Operators. There is an infinite number of strategies to compose policies. Some policies need higher priority; other policies may only trigger under certain conditions; the decision of one policy may only matter when another agrees; etc. It is, therefore, desirable to allow custom logic for composing policies.

3 THE POCO MONITOR ARCHITECTURE

PoCo's enforcement mechanism operates as a *monitor* that observes a target application's security-relevant *actions* (e.g., system or function calls) and the *results* of these actions, as shown in Figure 1. These actions and results can trigger the monitor to respond based on the logic of the enforced policies.

3.1 Monitor Operation

The PoCo monitor observes all security-relevant *events*—actions and results—and broadcasts each event to all policies. Policies take each trigger event e and suggest an obligation to be executed before e is processed. This obligation, which may be empty, can implement supplemental logic or alter e to meet the policy's goals. In PoCo, security-relevant events are inferred from the logic of enforced policies and can be further refined by the policy author. This ensures that the monitor only broadcasts the events required for policy enforcement.

The PoCo monitor can execute any number of obligations before relinquishing control back to the target application by returning a result to it. After relinquishing control, PoCo cannot execute additional obligations until receiving a new event. The monitor, therefore, operates in a loop, with each iteration performing the following steps:

(1) Input security-relevant event e
(2) Collect obligations from policies in response to e
(3) While there are obligations to process
  (a) Select an obligation o
  (b) Collect and process policies' *votes* on o
  (c) If o is approved, execute o
  (d) Collect obligations triggered in response to o
(4) If a new output event has been set, execute or return it
(5) Otherwise, output the original input event

This repeats until the output event is a result to be returned to the target application. With this design, the monitor maintains control of execution until all approved obligations have executed, that is, the pool of pending obligations is exhausted.

3.2 Monitor Configuration

Before examining the PoCo language, it is useful to have a high-level understanding of the options available for composing policies; a more detailed discussion appears in Section 5. Three elements can be supplied to the PoCo monitor: a list of policies to enforce, a *vote combinator* to combine policies' votes on an obligation into a single permit/deny decision, and an *obligation scheduler* to prioritize obligations for execution. The PoCo monitor can be viewed as an obligation dispatcher; it decides which obligations to execute and in what order. The monitor's parameters allow these decisions to be customized.

3.3 Obligations

The literature on policy enforcement has many definitions of *obligation* [3–9, 11]. Generally, an obligation is one or more actions required to execute under certain circumstances with specific timing in relation to security-relevant events. When conflict resolution is introduced, the idea that an obligation is guaranteed to execute must become less strict. When an obligation conflicts, the only options are to execute the obligation (i.e., ignore the conflict), execute the parts of the obligation that do not conflict (i.e., non-atomic obligations), or do not execute the obligation (i.e., execution is not guaranteed).

Since PoCo's goal is to resolve conflicts among atomic obligations, the only option is to not execute conflicting obligations. Other works have referred to this concept as "suggestions" since they are not guaranteed to execute [4]. However, even XACML—which does not provide obligation conflict resolution—suffers from non-guaranteed execution under certain circumstances [12, Section 7.18]. Therefore, we have opted to use the term *obligation* over such a variant with the understanding that the monitor is obligated to attempt execution.

3.4 Pre-obligation Completeness

Although the categories *pre-*, *post-*, and *ongoing-* are standard, they can all be implemented as *pre-*obligations by expanding the domain of security-relevant events to include actions and results. We refer to this property as *pre-obligation completeness*.

With this expanded definition of events, *pre-on-action* and *pre-on-result* obligations can be defined. An obligation o is *pre-on-action* to action a if o is fulfilled after a is requested by the application but before the monitor makes a decision regarding a. An obligation o is *pre-on-result* to result r if o is fulfilled after r is returned from the system but before the monitor makes a decision regarding returning r to the application.

Pre- and *pre-on-action* obligations are equivalent as are post- and *pre-on-result* obligations. Ongoing obligations can be defined using *pre-on-action* and *pre-on-result* obligations in any multi-threaded environment. This means that only *pre-on-action* and *pre-on-result* obligations are needed to support all standard obligation categories and, as such, these are the only obligations that are implemented in PoCo.

3.5 Complete Mediation of Obligations

Complete mediation—the ability to monitor events executed by other policies—is a desirable trait when enforcing obligation policies. Complete mediation generally means that each event can be responded to individually. We refer to this design as *event-by-event complete mediation*. At least one existing system has provided event-by-event complete mediation but without allowing for atomic obligations [4]. In fact, as Theorem 1 shows, it is impossible to have both event-by-event complete mediation and atomic obligations.

Theorem 1. *Event-by-event complete mediation and atomic obligations cannot both be achieved simultaneously.*

Proof. *For all monitors m, if m allows event-by-event complete mediation of policy obligations, then m must allow all policies that it enforces to examine and react to each event in an obligation o as it executes. If any of m's policies alter any event in o, then o was not executed atomically.*

Therefore, PoCo enforces *obligation-by-obligation complete mediation*, meaning that every policy can monitor and react to every other policy's atomic obligations (rather than every individual event within those obligations).

3.6 Non-termination of Policy Enforcement

By including branching, looping, and variables, PoCo is Turing complete, which introduces possible non-termination in enforcement code; e.g., policies may contain infinite loops. In addition, allowing policies to react to each other introduces an additional path to non-termination—two policies may generate an infinite sequence of obligations in response to each other's obligations (e.g., one policy monitors network connections and logs them to a file while another monitor's file writes and opens a network connection on each). This non-termination cannot be statically detected. This design prioritizes policy expressiveness over guaranteed enforcement termination.

3.7 PoCo Language Formalization

To express PoCo's core features in a precise and unambiguous manner and enable formal type-safety reasoning, the formal syntax and semantics for the PoCo Language were defined in the companion technical report [13]. PoCo is formalized as a functional language due to the inherently simpler specification compared to object-oriented languages such as Java. Using these semantics, the PoCo language was proven type safe through standard type-preservation and progress lemmas [14].

We have also proven four properties of PoCo obligations: *1*) obligations are atomic *2*) obligations allow conflict resolution *3*) policies react to obligations *4*) pre-obligations can implement post- and ongoing obligations. These proofs can also be found in the companion technical report [13].

4 POCO POLICIES

PoCo policies are granular pieces of logic that specify obligations in response to events. Each security-relevant event is broadcast to all enforced policies. The following policies will serve as examples throughout this section. *1*) P_{file} disallows users from opening the *secret.txt* file. *2*) $P_{postlog}$ logs each file-open action after it occurs. *3*) $P_{confirm}$ requires each file-open action to be confirmed through a pop-up window. *4*) P_{time} disallows popups unless 100 seconds have passed

since the last popup. These examples illustrate the core features of PoCo policies.

4.1 The `onTrigger` Policy Function

The first component of a PoCo policy, `onTrigger`, is an obligation which executes prior to security-relevant actions or results. The `onTrigger` takes an event as input and specifies logic to run before it is executed or returned. `onTrigger` may also set an *output event*, which is PoCo's final response to a trigger. Ultimately, PoCo must cede control to allow the application or system to continue executing. If no policy specifies an output event, the monitor uses the trigger to cede control.

For example, P_{file}'s `onTrigger` examines the trigger event e. If e is fopen(secret.txt) then P_{file}'s `onTrigger` sets `exit` as the output event, meaning that the monitor should cede control to the system to execute the `exit` action. P_{file} does not specify an output event when e is not fopen(secret.txt), thus allowing irrelevant events to execute normally. Hence, P_{file}'s `onTrigger` is defined as follows.

```
fun onTrigger(e:Event):Unit =
   (case e of act a =>
      if a.name == "fopen"
         ∧ tryCast(String,a.arg) == "secret.txt"
      then
         setOutput(event(act("exit",
            makeTypedVal(Unit,unit) ) ) ); unit
      else unit
   | res r => unit )
```

Output events are treated specially because the monitor must reach agreement on how to cede control. One of the objectives of any system for composing run-time policies must be to determine the output event for each trigger. Output actions cede control to the system; output results cede control to the application. Prior work has defined monitors that operate in this way, interposing between application and executing system and responding to trigger events with output events [15].

As seen in P_{file}'s `onTrigger`, setting the output event is done with `setOutput`. This call commits the monitor to using that output event. Once `setOutput` has been called for a trigger event, additional calls return $false$ indicating that the output cannot be overwritten. A policy may call `getOutput` or `outputNotSet` to get the current state; policy logic determines what happens if the output event is already set.

This ability to permanently set the output event is required for some policies' correctness. For example, $P_{confirm}$ tests whether the trigger, e, is a file-open action. If it is, then an obligation is specified to confirm e. Based on the result, the output is set to e (indicating the file open must be executed) or *unit* (indicating an empty result must be returned in lieu of opening the file). If it were possible for the output to not be used, the user could opt to allow a file open and it not execute, or the user could opt to disallow the file open and it execute anyways. This level of control also enables policies to self-manage when they conflict with other policies.

4.2 The `vote` Policy Function

The second part of a policy is the `vote` function, which votes on whether an obligation should be executed. The `vote` function takes an obligation and returns a boolean indicating approval or disapproval. For example, P_{file} tries to prevent secret.txt from being opened, even by other obligations. Therefore, P_{file} looks for fopen(secret.txt) in the obligation and, if found, it votes to disallow the obligation. Otherwise, P_{file} votes to allow it. Hence P_{file}'s `vote` is:

```
fun vote(cfg:CFG):Bool =
   ¬call(containsAct, (cfg=cfg, name="fopen",
   arg=(in_arg makeTypedVal(String, "secret.txt"))
      :(arg:TypedVal + none: unit), count=1)))
```

To ensure obligation atomicity, `vote` analyzes obligations before execution—specifically, policies vote on obligations' statically generated Control Flow Graphs (CFGs). These CFGs are conservative approximations since computing the exact CFG for an arbitrary program is undecidable. Because it is not always possible to determine the arguments to actions statically, it is necessary to allow obligation CFGs to identify such arguments as *unresolved* and allow policies to specify how to handle them.

4.3 The `onObligation` Policy Function

The third component of a policy is an obligation that responds to other obligations. This allows policies to react to other obligations. `onObligation` analyzes the results of all security-relevant actions performed during an obligation's execution, called a *result trace* (rt). For example, $P_{postlog}$ logs each file opened in other obligations with an `onObligation` as follows:

```
fun onObligation(rt: Res_List):Unit =
   (let results=ref rt in
      while(¬empty(!results)) {
         let event = head(!results) in
            results := tail(!results);
            if event.act.name == "fopen"
            then call(log,event)
            else unit
       end }
   end)
```

PoCo cannot insert obligations before execution of a triggering obligation without creating inconsistency. Prior to execution of obligation o_1, a decision on whether to execute it is made. Inserting obligation o_2 prior to o_1 may alter this decision to permit o_1. If PoCo re-queried policies after o_2 and the decision was to not execute o_1, it is possible that o_2 should not have been proposed. To have predictable behavior, voting on and execution of an obligation must be an atomic unit.

To summarize, there are two ways to specify obligations: `onTrigger` for obligations in response to trigger events, and `onObligation` for obligations in response to other obligations (which may be `onTrigger` or `onObligation`). The `vote` function enables policies to approve or disapprove of obligations.

4.4 Parameterized Policies and Local State

As in other works [4, 16], PoCo enables abstraction over common policy patterns to aid in code reuse by allowing functions to instantiate policies based on arguments. For example, the set of policies to disallow a specific action can be abstracted over by accepting the disallowed action as an argument. Other

uses could be to specify directory paths, port numbers, or any other data that may be relevant to a specific policy.

Without the ability to keep state, any policy that needs to "remember" earlier events cannot be enforced [17]. PoCo policies utilize let environments and memory references to manage this data. P_{time} tracks the last time a popup was displayed and exits the application if it opens another popup within 100 seconds. The last occurrence of the popup can be stored in a memory reference initialized by a let environment. For interested readers, the PoCo technical report [13] presents complete specifications of these policies. Their construction follows directly from the components described in this section.

5 POLICY COMPOSITION

The PoCo monitor handles composition of policies by scheduling obligations, dispatching the output event, and handing control to the application or system.

Conflicts in composition fall into two categories. The first is from obligation o attempting an action that policy p disallows. In PoCo, this manifests as p's `vote` function returning "deny" on o. This conflict is handled by a *vote combinator* that combines all votes into a final permit or deny decision. The second is a timing issue between obligations; If the execution of obligation o_1 renders the execution of obligation o_2 undesirable, execution of o_1 should cause o_2 not to execute. This conflict is handled by using the *obligation scheduler* to execute the most vital obligations first.

Using the parameters provided allows both types of conflicts to be handled in the manner that the policy architect decides is the best fit for their use case. The following sections consider each of these configuration parameters in turn.

5.1 Vote Combinator

The *Vote Combinator* or VC is a function for combining the boolean outputs of the policies' `vote` functions into a decision on if an obligation should be executed. In addition to the vote of each policy, the VC may need the policy name to implement combinators that give preference to specific policies. Therefore, the type of this argument is $(name : String \times vote : Bool)_{List} \rightarrow Bool$.

A VC can implement any logic that is desired. For example, one could write a VC that executes an obligation if a specified policy, say Pol1, does not veto it. This VC would look like:

```
fun VC_vote(votes:(name:String×vote:Bool)_List):Bool=
  let output = ref true in
    let rvotes = ref votes in
      while(¬empty(!rvotes)) {
        case head(!rvotes) of
          some v =>
            if v.name == "Pol1"
            then output := v.vote
            else unit
          none unit => unit;
        rvotes := tail(!rvotes)} end; !output end
```

The PoCo implementation includes built-in VCs that can be used in their entirety or as a building block to create other VCs. For example, one could implement a VC that allows an obligation if either the first policy allows it or all other policies allow it using the built in conjunction and disjunction VCs.

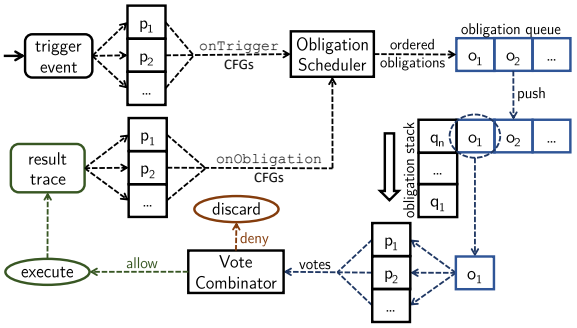

Figure 2: PoCo obligation flow — obligations are generated by policies, prioritized by the obligation scheduler and voted on.

A convenient side effect of PoCo's event-broadcasting and voting mechanism is that policy conflicts are obvious during execution of the VC; votes to disallow an obligation or any vote that gets overruled by the VC are conflicts between policies. It is, therefore, straightforward to detect and act on these conflicts dynamically by adding logic to the VC.

5.2 Obligation Scheduler

The *Obligation Scheduler* or OS is a function that orders obligations based on specific criteria. This prioritization is important because it determines the single output event for a given input event. Lower-priority obligations will not be able to set the output event if an obligation of higher priority has already set it. Like the `vote` function for policies, the OS works with CFG representations of obligations, therefore the type for the OS is $(pol : Pol \times cfg : CFG)_{List} \rightarrow (pol : Pol \times cfg : CFG)_{List}$.

The OS allows arbitrary logic to perform its function. One example OS could be a strict ordering of policies. If the policy writer wants to prioritize the obligations in the order that the policies were provided to the monitor, they could simply return the same list. The PoCo implementation includes example OSs including this default ordering:

```
fun OS_default(obs:(pol:Pos × cfg:CFG)_List):
    (pol:Pos × cfg:CFG)_List = obs
```

Other interesting ways of ordering obligations may include prioritizing simpler obligations, weighting specific actions, or applying priorities to the policies.

5.3 Monitor Operation

Now that all monitor inputs have been described, let's examine how PoCo uses them during policy composition. For each security-relevant event, PoCo uses the specified *OS* to generate an ordered list of obligations based on their CFGs. These obligations are pushed onto a stack that holds all obligations waiting to execute.

To process an obligation, PoCo pops the top obligation off the stack and collects votes on it from all policies. The VC composes these votes into a single permit/deny decision. A denied obligation will be discarded; a permitted obligation will be executed, and its *result trace* will be dynamically collected.

To avoid time-of-check to-time-of-use (TOCTOU) vulnerabilities, the voting on and execution of an obligation happen sequentially in a single thread. PoCo uses the result trace and policies' `onObligations` to determine if the executed obligation triggers additional obligations. As with the *onTriggers*, each obligation is pushed onto the stack. This ensures that obligations generated by other obligations are executed as soon as possible after execution of the trigger. Then PoCo repeats this process until the obligation stack is empty. Figure 2 illustrates this process.

6 IMPLEMENTATION AND CASE STUDY

We have implemented a prototype of PoCo to evaluate and refine its design. The implementation, written in Java, consists of 3,299 lines of code and is available online [18]. To demonstrate the expressiveness and analyze the performance of PoCo, we replicated the case study that was used to validate Polymer [4], which is the most directly comparable work.

6.1 Implementation

The PoCo compiler builds a trusted application by inlining security-enforcement code into an untrusted application using AspectJ [19], an aspect-oriented extension to Java. The AspectJ compiler inlines code, called *advice*, that executes before and/or after methods specified with *pointcuts* [20]. The decision to use AspectJ over manual byte code re-writing was made for simplicity and because byte code re-writing to enforce runtime policies has already been accomplished by other projects [4] so there is no novelty in a new implementation.

The PoCo compiler is comprised of four modules: the *pointcut extractor*, *policy converter*, *static analyzer*, and *AspectJ compiler*. Following the flow of code translations, the PoCo compiler takes a list of policies specified in `.pol` files and uses the *pointcut extractor* to create an AspectJ (`.aj`) file with all methods monitored by the policies as its pointcut set. Next, the *policy converter* reconstructs the `.pol` files into Java (`.java`) files and creates a policy-scheduler file using the specified obligation scheduler and vote combinator (in `.os` and `.vc` files respectively). Then the *static-analyzer* creates CFGs to represent the actions that may be invoked for each obligation. Finally, the *AspectJ compiler* inlines the policy-enforcement code into the target application.

6.2 Case Study

To evaluate the implementation, we replicated Polymer's case study [4] by implementing ten policies to prevent unsafe behavior in an email client. Encoded in 1138 lines of PoCo (available online at [18]), these policies were enforced on Pooka [21], an open-source email client, without modifying the application's source code. All policies show identical behavior in PoCo and Polymer. When enforcing these policies, the monitor *1*) allows only POP and IMAP email protocols (or HTTP(S) with user confirmation); *2*) logs incoming emails, flags emails from unknown addresses as spam, and warns users about potentially dangerous attachments; *3*) confirms recipients of outgoing emails, adds a BCC and logs them; and *4*) monitors memory usage and prevents dynamic class loading, reflection, or system calls.

PoCo performance was measured as the run-time overhead incurred by the system during application and email loading on a MacBook Pro laptop. Overhead was measured for four scenarios: no enforced policies, one *Trivial* policy, ten *Trivial* policies, and the entire composed email policy, thereby establishing how much overhead is due to the monitor versus individual policies. The empirical results demonstrate that the monitor's overhead is relatively low. With one trivial policy and ten trivial policies, the average overheads for loading the client are 0.87% and 1.54%, respectively, and the overheads for loading an email are 0.39% and 5.06%. The overhead of policy enforcement is dominated by obligation logic, which varies by policy. Obligations can run for an arbitrarily long time, so the overhead of a composed policy is almost entirely dependent on the complexity of its obligations.

7 RELATED WORK

eXtensible Access Control Markup Language [22] allows policies to be composed in XML. XACML allows policies to return one of four result values and, optionally, an obligation, to express its response to a request. Due to its stateless nature and relatively simple rule structure, XACML has been widely adopted and has been implemented into commercial and open-source software products. However, even with significant research extending XACML to overcome its limitations [23–27], it is still lacking in some areas. Stateless policies are less expressive than stateful policies [17] and cannot express simple policies such as "disallow network sends after file reads" [28].

Polymer is an object-oriented language with well-defined semantics that allows policy composition for Java programs [4]. Polymer policies issue "suggestions" indicating what they want the monitor to do. By separating policies into an effect-free *query* method and an effectful *accept* method, Polymer ensures that querying a policy will have no permanent effect when its suggestion is not followed. Because Polymer implements event-by-event complete mediation, it cannot ensure obligation atomicity (Theorem 1).

Ponder composes access-control and general-purpose policies [6, 29] based on logical relations between policies and hierarchical relationships between subjects' policies. Complex obligations are specified with *concurrency operators*. If any action in an obligation violates a policy, the target application halts. Like Polymer, Ponder inspects actions individually, so the execution of an obligation can be interrupted which prevents obligation atomicity. Ponder does not allow policies to react to obligations; the only allowed response to a conflict is to halt the application, which may be unacceptable in practice.

Security Policy Language (*SPL*) enables composition of authorization policies using policy combinators [5, 30] to resolve conflicts. SPL requires obligations to be atomic as conflicts are resolved by resetting the application state to before the execution of the trigger action. For this solution to work, obligations must be pure (free of side-effects), because effectful actions generally cannot be rolled back. Excluding effectful obligations significantly limits SPL's expressiveness.

Heimdall uses *compensatory* actions in response to failures in obligations [31]. However, there may not always exist an effective compensation for security violations; a policy may prevent future leakage of sensitive data but be unable to compensate for data that has already been leaked. Heimdall does not support conflict resolution between policies and obligations; obligations are not validated against other policies before execution. When an obligation is not fulfilled, Heimdall executes the compensatory action of the obligation. Enforced policies are unable to react to the executed obligations.

Rei identifies conflicts between *pre-on-result* obligations and prohibition policies and offers two ways to resolve them [32]. The first is to specify priorities among policies or rules and the second is to set negative/positive-modality on actions, entities, or policies. The authors do not address conflicts in complex obligations specifically, but the context suggests that actions are handled individually, and thus complex obligations would not be atomic. It is also unclear if Rei is able to react to other policies' obligations since the exact details of how obligations are enforced are not included.

No previous works that we are aware of have been able to accomplish all goals set for PoCo (Section 2). SPL and Heimdal do not handle effectful obligations as they rely on rollback mechanisms, only SPL allows resolution of conflicts among atomic obligations and none of the previous work that we are aware of allows policies to react to other atomic obligations.

8 CONCLUSIONS

PoCo is a policy-specification language and enforcement system that enables principled composition of atomic obligations. It is Turing complete and supports effectful pre-, post-, and ongoing-obligations. PoCo employs static analysis to allow policies to validate the obligations of other policies before they are executed. PoCo also allows policies to react to completed obligations and enables custom operators to define how policies should be prioritized and combined. Taken together, these techniques enable a versatile composition of atomic obligations with the most granular complete mediation possible (Theorem 1). PoCo has been implemented and evaluated by enforcing a case-study composition of ten policies for securing an email client. We have also defined the formal syntax and semantics for the PoCo Language and proven it to be type safe and that it supports atomic obligations, conflict resolution between obligation policies, and allows policies to react to obligations. As far as we are aware, PoCo is the first system to provide support for atomic obligations, including conflict resolution and allowing policies to react to obligations.

REFERENCES

[1] S. Moshtari, A. Sami, and M. Azimi, "Using complexity metrics to improve software security," *Computer Fraud & Security*, pp. 8 – 17, 2013.

[2] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, pp. 125–143, March 1977.

[3] C. Xu and P. W. Fong, "The specification and compilation of obligation policies for program monitoring," in *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, pp. 77–78, 2012.

[4] L. Bauer, J. Ligatti, and D. Walker, "Composing expressive runtime security policies," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 3, pp. 1–43, 2009.

[5] C. Ribeiro, A. Zúquete, P. Ferreira, and P. Guedes, "SPL: An access control language for security policies with complex constraints," in *In Network and Distributed System Security Symposium*, pp. 89–107, 2001.

[6] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "Ponder: A language for specifying security and management policies for distributed systems," tech. rep., Imperial College. UK, Research Report Department of Computing, 2000.

[7] J. Park and R. Sandhu, "Towards usage control models: beyond traditional access control," in *Proceedings of the ACM symposium on Access control models and technologies*, pp. 57–64, 2002.

[8] J. Park and R. Sandhu, "The UCON ABC usage control model," *ACM Transactions on Information and System Security*, vol. 1, no. 1, pp. 128–174, 2004.

[9] C. Bettini, S. Jajodia, X. S. Wang, and D. Wijesekera, "Provisions and obligations in policy management and security applications," in *Proceedings of the International Conference on Very Large Data Bases*, pp. 502–513, 2002.

[10] J. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[11] K. Irwin, T. Yu, and W. H. Winsborough, "On the modeling and analysis of obligations," in *Proceedings of the 13th ACM conference on Computer and communications security*, pp. 134–143, 2006.

[12] OASIS, "eXtensible Access Control Markup Language (XACML) Version 3.0." http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html. Accessed: 2017-04-22.

[13] D. Ferguson, Y. Albright, D. Lomsak, T. Hanks, K. Orr, and J. Ligatti, "Composition of atomic obligation security policies." http://www.cse.usf.edu/ ligatti/papers/PoCoTR.pdf, 2019.

[14] A. Wright and M. Felleisen, "A syntactic approach to type soundness," pp. 38–94, 1994.

[15] J. Ligatti and S. Reddy, "A theory of runtime enforcement, with results," in *Proceedings of the 15th European conference on Research in computer security*, ESORICS, pp. 87–100, 2010.

[16] D. Lomsak and J. Ligatti, "PoliSeer: A tool for managing complex security policies," *Journal of Information Processing*, vol. 19, pp. 292–306, 2011.

[17] P. W. L. Fong, "Access control by tracking shallow execution history," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 43–55, IEEE, May 2004.

[18] Y. Albright, "PoCo source code." https://github.com/caoyan66/PoCo-PolicyComposition/, 2018.

[19] AspectJ, "The Aspectj & Project." https://www.eclipse.org/aspectj. Accessed: 2017-04-12.

[20] J. Ligatti, B. Rickey, and N. Saigal, "LoPSiL: A location-based policy-specification language," *Security and Privacy in Mobile Information and Communication Systems*, pp. 265–277, 2009.

[21] A. Petersen, "Pooka: A java email client, 2003." http://www.suberic.net/pooka/. Accessed: 2018-01-12.

[22] H. L. Bill Parducci and R. Levinson, "Oasis extensible access control markup language (xacml)." http://www.oasis-open.org/committees/tc_home.phpl, 2012.

[23] J. Alqatawna, E. Rissanen, and B. Sadighi, "Overriding of access control in xacml," in *Policies for Distributed Systems and Networks, 2007. POLICY '07. Eighth IEEE International Workshop on*, pp. 87–95, June 2007.

[24] D. W. Chadwick, L. Su, and R. Laborde, "Providing secure coordinated access to grid services," in *Proceedings of the 4th International Workshop on Middleware for Grid Computing*, MCG '06, (New York, NY, USA), pp. 1–, ACM, 2006.

[25] N. Li, Q. Wang, W. Qardaji, E. Bertino, P. Rao, J. Lobo, and D. Lin, "Access control policy combining: theory meets practice," in *Proceedings of the 14th ACM symposium on Access control models and technologies*, SACMAT '09, (New York, NY, USA), pp. 135–144, ACM, 2009.

[26] M. Lischka, "Dynamic obligation specification and negotiation," in *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pp. 155–162, April 2010.

[27] N. Li, H. Chen, and E. Bertino, "On practical specification and enforcement of obligations," in *Proceedings of the ACM conference on Data and Application Security and Privacy*, pp. 71–82, 2012.

[28] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, pp. 30–50, Feb. 2000.

[29] K. Twidle, N. Dulay, E. Lupu, and M. Sloman, "Ponder2: A policy system for autonomous pervasive environments," in *International Conference on Autonomic and Autonomous Systems*, pp. 330–335, 2009.

[30] C. Ribeiro, A. Zúquete, and P. Ferreira, "Enforcing obligation with security monitors," in *International Conference on Information and Communications Security*, pp. 172–176, 2001.

[31] P. Gama and P. Ferreira, "Obligation policies: An enforcement platform," in *IEEE International Workshop on Policies for Distributed Systems and Networks*, pp. 203–212, 2005.

[32] L. Kagal, T. Finin, and A. Joshi, "A policy language for a pervasive computing environment," in *Proceedings POLICY 2003. IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, pp. 63–74, June 2003.