March 2020

Composition of Atomic-Obligation Security Policies

Yan Cao Albright

University of South Florida

Follow this and additional works at: https://scholarcommons.usf.edu/etd

Part of the Computer Engineering Commons

Scholar Commons Citation

Albright, Yan Cao, "Composition of Atomic-Obligation Security Policies" (2020). Graduate Theses and Dissertations.
https://scholarcommons.usf.edu/etd/8154

This Dissertation is brought to you for free and open access by the Graduate School at Scholar Commons. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact scholarcommons@usf.edu.
Composition of Atomic-Obligation Security Policies

by

Yan Cao Albright

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Jay Ligatti, Ph.D.
Adriana Iamnitchi, Ph.D.
Yicheng Tu, Ph.D.
Sean Barbeau, Ph.D.
Brendan Nagle, Ph.D.

Date of Approval:
04/23/2020

Keywords: Security, Policy composition, Policy specification, Obligations

Copyright © 2020, Yan Cao Albright
# Table of Contents

List of Tables ........................................................................................................ iv

List of Figures ........................................................................................................ v

Abstract ................................................................................................................ vii

Chapter 1: Introduction ......................................................................................... 1

Chapter 2: Obligation and Obligation Policies ..................................................... 8
  2.1 Pre-on-action and Pre-on-result Obligations ............................................. 9
    2.1.1 Pre-on-action obligations .................................................................. 10
    2.1.2 Pre-on-result obligations ............................................................... 10
  2.2 Other Different Categorizations ............................................................... 12
    2.2.1 Different kinds of obligations based on their complexity ............... 12
    2.2.2 Different kinds of obligations based on how they should be executed ................................................................................................. 12
  2.3 Summary ................................................................................................. 13

Chapter 3: Desired Properties of PSLs for Composing Obligation Policies ....... 14
  3.1 Obligation-type Support ......................................................................... 14
  3.2 Atomic Obligations ............................................................................... 15
  3.3 Obligations with Side Effects ................................................................ 15
  3.4 Complete Mediation of Obligations ...................................................... 16
  3.5 Turing Completeness ........................................................................... 16
  3.6 Conflict Resolution .............................................................................. 16
  3.7 Custom Composition Operators ......................................................... 17
  3.8 Summary ............................................................................................... 18

Chapter 4: Related Works .................................................................................... 19
  4.1 XACML ................................................................................................. 19
  4.2 ExtXACML ......................................................................................... 22
  4.3 Polymer ............................................................................................... 23
  4.4 Ponder ................................................................................................. 25
  4.5 SPL .................................................................................................... 27
  4.6 Heimdall .............................................................................................. 28
List of Tables

Table 4.1  Summary of the extent to which existing PSLs satisfy the goals listed in Chapter 3. ................................................................. 31

Table 10.1  PoCo performance statistics on email client over 100 runs. ......................... 70

Table 10.2  PoCo performance statistics on WordPress over 150 runs. ......................... 79
List of Figures

Figure 2.1 Obligations are either pre-on-action which should be fulfilled before a decision being enforced (time $t_1 < t_2$) or pre-on-result which should be fulfilled before an enforcement result being returned (time $t_4 < t_5$). .......... 11

Figure 4.1 The policy Login permits legitimate users to login only during the working hours (from 9:00am to 5:30 p.m.). It also includes an obligation InvalidLogin, which is carried out whenever the policy denies an invalid log-in request. ................................................ 20

Figure 5.1 Obligations are either pre-on-action, which are fulfilled before a decision on an action is enforced, or pre-on-result, which are fulfilled before a decision is made to return a result.................................................. 36

Figure 6.1 Formal syntax for PoCo ................................................................................. 40

Figure 6.2 Static semantics (rules for actions, results, events, CFGs, labels, and function calls).................................................... 42

Figure 6.3 Dynamic semantics (selected rules)................................................................. 44

Figure 9.1 Overview of the PoCo compiler architecture. The compiler takes as input an untrusted application and outputs the same application with policy-enforcement code inlined before and after all security-relevant methods. Ovals are used to represent code files while rectangles represent processes executed during compilation.................................................. 61

Figure 9.2 The control flow graph of $P_{confirm}$’s onTrigger function......................... 65

Figure 9.3 Policy scheduler flow — obligations are generated by policies based on the trigger event, prioritized by the obligation scheduler and then voted on. Executed obligations can result in additional obligations ............... 66

Figure A.1 The PoCo Monitor Algorithm................................................................. 92

Figure B.1 pol_file disallows users and obligations from opening the secret.txt file .......... 93

Figure B.2 pol_disallow fun disallow specifies a family of policies which disallow the action x ......................................................... 93
Figure B.3  $pol_{postlog}$ logs all file-opens after they occur .................................................. 94

Figure B.4  $pol_{prelog}$ logs file-open actions before they are executed .......................... 94

Figure B.5  $pol_{confirm}$ requires all file-open attempts to be confirmed by the user through a pop-up window .......................................................... 95

Figure B.6  $pol_{time}$ disallows pop-ups unless at least 100 seconds have passed since the last pop-up .......................................................... 95
Abstract

Existing security-policy specification languages allow users to specify obligations, but open challenges remain in the composition of complex obligations, including effective approaches for resolving conflicts between policies and obligations and allowing policies to react to the obligations of other policies.

An atomic obligation requires that either all or none of the included actions are executed. Atomicity can be extended to include the decision to permit or deny an event after the obligation executes. For many practical policies, obligation atomicity is necessary for correctness. Executing only the parts of such an obligation violates its atomicity which can lead to an undesirable result.

Presented here in this dissertation is PoCo, a policy Specification language and enforcement system for the principled composition of atomic-obligation policies. PoCo enables policies to interact meaningfully with the obligations of other policies, thus preventing the unexpected and insecure behaviors that can arise due to partially executed obligations or obligations that execute actions in violation of other policies. As far as we are aware, PoCo is the first system that supports the composition of atomic obligations, including conflict resolution between policies and obligations as well as allowing a policy to react to obligations of other policies.
Chapter 1: Introduction

The importance of, and interest in, software security has been increasing, as has our reliance on software in our daily lives. One common way for ensuring the security of software is to enforce security policies, typically runtime policies enforceable by monitoring and modifying software’s behavior at run time. For instance, password-protection policies can be enforced to prevent unauthorized access to computers and data; access-control policies can be enforced to protect sensitive data from being compromised; firewall policies can be enforced to filter forbidden or unwanted traffic, etc.

As software becomes more complex, the quantity and severity of security vulnerabilities increases [64]. 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 cross-cutting 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 [35], 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.

\footnote{Parts of this chapter is published in the International Conference on Software and Computer Applications[20]. Permissions to use the materials are provided in Appendix C.}
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 [74], complicate the process of composing policies.

Obligation support enables 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.

Most modern Policy Specification Languages (PSLs) support specifying obligation policies. Unlike the policies that make simple permit or deny decisions, in practice, some policies make “strings attached” decisions [74]. That is, these decisions include additional actions that are obligated to be fulfilled. For instance, a bank policy that grants or denies fund-transfer activities not only renders a decision about a fund-transfer activity, but also includes an additional action to log the activity for auditing purposes. Another common example can be seen in most OS file-deletion policies that prevent accidental file deletion. In this case, a policy includes an obligated action asking a user to confirm a deletion request before rendering the decision to this request. Intuitively, these additional actions that should be fulfilled are called obligations [57, 56]. Policies that include obligations are obligation policies.

While obligation policies have been widely studied, the complexity of implementing the mechanism is also well known. One pressing security challenge, that arises when attempting to enforce multiple policies at the same time, involves resolving conflicts between one policy’s obligations and the constraints of other policies. To give an example, consider two policies $P_a$ and $P_b$, in which $P_a$ and $P_b$ respectively disallow all file downloading and pop-ups. Policy $P_a$ in addition defines an obligation that pops up warnings for any downloading attempt; this obligation violates $P_b$’s constraint of no pop-ups.
There are three approaches when it comes to resolving the aforementioned conflicts, and the first one is to ignore the conflicts and execute the obligations regardless. This approach argues that obligations, as the name suggests, are obliged to be executed. Taking on this argument, many works (e.g., [12, 23, 76, 7], etc.) will execute any triggered obligations. To ensure obligations will indeed be carried out, some works (e.g., [12, 23, 18, 44], etc.) track the executions of obligations and invoke compensatory actions for obligations that are failed to be fulfilled or fulfilled in time.

However, not all works that take the first approach will guarantee the executions of obligations. One such example is the OASIS standard for specifying access control policies — XACML [50]. XACML’s specification stipulates that obligations may be unable to be carried out if they cannot be understood by the XACML engine. This specification puzzles users because without defining for the term “understood,” XACML’s enforcement result can be unreliable since whether a given obligation can be carried out is unpredictable [40].

Evidently, the first approach could lead to security breaches since conflicts between obligations and policies are ignored. To illustration, let us continue with the above example policies $P_a$ and $P_b$. Theoretically, the policy $P$ that combines $P_a$ and $P_b$ via the conjunction logic would always disallow actions for file downloading and pop-ups. However, without any validation of $P_a$’s obligation, $P$ actually allows pop-ups in response to any actions that attempt to download files. This enforcement result actually violates $P$.

The second approach, conversely, acknowledges such conflicts and resolves them by not executing the parts of the obligations that violate security concerns. In practice, an obligation can be composed by a set of actions but not all of them may violate other policies. To comply with the obligations’ execution commitments while ensuring the executions do not violate any security concerns, works like Polymer [32] execute only the parts of an obligation that are not in conflict with other policies.
Although this approach can mitigate the vulnerabilities that might be present in obligations, it could result in undesired enforcement results. Obligations can be categorized into atomic and non-atomic obligations based on how they need to be executed. An atomic obligation requires that either all of its actions are executed or none of them are executed; in some cases, this atomicity is extended to this obligation’s associated decision. Executing only the parts of such an obligation violates its atomicity which can lead to an undesirable result.

The last approach, taking the viewpoint of each obligation being atomic, resolves such conflicts by not executing obligations that have one or more actions that violate security concerns. Controlling obligations in this way prevents unexpected behaviors and security breaches due to partially executed obligations or obligations that execute actions in violation of other policies. Current solutions for this approach is to rollback executed actions of obligations that violate security concerns (e.g., [16]) or to carry out actions to compensate the executed actions (e.g., [24]).

However, the current solution for the last approach can be impractical. The rollback solution requires the actions that have observable effects to be excluded from being specified in obligations. For this reason, effect actions, such as printing and pop-up actions, are ruled out for use in obligation (since a printed file cannot be un-printed or a pop-up cannot be un-popped). This restriction severely limits PSLs’ expressiveness in specifying obligation policies. Similarly, the compensatory approach is impractical because an effective compensation for security violations does not always exist. For instance, a blacklist may be able to prevent future data leakage, however, it is far from being enough to compensate for a leak of sensitive data.

Nonetheless, how to effectively enable policies to react to the obligations of other policies can also be challenging when attempting to enforce multiple policies at the same time. In order for a policy to achieve its desired security enforcement results, the policy may
need to react to certain actions that are performed by the target application, including the actions that are triggered by enforced policies' obligations. For instance, a policy that limits the number of files that can be open at the same time needs to maintain a correct count of the number of files currently open, regardless of whether the actions being executed to open and close files originated in policy obligations.

Aiming to bridge the above gaps, this dissertation proposes a policy specification language and enforcement system that allows composed obligations to maintain their atomicity and enables policies to interact meaningfully with the obligations of other policies.

Contributions

This dissertation presents PoCo (short for Policy Composition), a policy-specification language and enforcement system that allows for composition of policies with complex atomic obligations. Specifically, PoCo

- Allows for principled (i.e., with provable guarantees) composition of complex atomic obligations
- Supports specifying and enforcing pre-, post-, and ongoing-obligations
- Allows policies and their obligations to be effectful
- Allows policies with Turing-complete computation
- Enables resolving conflicts between a policy and obligations of other policies through static analysis
- Enable one policy to react to other policies’ obligations
- Supports custom policy-composition operators

As far as we are aware, PoCo is the first system to provide full support for atomic obligations, including conflict resolution and allowing policies to react to obligations. In addition to the design of the PoCo language and enforcement system, this dissertation presents
the language’s formal semantics and an analysis of type safety and other important language properties, as well as an implementation and evaluation of PoCo on two both desktop and mobile platforms.

This project is a collaboration with my partners Danielle Ferguson, Tyler Hanks, and Kevin Orr. Specifically, I mainly focused on designing the language, implementing the prototypes, and conducting two case studies. Thus, in this dissertation, I mainly focus on these parts of contributions to the PoCo project.

Organization The remainder of this dissertation is organized as follows:

- Chapter 2 discusses obligation and obligation policies,
- Chapter 3 discusses desired properties of PSLs for composing obligation policies,
- Chapter 4 analyzes most of the popular PSLs that support obligations based on obligation’s desired properties that are discussed in Chapter 3,
- Section 5 presents the design of PoCo’s enforcement mechanism,
- Chapter 6 summarizes the formal syntax and semantics that highlight the key features of the PoCo Language and enable formal type-safety reasoning,
- Chapter 7 presents the design of PoCo language’s policy structures,
- Chapter 8 explains how PoCo handles composition of policies, of useful properties about the PoCo language and architecture,
- Chapter 9 presents the implementation of the PoCo prototype to evaluate and refine its design,
- Chapter 10 discusses two case studies we conducted for demonstrating the expressiveness of the PoCo Language,
• Chapter 11 concludes and presents the future work.

Portions of the text presented in this dissertation is published in the International Conference on Software and Computer Applications [20]. Permissions to use the materials of these papers are in Appendix A.
Chapter 2: Obligation and Obligation Policies

Unlike the policies that make simple permit or deny decisions, in practice, some policies make “strings attached” decisions [74]. That is, these decisions include additional actions that are obligated to be fulfilled. For instance, a bank policy that grants or denies fund-transfer activities not only renders a decision about a fund-transfer activity but also includes an additional action to log the activity for auditing purposes. Another common example can be seen in most OS file-deletion policies that prevent accidental file deletion. In this case, a policy includes an obligated action asking a user to confirm a deletion request before rendering the decision to this request. Intuitively, these additional actions that should be fulfilled are called obligations [57, 56]. Policies that include obligations are obligation policies.

In the literature of policy specification and enforcement, there exist different definitions of obligations. Many pieces of literature consider only the actions that are obligated to be carried out after, not before, enforcing its associated decision to be obligations [74, 23, 33, 29, 45]; these types of obligations also be termed as post-obligations. For instance, a bank policy that renders a decision about a fund-transfer activity may include an obligation to log a fulfilled fund-transfer activity. Here, the logging action is a post-obligation because it needs to be carried out after this policy’s permission decision of a fund-transfer activity is carried out.

\footnote{Parts of this chapter is published in the International Conference on Software and Computer Applications[20]. Permissions to use the materials are provided in Appendix C.}
On the other hand, some pieces of literature also include that actions that need to be carried out before enforcing its associated decision as obligations [57, 56, 12, 50, 17, 51, 13]; these types of obligations also be termed as provisions or pre-obligations. For example, a file-downloading policy that includes two obligations, one warning end-users about suspicious files being downloaded and the other logging attempts to download suspicious files. In this case, both the warning and logging actions are pre-obligations because they need to be fulfilled before permitting a file download.

In addition, some pieces of literature also take in the actions that need to be carried out during its associated decision is enforced to be obligations, or ongoing-obligations [57, 56, 50, 51]. These types of obligations are implemented as help functions that run in the background to assist the triggers of the obligations. One example can be seen in a browser policy that filters out advertisements while users surf the internet. In this case, the action of filtering advertisements is an ongoing-obligation since it needs to be running while users browse the internet.

In this dissertation, the scope of the term obligations includes pre, ongoing, and post. One kind of obligation cannot always be substituted by another, supporting all three kinds of obligations riches policy expressiveness. For instance, the earlier example of a file-downloading policy includes an obligation to warn end-users about suspicious files being downloaded; this warning action needs to be fulfilled before, not during or after, allowing a download attempt. Likewise, the bank policy discussed earlier includes an obligation of logging a fulfilled fund-transfer; in order to generate correct data for auditing, this logging action should need to be fulfilled after, not before or during executing this transfer activity.

2.1 Pre-on-action and Pre-on-result Obligations

Although this categorization of obligations into pre-, post-, and ongoing- is standard, in practice all obligations are implemented as pre-obligations — post- and ongoing-
obligations are not strictly necessary. Security-relevant behavior of an untrusted application
can be either an action (e.g., a file-open action) or an action’s result (e.g., a file descriptor),
all obligations are either pre-obligations on security-relevant actions or pre-obligations on
security-relevant actions’ results or both.

2.1.1 Pre-on-action obligations

Pre-obligations are essentially pre-on-action obligations. By definition, a pre-type of
obligation is required to be fulfilled before a decision about a security-relevant behavior is
enforced. As we can see in Figure 2.1, these obligations are required to be fulfilled (at time
$t_1$) before a decision about the action is enforced at time $t_2$ (time $t_1 < t_2$). Thus, it is
more precise to categorize pre-obligations as pre-on-action obligations since they should be
fulfilled before a decision about a security-relevant action being enforced.

2.1.2 Pre-on-result obligations

Post-obligations are essentially pre-on-result obligations. Following the above discus-
sion, once a decision about a security-relevant action is carried out, an enforcement result
of this action is made. Afterwards, obligations that are categorized into post- need to be
carried out (at time $t_4$) before the result is returned to the target application (at time $t_5$);
this is because once an enforcement result is returned to a target application, the application
regains control and continues to execute. Since these obligations need to be fulfilled prior to
a result being returned (time $t_4 < t_5$), it is more precisely to categorize them as pre-on-result
obligations.

As to the obligations that should be carried out after an enforcement result is re-
turned, they are also pre-on-action obligations. Specifically, they are pre-on-action obliga-
tions to the next action of the target application. This is again because a policy enforcement
system loses control once an enforcement result is returned to a target application, and the
enforcement system will not be able to enforce any obligation until it seizes control again by detecting a new security-relevant action of the target application (at time $t_6$). In case no more actions left for a target application to execute after receiving an enforcement result, one such obligation can still be implemented as a pre-on-action obligation. That is, by monitoring the application’s exit action, one such obligation can be implemented as a pre-on-action obligation which will be carried out before allowing the application’s exit action to execute. One such example can be seen in Polymer [32]. As a runtime monitoring system, Polymer is able to detect a target Java application’s termination by monitoring all exit points in the target program’s control-flow graph.

In addition, obligations that are asynchronous to the triggers are special cases of pre-on-action obligations. Although normally being categorized as ongoing obligations, these types of obligations are carried out right before enforcing the decision about the triggers. As a result, in practice, these obligations are implemented with pre-on-action obligations. If necessary, such obligations also can be implemented by pairing pre-on-action with pre-on-result obligations, which are used to terminate the execution of the pre-on-action obligations right before enforcement results being returned. For instance, the browser policy mentioned
earlier can be specified by including both a pre-on-action obligation to start an advertisement filter before a user’s action to surf the internet and a pre-on-result obligation to terminate the execution of the filter after the user finishes browsing.

In short, all obligations are pre-obligations either on security-relevant actions or on the security-relevant actions’ results or both. Therefore, all obligations can be implemented as pre-obligations.

2.2 Other Different Categorizations

Besides the above categorization, obligations can also be characterized by their complexity, as well as how they should be carried out.

2.2.1 Different kinds of obligations based on their complexity

Obligations can be characterized as simple and complex based on their complexity. As the name suggests, a simple obligation is comprised of a single action while a complex obligation is comprised of more than one actions. For instance, the file-deletion policy mentioned in Chapter 1 includes a simple obligation which includes only one action — pops up a confirmation dialog before granting a file-deletion request. Likewise, in [32], IncomingEmail policy includes a complex obligation since it comprised of five actions.

2.2.2 Different kinds of obligations based on how they should be executed

Obligations can in addition be categorized into atomic and non-atomic obligations based on how they need to be executed. An atomic obligation is the one that requires either all of its actions are executed or none of them are executed; in some cases, this atomicity is extended to this obligation’s associated decision.

One such example can be seen in a policy that prevents accidental file deletion by issuing confirmation dialogs prior to granting the file-deletion request. In this example, if the
action of issuing a confirmation dialog is carried out, then the action’s associated decision needs to be followed. Otherwise, the policy may lead to an undesirable result such as when a deletion decision is not enforced after a user confirms the file-deletion request. Conversely, the policy will guarantee a confirmed file-deletion decision to be carried out if the issuing action and its associated decision are carried out atomically.

2.3 Summary

Obligation policies make “strings attached” decisions, that is, these decisions include additional actions that are obligated to be fulfilled. These additional actions are called obligations and they can be categorized in different ways. Based on their time of execution, obligations can be divided into \textit{pre-on-action} and \textit{pre-on-result} obligations. In addition, obligations can also be characterized as \textit{atomic} or \textit{non-atomic} (based on how they should be carried out), \textit{simple} and \textit{complex} (based on their complexity).
Chapter 3: Desired Properties of PSLs for Composing Obligation Policies

When policies are simple enough, as with classic safety properties, such composition is trivial because the only decision policies make is whether to permit or deny any given action; such decisions can be composed with algebraic or logical 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. The presence of these actions, referred to as obligations, makes it difficult to compose obligation-based policies.

It is challenging for PSLs to compose obligation policies while ensuring that the composition achieves desired security enforcement results. For obligation-based policies to be expressive and composable, an ideal general-purpose 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. These desired properties of PSLs are explained as follows:

3.1 Obligation-type Support

PSLs should be able to specify both pre-on-action and pre-on-result obligations, since one kind cannot always be substituted by another. For instance, the earlier example of a file-downloading policy includes a pre-on-action obligation to warn end-users about suspicious files being downloaded. This warning action cannot be defined as a pre-on-result obligation

---

1Parts of this chapter is published in the International Conference on Software and Computer Applications[20]. Permissions to use the materials are provided in Appendix C.
since it needs to be carried out before allowing a download attempt. Likewise, the bank policy discussed earlier includes a \textit{pre-on-result} obligation of logging a fulfilled fund-transfer; this logging action can not be defined as a \textit{pre-on-action} in order to generate correct data for auditing.

In addition, \textit{pre-on-action} and \textit{pre-on-result} obligations suffice for specifying arbitrary obligations. This is because, as discussed earlier, all obligations are pre-obligations either on security-relevant actions or on the security-relevant actions’ results or both.

\subsection{Atomic Obligations}

PSLs should also support \textit{atomic} obligations. An \textit{atomic} obligation demands that either all of its included actions are executed or none of them are executed; and, the atomicity can be extended to include the permit/deny decision that is associated with this obligation.

Obligation atomicity is essential for correctness in many real work practices. One such example can be seen in a policy that prevents accidental file deletion by issuing a confirmation dialog prior to granting the file-deletion request. In this example, if the action of issuing a confirmation dialog was executed and a user confirmed the confirmation request, then the user’s decision is obliged to be followed. Otherwise, the policy may lead to an undesirable result such as when a deletion decision is not enforced after a user confirms the file-deletion request. Conversely, the policy will guarantee a confirmed file-deletion decision to be carried out if the issuing action and its associated decision are carried out atomically.

\subsection{Obligations with Side Effects}

For many practical policies, an obligation may needs to include actions that are effectful. Effectful actions generally cannot be rolled back. For instance, the previous file-deletion policy’s obligation is comprised of an effectful action - issuing a confirmation dialog;
this action is effectful since an issued confirmation dialog cannot be rolled back. Related work [16] does support specifying and enforcing atomic obligations. However, it also demands obligations to be side-effect free; this limitation narrows its expressiveness significantly.

3.4 Complete Mediation of Obligations

The principle of complete mediation requires that all actions performed by the target application need to be inspected, regardless of where the actions originated [66]. When an obligation is issued in response to a security-relevant action, the actions included in this obligation should be validated. This is because one or more of these included actions may violate security concerns of other policies, and executing such an obligation will be a security violation.

3.5 Turing Completeness

In real work practices, security policies’ logic may need to include branching, looping, and variables. For instance, a policy that needs to validate files with unknown length line by line necessitates an iteration. Without supporting Turing complete, a PSL’s expressiveness can be limited since it may be unable to specify these practical policies. On the other hand, it is worth noting that a Turing-complete PSL cannot guarantee enforcement termination.

3.6 Conflict Resolution

When attempting to enforce multiple policies at the same time, one policy’s obligations may violate other policies, resulting in undesirable enforcement results.

When one policy’s obligation violates the rules of another policy, the resulting behavior can be inconsistent with the behavior of each policy in isolation. Again, consider two policies $P_a$ and $P_b$, which respectively disallow all file downloading and pop-ups. Policy $P_a$
in addition includes a *pre-on-action* warning obligation which violates $P_b$’s constraint of no pop-ups. Theoretically, the policy $P$ that combines $P_a$ and $P_b$ via conjunction logic would always disallow actions for file downloading and pop-ups. However, without handling the conflicts between $P_b$ and the obligation of $P_a$, the composed policy in fact allows pop-ups in response to any actions that attempt to connect to the internet. This enforcement result actually violates policy $P$.

Conflicts may also arise when a set of obligations does not execute in certain orders when the execution of one obligation causes the execution of another incorrect. In addition, given a set of obligations, the execution of one obligation may lead to the execution of another unnecessary. To resolve this type of conflict, it is necessary to include mechanisms to enforce the correct execution orders of a given set of obligations.

Besides, in order for a policy to achieve its desired security enforcement results, it may need to react to other policies’ obligation actions. To illustrate, consider a network policy that limits the number of connections that can be served at the same time. To correctly enforce the limiting constraint, this policy needs to count all open connections, regardless of whether the connections originated in policy obligations. This is because without taking into account the open or closed connections originating in policy obligations, the policy may not be able to maintain a correct number of open connections, leading to incorrect enforcement results. Malicious attackers could take advantage of this oversight to launch DDoS attacks, by establishing a large number of open connections within obligations thus circumventing the limitation on the number of network connections allowed.

### 3.7 Custom Composition Operators

From the usability and expressiveness point of view, PSLs would ideally support customized policy composition to enable users to compose policies that they want to specify. For instance, providing a commutative policy combinator could relieve policy composers’
concerns about the order in which a set of policies are composed since the order will not impact the composed result.

3.8 Summary

There are seven main desired properties of PSLs. Among them, the first six are essential because the first three enable obligation policies to be correctly specified, meanwhile the third and the fourth, and the fifth enable obligation policies to be correctly enforced while mitigating vulnerabilities that might be present in obligations. The seventh property, on the other hand, can facilitate users to specify policies that they want. These properties seem intuitive but can be hard to integrate into individual PSLs.

In the following section, we analyze the challenge via concrete PSL examples. Through this analysis, we not only experience the complexity of the obligation mechanism but also understand that currently there are no satisfactory solutions for implementing obligations.
Chapter 4: Related Works

In this chapter, we analyze most of the popular PSLs that support obligations based on obligation’s desired properties that are discussed in Chapter 3. Through this analysis, we not only experience the complexity of the obligation implementation but also understand that currently there are no satisfactory solutions for implementing obligations. So far, only one work (Polymer) is able to resolve conflicts as well as relevancies between policies and simple obligations. As far as we know, there exists no PSLs that support conflict and relevancy resolution between policies and complex obligations.

4.1 XACML

XACML [50], an OASIS standard for specifying access control policies, provides combining algorithms for policy composition. These algorithms highlight intuitiveness and simplicity, thus easing the processes for composing complex policies. For example, to prioritize a set of policies in XACML, policy composers can eschew assigning explicit priorities to each of these policies by simply applying the First-Applicable algorithm. XACML has been widely adopted and has been implemented into both commercial (e.g., Axiomatics [1]) and open-source software products (e.g., WSO2 [72]).

XACML supports specifying obligation policies. To simplify obligation specification, the XML-based standard includes a “FulfillOn” property in each obligation definition for specifying its execution condition. For instance, the policy Login in Figure 4.1 permits

\footnote{Parts of this chapter is published in the International Conference on Software and Computer Applications[20]. Permissions to use the materials are provided in Appendix C.}
legitimate users to login only during the working hours (from 9:00am to 5:30 p.m.). This policy also includes an obligation InvalidLogin to record login attempts that happened outside the working hours for auditing purposes. With its “FulfillOn” property set to “deny”, the InvalidLogin obligation is carried out whenever the policy denies a log-in request that happens outside the working hours.

```xml
<Policy PolicyId="Login" RuleCombiningAlgId=
"urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides">
  <Target/>
  <Rule RuleId="RuleLogin" Effect="Permit">
    <Target>
      <Actions><ActionMatch MatchId=
      "urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">login</AttributeValue>
      </ActionMatch></Actions>
    </Target>
    <Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">
      <Apply FunctionId=
      "urn:oasis:names:tc:xacml:1.0:function:time-greater-than-or-equal">
        <Apply FunctionId=
        "urn:oasis:names:tc:xacml:1.0:function:time-one-and-only">
          <EnvironmentAttributeSelector DataType="http://www.w3.org/2001/XMLSchema#time" AttributeId="urn:oasis:names:tc:xacml:1.0:environment:current-time">09:00:00</EnvironmentAttributeSelector>
        </Apply>
        <Apply FunctionId=
        "urn:oasis:names:tc:xacml:1.0:function:time-less-than-or-equal">
          <EnvironmentAttributeSelector DataType="http://www.w3.org/2001/XMLSchema#time" AttributeId="urn:oasis:names:tc:xacml:1.0:environment:current-time">17:30:00</EnvironmentAttributeSelector>
        </Apply>
      </Condition></Rule>
    <Obligations>
      <Obligation ObligationId="InvalidLogin" FulfillOn="Deny">
        <AttributeAssignment AttributeId="user" DataType=
        "http://www.w3.org/2001/XMLSchema#anyURI">
          urn:oasis:names:tc:xacml:1.0:subject:subject-id
        </AttributeAssignment>
        <AttributeAssignment AttributeId="LogAttempt" DataType=
        "http://www.w3.org/2001/XMLSchema#anyURI">
          urn:oasis:names:tc:xacml:1.0:action:action-id
        </AttributeAssignment>
      </Obligation>
    </Obligations>
</Policy>
```

**Figure 4.1:** The policy Login permits legitimate users to login only during the working hours (from 9:00am to 5:30 p.m.). It also includes an obligation InvalidLogin, which is carried out whenever the policy denies an invalid log-in request.
Yet, XACML does not support *pre-on-result* obligations nor support conflict resolution between policies and obligations. XACML supports only *pre-on-action* obligations because XACML’s obligations are always fulfilled *before* the decision is carried out. Nonetheless, XACML does not consider that some policies’ obligation may violate other policies. As such, XACML neither supports conflict resolution between policies and obligations nor supports its policies to react to the obligations of other policies.

In addition, the idea of atomic obligations is proposed but XACML does not include the details about how to specify or implement them. To guarantee that both the access decision and the obligation are successfully enforced, XACML specification states that the decision and the obligation need to be fulfilled jointly and atomically [51]. However, XACML fails to provide documentation that shows how to specify or implement atomic obligations. If users want to specify atomic obligations, they will have to comprehensively understand XACML’s implementation details first; due to a lack of detailed documentation, this process can be challenging. None of the XACML open-source implementations (e.g., [9], [2], [67], and [3]) that we’ve been able to find support atomic obligations.

XACML handles obligations in a complicated way. For instance, given a decision with attached obligations, XACML’s specification stipulates that the evaluation engine follows the decision only if the engine *understands* and can fulfill all these obligations [50]. That is to say, a denial (or, permitted) decision might be permitted (or, denied) when the decision’s obligation cannot be *understood* by the engine. Without defining for the term “understood”, XACML’s enforcement results can be unreliable since obligations may be ignored in some indeterminate circumstances [40]. This indeterminacy also results in XACML’s lack of algebraic combinators for policy composition.
4.2 ExtXACML

Li et al. [48] proposed ExtXACML, a system that extended XACML implementation to support obligation policies more effectively. Having seen XACML’s deficiency in specifying pre-on-result obligations, Li et al. augmented XACML implementation to support both pre-on-action and pre-on-result obligations. Additionally, the concept of obligation modules is introduced to resolve XACML’s indeterminacy problem in handling obligations.

As discussed in Section 4.1, whether a given obligation can be carried out is unpredictable so XACML’s enforcement result can be unreliable. To resolve this issue, in ExtXACML, the execution of obligations are handled by obligation modules. Specifically, an obligation module is a module that performs a set of application-specific actions such as notifying users or event logging. Once being implemented, an obligation module can be registered to the ExtXACML with its specific functional abilities. Thus, the ExtXACML monitor can determine which obligations are able to be “understood” based on the monitor’s registered obligation modules’ capabilities.

Specifically, at runtime, the ExtXACML monitor assigns obligations to its applicable obligation module based on the specific tasks the modules can manage. In such way, users are able to implement obligations to be carried out atomically. However, this atomicity cannot be extended to an atomic obligation’s associated decision. This is because obligation modules only take charge of obligation execution, and the decision enforcement is carried out by the ExtXACML monitor. Recall an earlier example policy that prevents accidental file deletion. If enforced in ExtXACML, the policy’s obligation will be fulfilled by an obligation module that can perform the task of issuing confirmation dialogs while the policy’s decision will be carried out by the ExtXACML system. Therefore, it is possible that a deletion decision is not enforced after a user confirms the file-deletion request.
More significantly, ExtXACML does not support conflict resolution between policies and obligations nor support policies to react to the obligations of other policies. Therefore, ExtXACML’s obligations can still lead to security breaches of the system when policies are composed. In other words, ExtXACML has not addressed the core issue of XACML obligation-policy composition.

4.3 Polymer

*Polymer* is an object-oriented PSL and a runtime monitoring system [32]. Polymer has a well-defined semantics that enables users to compose modularized runtime-security policies on untrusted Java programs. By separating an effect-free *query* method from an effectful *accept* method with in a policy, Polymer enables users to roll back a *query* method when its response is not adopted (there is no guarantee that a policy’s response will be followed). Polymer supports *pre-on-action*, *pre-on-result* and *atomic* obligations. Applied the principle of complete mediation, Polymer is able to resolve the conflicts between policies and *simple* obligations. The price of this application is that *complex* obligations that are unable to be executed atomically, ensuing conflicts between policies and *complex* obligations cannot be effectively resolved in Polymer.

As an expressive PSL, Polymer supports both *pre-on-action* and *pre-on-result* obligations. Specifically, Polymer policies issue different kinds of “suggestions” in response to security-relevant actions. Among these suggestions, an *InsSug* suggests that auxiliary code needs to be executed prior to making a final decision about a target action. Thus, a *pre-on-action* obligation can be directly defined by issuing an *InsSug*. In addition, an *OKSug* suggests permission to a trigger action and enables the action’s execution result to be monitored [32]. Therefore, by first issuing an *OKSug* to enable the monitoring of the action’s execution result, *pre-on-result* obligations can be specified when monitoring this action’s execution results.
Moreover, applying the principle of complete mediation [66], Polymer is able to resolve the conflicts between policies and obligations. The principle of complete mediation requires that all actions performed by the target application need to be inspected, regardless of where the actions originated. Thus, in Polymer, when an obligation is issued in response to a security-relevant action, the obligation’s actions will be validated against all enforced policies one by one just like any action that is originated by the target application itself. For each action of the obligation, an OkSug will be issued to it if executing it will not be a security violation. Conversely, if the execution of an action would result in a security violation, an ExnSug or an HaltSug will be issued to convey a denial suggestion (ExnSug in addition suggests the continue execution of the target application while HaltSug suggests termination). Additionally, Polymer policies are able to react to obligations of other policies since executed obligations’ results can be monitored.

Applying the principle of event-by-event complete mediation requires interruptible obligations; this interruption can result in an obligation and its associated decision will not be fulfilled jointly and atomically. To apply the principle of complete mediation, a Polymer monitor inspects all actions of the target application one at a time. Given an atomic obligation, the monitor will inspect all actions of the obligation one by one first before carrying out its associated decision. The execution of the obligation can get interrupted in two situations: i) there is at least one action that violates security concerns and will not be permitted to execute; or, ii) there is at least one action that triggers a Polymer monitor to issue another obligation prior to making a final decision about the original trigger action. Either situation would lead to an issued atomic obligation’s getting interrupted after some but not all actions of the obligation being executed, which can lead to an inconsistent enforcement of a Polymer policy.

Consider the earlier example: a file-downloading policy includes a complex and atomic obligation which first warns end-users about suspicious files being downloaded then logs
downloading attempts. If the logging action violates security concerns of other enforced policies or triggers another obligation to be issued prior to permitting the logging action, this obligation will get interrupted. This interruption may lead to an inconsistent enforcement, for example, a file is not downloaded after an end-user confirmed the file download attempt. Therefore, inconsistent enforcement results would lead to execution uncertainty of the target application, resulting in a weakened enforcement of Polymer monitor.

4.4 Ponder

Ponder (current version is Ponder2) is a declarative, object-oriented PSL that can be used to compose both access-control policies and general-purpose management policies [45, 46, 47, 30]. With Ponder, users can flexibly compose complex policies. Complex policies can be composed based on the logical relations between modular policies and the hierarchical relationships between subjects of modular policies.

However, in some cases, Ponder’s obligation policies may not be correctly enforced. This problem exists because, like Polymer, Ponder cannot resolve the conflicts between complex obligations and policies; and like XACML, Ponder fails to enable policies to react to the obligations of other policies. These problems are demonstrated below.

Firstly, Ponder supports pre-on-action but not pre-on-result obligations. In Ponder, obligation policies are specified in the format of “on triggering-events do obligated actions” [47]. For instance, the policy pOpen, as shown below, demands a logging obligation to be carried out in response to a denied file-open request.

```plaintext
type oblig pOpen (subject f, target user) {
    on deny_fileOpen(f, user);
    do log (f, user, file_open_attempt);
}
```
In this policy, the logging action will be fulfilled before carrying out a deny decision. Having been implemented in such way limits Ponder’s expressiveness in specifying pre-on-result obligations.

Secondly, although Ponder supports atomic obligations, conflicts between complex obligations and policies cannot be resolved. Ponder enables users to compose complex obligations by using its concurrency operators. Among them, both the operators → and && can be used to compose atomic obligations (the operator → restricts the execution order of grouped obligations while the operator && enables united obligation actions to be performed concurrently) [47]. For instance, the earlier example policy that warns users and logs information about suspicious file-downloading activities can be specified as below.

```plaintext
type oblig P_download (subject f, target u) {
  on isSuspiciousFile(f);
  do issueConfirmDialog(f) → log(f, u, file_download_attempt);
}
```

The obligation in this example is comprised of two actions: an issuing action and a logging action. The operator stipulates that the two actions need to be consecutively carried out in an atomic way. If either of the two actions violates any refrain policies (refrain policies are policies that specify a given subject must be refrained from doing certain actions on certain objects), the target application will halt.

However, like Polymer, a Ponder monitor inspects all actions of an obligation one at a time. As such, the execution of a complex obligation can get interrupted if its actions result in at least one security violation. This interruption can lead to an inconsistent enforcement of a Ponder policy. Additionally, Ponder fails to support policies to react to the obligations of other policies.

Finally, in Ponder, there are other difficulties in handling obligations. For instance, within a composed environment, the only allowed response to a conflict between an obligation and other policies is to halt the target application. This flexibility resolution limits
Ponder’s expressiveness. Given the above information, we can see that Ponder’s obligation implementation is inadequate.

4.5 SPL

Security Policy Language (SPL) is a PSL that enables users to engage policy combinators to compose complex authorization policies. In SPL, these authorization policies can be composed by uniting a set of different types of modular policies (e.g., mandatory access control and discretionary access control policies) using policy combinators [16, 15]. These combinators can also resolve the conflicts among decisions of composed policies. For instance, consider the two policies $P_a$ and $P_b$, in which $P_a$ always permits file-reading actions while $P_b$ always denies. A policy that combines $P_a$ and $P_b$ via the combinator “AND” will be able to resolve the conflict decision between $P_a$ and $P_b$ about a file-reading action by prioritizing denial decisions over permitted ones.

Mainly focusing on history-based policies, SPL supports pre-on-result but not pre-on-action obligations. History-based policies make their decisions based on actions that are executed in the past. In SPL, pre-on-result obligations are defined as constraints of the past events. For example, in the policy below, a logging obligation is defined as a future event $(fe)$ that needs to be carried out in the future after the execution of the current event $(ce)$ — file downloading action.

```
policy log {
  ?log: EXIST fe IN FutureEvents
    ce.action.name = "download" &
    ce.target = "files" &
    ::
    ce.author = fe.author &
    fe.action.name = "log" &
    fe.target.name = "logServer" &
    fe.parameters[0] = "files"
}  
```
SPL requires all obligations to be atomic, to ensure that future obligations are carried out. In cases when a policy’s obligation violates other enforced policies, an SPL monitor will reset the target application to the original state before the execution of the obligation’s trigger action. To realize this solution, real actions (i.e., actions that have observable effects) are excluded from being specified in obligation policies because these actions cannot be rolled back. For example, printing actions and pop-ups are ruled out for use in obligation policies since a printed file cannot be un-printed or a pop-up cannot be un-popped. Excluding these actions severely limits SPL’s expressiveness in specifying obligation policies.

Similar to Polymer, SPL’s obligation actions are validated one at a time at runtime, resulting in interruptible complex obligations. As demonstrated in Section 4.3, this interruption can lead to inconsistent enforcement results of enforced policies. Nonetheless, leaving out the possibilities that policies may need to react to the obligations of other policies, SPL’s obligation implementation can lead to security breaches of the target applications.

4.6 Heimdall

Heimdall, a platform for enforcing obligation policies, proposed the compensatory actions concept in response to execution failures of obligations [24]. Different from SPL’s roll-back mechanism for handling execution failures of obligations, Heimdall builds on the hypothesis that any executed action can be compensated by future actions. We argue that this implementation of using compensatory actions is impractical, even dangerous because there not always exists an effective compensation for security violations. For instance, a blacklist may be able to prevent future data leakage, however, it is far from being enough to compensate a leak of sensitive data.

More significantly, Heimdall does not support conflict resolution between policies and obligations. Specifically, if a target application’s executed action triggers any obligations, a Heimdall system first creates instances of these obligations. The Heimdall system then
sends the execution request of these obligation instances **directly** to the underlying execution system. That is to say, obligations are not validated against enforced policies before execution. Afterwards, if an obligation is fulfilled, the underlying execution system sends the information about this action to the Heimdall system, which then deletes the corresponding obligation instance. In cases when an obligation is not fulfilled, Heimdall will request the underlying execution system to execute the compensatory action of the obligation [24]. As such, enforced Heimdall policies cannot react to the executed obligations. Therefore, just like XACML, Heimdall’s obligation policies can cause security breaches of the system when policies are composed.

The work [29] proposed a formal meta-model for obligation management. Viewing obligations as contracts between systems and subjects, [29] mainly focused on the obligations that their executions will not interrupt (or at least not immediately) the execution of the target application. In this context, the authors formally defined a set of conditions to check accountability of the obligations.

In [29], the authors believe that it is policy writers’ responsibility to ensure that conflicts do not raise between obligations and other policies. The authors argued that obligations should only be assigned to subjects that are able to carry out them; this argument requires policy writers to only specify obligations that will not violate other specified policies. This is impractical in today’s intricate security environment.

### 4.7 Rei

Rei, modeled on deontic concept of permissions, prohibitions, obligations and dispensations, is a non-domain specific language that supports specifying **pre-on-result** obligations [33]. In Rei, a policy is comprised of a set of rules, in which each rule is comprised of an entity and a **policy object**. A **policy object** can specify allowed or prohibited actions, and an entity entitles the specified permissions or prohibitions when associated with such
a policy object. A *policy object* also can specify obligations, and an entity is obligated to perform the specified obligations when associated with such a policy object.

Rei recognizes the conflicts between obligation and prohibition policies and offers two ways to resolve such conflicts. The first way is to specify priorities among policies and/or policies rules and the second way is to set negative/positive-modality precedence on actions, entities and policies. For example, given a policy that prohibits a user from printing, an obligation policy that stipulates the user to print will cause a conflict — the user does not have the permission to perform the obligated printing action. To resolve this conflict, policy writers can assign different priorities to them: the user will be able to carry out the obligation if the obligation policy is given priority over the prohibition policy, and vice versa. Apart from assigning priorities, this conflict also can be resolved by setting negative/positive-modality precedence: if the user’s modality precedence has been set as negative, then he/she will not be able to carry out the printing obligation, and vice versa.

Yet, it is unclear how conflicts are handled between complex obligations and prohibition policies. As mentioned before, a complex obligation is comprised of a set of actions. Given a complex obligation, if some, but not all, actions of it violate specified prohibition policies, it is unclear that whether this obligation will not be carried out or only those violated actions will not be carried out. If it is latter, then the atomicity of an atomic obligation will not be able to be preserved.

Nonetheless, it is also unclear that whether Rei is able to react to obligations of other policies. In [33], how obligations’ executions are enforced and monitored was not mentioned. If Rei policy engine only concerns about whether an obligation is carried out and does not enable other policies to react to the obligation, then the policy engine may not be able to guarantee some policies’ desired security enforcement results.
Table 4.1: Summary of the extent to which existing PSLs satisfy the goals listed in Chapter 3.

<table>
<thead>
<tr>
<th></th>
<th>Supports obligations that are</th>
<th>Supports resolving conflicts for obligations that are</th>
<th>Supports reacting to obligations that are</th>
<th>supports custom composition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pre-</td>
<td>post-</td>
<td>effectful</td>
<td>Turing</td>
</tr>
<tr>
<td>XACML</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>XACML Extensions</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Polymer</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Ponder</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SPL</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Heimdall</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Rei</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Aspect Oriented</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

From the table, we see that currently there are no satisfactory solutions that provide support for atomic obligations, including conflict resolution and allowing a policy to react to atomic obligations of other policies.

### 4.8 Aspect Oriented Programming

Designed for separating concerns of a program from its functionality implementations, *Aspect-oriented programming (AOP)* is commonly used by policy writers to specify security policies [31]. One such example is AspectJ.

AspectJ is an extension of the Java programming language that supports AOP. It is not solely designed for security policy-specification. Instead, AspectJ’s unique programming model makes the language amenable to security policy-related tasks.

The central idea of AspectJ is that it provides a way to cleanly organize and compose classes of actions that belong outside of the traditional Java inheritance hierarchy of a program. These Aspects, as they are called in AspectJ, crosscut various parts of the inheritance hierarchy. This crosscutting is achieved by the use of pointcuts. Pointcuts specify where an Aspect’s methods will take effect. Any method will specify the pointcuts that trigger it. These pointcuts can have varied uses, but the most applicable to policy-specification are method signature pointcuts. Using method names built with wildcards and variable-length
argument lists, an AspectJ pointcut can specify that an action is performed when methods matching the signature are executed [73].

AspectJ can be incorporated into a program at compile-time, or it can instrument Java bytecode programs. This feature combined with the pointcuts described above makes AspectJ a very compelling policy-specification language. A direct consequence of AspectJ’s tight integration with the Java platform means that it has the same potential for arbitrary logic. However, AspectJ does not enforce any sort of interface on its code, allowing its aspects to be constructed in arbitrary ways [73].

4.9 Summary

Upon studying, we summarized the aforementioned solutions in Table 1 based on the five main desired properties of PSLs discussed in Chapter 3. As shown in the table, so far there is only one work, Polymer, which is able to resolve both the conflicts and the relevances between policies and atomic obligations. Nonetheless, as far as we know, there exists no PSLs that can support conflict resolution between policies and atomic obligations.

Aiming to bridge this gap, we proposed a policy specification language and enforcement system, PoCo, to satisfy all the desired properties. Specifically, PoCo supports both pre-on-action and both pre-on-result obligations; PoCo can resolve conflicts between atomic obligations and policies as well as between atomic obligations and policies; PoCo also enables policies to react to atomic obligations of other policies. In short, PoCo allows composed obligations to maintain their atomicity and enables policies to interact meaningfully with the obligations of other policies.

In the following four chapters, we will explain our solution PoCo in detail regarding the design of the PoCo language, the enforcement mechanism, as well as the prototype implementation and two case studies.
Chapter 5: The PoCo Monitor Architecture

PoCo’s enforcement mechanism operates as a monitor that has the ability to observe a target application’s security-relevant actions (e.g., system or method calls) and the results of these actions, as shown in Figure A.1. Hence, actions and their results can trigger the monitor to respond, with the response depending on the logic of the policies being enforced.

5.1 Monitor Operation

The PoCo monitor observes all security-relevant events—actions and results—and broadcasts each event to every policy being enforced. Each policy inputs the current trigger event $e$ (i.e., the security-relevant event triggering policy enforcement) and suggests an obligation to be executed before $e$ is processed. This obligation, which may be empty, can implement supplemental logic or alter the input event to meet the policy’s goals.

The PoCo system infers a listing of security-relevant events from the logic of the policies being enforced: any event named within any policy is considered security relevant. Policy authors can always refine this listing of security-relevant events. By ensuring that PoCo’s has an accurate listing of security-relevant events, policy authors ensure that the PoCo monitor only notifies policies about (i.e., broadcasts) events that are 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

\[1\] Parts of this chapter is published in the International Conference on Software and Computer Applications[20]. Permissions to use the materials are provided in Appendix C.
cannot execute additional obligations until receiving a new event. Essentially, the monitor operates in a loop, with each iteration performing the following five steps:

1. Input security-relevant action or result \( e \)

2. Query all enforced policies and collect candidate obligations

3. Prioritize the collected candidate obligations and then,
   
   (a) Pop the first obligation \( o \)

   (b) Query all the enforced policies and collect votes by processing policies’ \( votes \) on \( o \)

   (c) Combine the collected votes into a single permit/deny decision to whether permit or deny the execution of \( o \)

   (d) Execute \( o \) if it is permitted to execute and then collect obligations triggered in response to \( o \)

   (e) Discard \( o \) if it is denied for execution

4. If a new output action has been set, execute it, if a new output result has been set return it

5. Otherwise, execution the original input action or output the original output result

In response to every security-relevant event, PoCo monitor iterates the above five steps until obtain a valid result that can be returned to the application being monitored. With this design, PoCo ensures that obligations that violate security concerns of other policies will not be executed. The pseudo-code for the monitor algorithm is presented in Appendix A.
5.2 Monitor Configuration

It is necessary to understand PoCo monitor configuration before discussing PoCo policies in detail. Thus, in the text below a brief description of PoCo monitor configuration is given and a more detailed discussion appears in Section 8.

Three elements can be supplied to the monitor when specifying a composed policy. The first is a list of policies to be enforced. These base policies are the building blocks used to construct the composed policy.

The Vote Combinator or VC is the second parameter and is responsible for combining the policies’ votes into a permit/deny decision for each obligation that wants to execute. Because the policy author may wish to combine votes based on information such as policy name, the Vote object was introduced to hold a boolean vote as well as relevant information about the policy that generated it. The VC’s single method, called evaluate, takes a list of Vote objects and returns a boolean decision, thus making the VC’s type $\text{Vote}[] \rightarrow \text{bool}$. PoCo’s default vote combinator is conjunctive and only permits obligations that are permitted by all base policies. PoCo users can alternatively define and apply custom vote combinators.

The Obligation Scheduler or OS is the last parameter and is used to prioritize input policies based on specified criteria. The OS deals with conflicts that arise between obligations when the execution of one obligation would render the execution of another meaningless or detrimental. To resolve this type of conflict, the OS includes a single method named prioritize to schedule the execution of obligations. This scheduling is crucial because it determines the final output event for a given input event; the highest priority policy to call setOutput will be the one whose output event is returned to the application or system.

Hence, the PoCo monitor can be viewed as a policy scheduler. The monitor decides which obligations to execute and in what order. The monitor’s parameters allow this scheduling to be customized.
5.3 Obligations

Many definitions of obligation have been given in the literature on policy specification and enforcement [32, 46, 16, 74, 56, 57, 13, 29]. There are subtle differences among these works. However, in essential, an obligation is one or more actions required to execute under certain circumstances with specific timing in response to the security-relevant events originated from the target applications.

There are three possible solutions when there are conflicts between obligations and policies. The first solution is to evade the conflict and do not execute the obligation at all; however, ignoring obligations may lead to a violation of an organization’s security code itself due to some intended security actions not being performed. The second possible solution is to ignore the conflict and execute the offending obligation; ignoring conflicts can lead to a security violation. The third possible solution is to only execute the parts of the obligation that are not in conflict with other policies; although this option cannot preventing insecure behaviors that can arise from partially executed obligations or obligations that execute actions in violation of other policies.

Figure 5.1: Obligations are either pre-on-action, which are fulfilled before a decision on an action is enforced, or pre-on-result, which are fulfilled before a decision is made to return a result.
Aiming to dynamically resolve conflicts between policies and atomic obligations, we choose the first solution, that is to say, PoCo is designed to not execute conflicting obligations. Other works have referred to this definition of obligations as “suggestions” since they are not guaranteed to execute [32]. However, even XACML—which does not provide conflict resolution among obligations—suffers from non-guaranteed obligation execution. XACML’s obligations are intended to narrow the gap between the practical functional requirements of an organization and policy enforcement via access control. However, obligations may be ignored in two situations: i) Only the policies/policy sets whose results are the same as the root policy will have their obligations propagated [50]; ii) Some policy combining algorithms (i.e. First-applicable, Deny-overrides and Permit-overrides) may not evaluate all sub-policies, and all those unevaluated policies’ obligations will be ignored. Due to the obligation propagation’s indeterminacy, some implementations of the standard (e.g. WSO2 Identity Server[72]) omit sub policies’ obligation propagation altogether. However, ignoring obligations may lead to a violation of an organization’s security code itself due to some intended security actions not being performed.

5.4 Complete Mediation of Policies with Atomic Obligations

*Complete mediation* demands that all actions performed by the target application need to be inspected, including the actions that are originated in obligations. To precisely inspect each security-relevant event at run-time, most approaches choose to inspect one event at a time, and we refer to this design as *event-by-event complete mediation*.

However, such approaches may be unable to handle conflicts and reactions among obligations while maintaining the obligations’ atomicity. This is because unless all actions and their parameters included in an obligation can be statically determined, it is impossible to have both event-by-event complete mediation and atomic obligations.
Thus, aiming to handle conflicts and reactions among obligations while maintaining the obligations’ atomicity, PoCo enforces *obligation-by-obligation complete mediation* by analyzing obligations before executing them. Specifically, policies vote on candidate obligations based on their statically generated CFGs. These CFGs are conservative approximations since computing the exact CFG for an arbitrary program is undecidable.

### 5.5 Non-termination of Policy Enforcement

This design prioritizes policy expressiveness over guaranteed enforcement termination, thus PoCo is designed to be Turing complete. Thus, PoCo supports specifying policies with branching, looping and variables (Chapter 6); this design introduces possible non-termination in the enforcement code. For instance, a policy may include an obligation with infinite loops.

Besides, when policies are able to react to each other, a set of policies may create an infinite sequence of obligations in response to each other’s obligations thereby generate a non-termination path. This non-termination cannot be statically detected in general. For example, given two policies in which one monitors all network connections and logs them to a file while another policy monitors all file writes and opens a new network connection on each, to log the file write in a database, a non-termination path will be generated including an infinite loop that contains a log and a network open actions. This non-termination cannot be statically detected in general.
Chapter 6: Formal Syntax and Semantics of PoCo language

To highlight the key features of the PoCo Language, enable formal type-safety reasoning, and demonstrate desired properties of obligations, we have defined a precise syntax and a formal semantics for PoCo language. Specifically, PoCo is formalized as a functional language due to the inherently simpler specification compared to object-oriented languages such as Java. The dynamic semantics, which includes all of the core features of the PoCo Language, is to express the workings of these features in a precise and unambiguous manner.

Using these semantics, we have proven that PoCo language is type safe via standard type-preservation and progress lemmas. We have in addition proven that the PoCo Language supports atomic obligations, conflict resolution between policies with obligations, and allows all policies to react to the obligations of other policies.

6.1 Syntax

Figure 6.1 lists the syntactic elements of PoCo. Aside from basic common types (e.g., boolean and string), we introduced five syntactic sugar to simplify semantic presentation. Among them, Act and Res specify security-relevant actions and results respectively, and Event is a sum type that can represent either an Act or a Res. Obligation is also a sum type that holds a function-typed value that takes in either a Event type to depict onTriggers or a ResList onObligations. To depict the CFG representation of an obligation, CFG type is 3-tuples consisting of nodes, edges, and a reference to the obligation the CFG represents.

\footnote{Parts of this chapter is published in the International Conference on Software and Computer Applications [20]. Permissions to use the materials are provided in Appendix C.}
We then added introduction and elimination expressions for the five types. It is worth mentioning that, the two expressions: \texttt{call}(e_1, e_2) and \texttt{invoke}(e_1, e_2) are introduced for function invocation. Designed to call functions within applications and policies, \texttt{call}(e_1, e_2) takes two parameters where the first is corresponding to the function to be called and the second is the argument for the function. For instance, a policy can use the expression \texttt{call(log, fileOpen)} to log every file-open action. On the other hand, the expression \texttt{invoke}(e_1, e_2) is used for a PoCo monitor to execute valid actions that are output from

\textbf{Figure 6.1: Formal syntax for PoCo}
the monitor. Thus, the first parameter is a \textit{String} type which specifies a valid function name that is security relevant.

Based on these types and expressions, a PoCo policy is defined as a function. For better customization, the \texttt{Pol} function takes an arbitrary type parameter. To specify a policy’s three components (i.e., \texttt{onTrigger}, \texttt{vote}, and \texttt{onObligation}), the \texttt{Pol} function returns a tuple of three functions: $e_{ot}$, $e_{vote}$, and $e_{oo}$ to respectively represent the three components. Let’s consider the policy $P_{\text{dis}}$, a policy that disallows a specific action that is specified in the argument, for example. As shown below, to disallow a specific action, the policy takes an \texttt{Act}-type parameter. To prohibit a specific action from execution, the policy returns a 3-tuples with $e_{ot}$ and $e_{vote}$ elements to disallow the action that is originated from target application and from other enforced policies respectively. In addition, without the need to react to obligations of other policies, the $e_{oo}$ element is defined as a empty function.

For example, The $P_{\text{dis}}$ policy which disallows an action specified by the policy argument can be specified in the formal syntax as seen below. Additional example policies are listed in Appendix B.

```pseudocode
fun disallow (x:Act):Pol = 
  name = dis_x,
  onTrigger = (fun ot(e:Event):Unit =
    case e of
    act a =>
      if a.name == x.name \ a.arg == x.arg
      then setOutput(event(act("exit",makeTypedVal(Unit,unit))))
      else unit
    | res r => unit),
  onObligation = (fun oo(rt: ResList):Unit = unit),
  vote = fun vt(cfg:CFG):Bool =
    -call(containsAct,cfg=cfg, name=x.name,
      arg=(in_arg(x.arg)):TypedVal + none:Unit),count=1)
)
```

To facilitate proving the obligation property, we included two \texttt{Label} elements: $begin_f$ and $end_f$. These two elements is only used for mark the beginning and end of a function’s ex-
ecution as well as parameters and return values; these elements have no effect on a program’s execution.

6.2 Static Semantics

Figure 6.2 presents some of the static-semantics rules of the PoCo language. These typing rules are straightforward to understand and reason about. In the main judgement, $\Lambda, \Gamma, F \vdash e : \tau$, the context $\Gamma$ maps variables to their types while $F$ maps references to monitored functions. The rules $\text{eventAct}$ and $\text{eventRes}$ illustrate that an Event type can hold either an Act or a Res type. The rule $\text{makeCFG}$ specifies the typing semantics of the CFG type. For function calls within target applications and policies, the rule $\text{call}$ specifies that the first parameter is a function type and the second is the type of this function’s parameter. Different from $\text{call}$, the rule $\text{invoke}$ is used for a PoCo monitor to execute valid actions that are output from the monitor, thus it requires the first parameter to be a String type which specifies a valid function name. Lastly, since $\text{label}$ elements are merely used for marking start and end points, they will not change the types of expressions.

$$
\Lambda, \Gamma \vdash e : \tau
$$

$$
\frac{}{\Lambda, \Gamma \vdash e : \text{Obligation}} \quad (\text{makeCFG})
$$

$$
\frac{}{\Lambda, \Gamma \vdash \text{makeCFG}(e) : \text{CFG}}
$$

$$
\frac{\Lambda, \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2}{\Lambda, \Gamma \vdash \text{call}(e_1, e_2) : \tau_2} \quad (\text{call})
$$

$$
\frac{\Lambda, \Gamma \vdash e_1 : \text{String}}{\Lambda, \Gamma \vdash \text{invoke}(e_1, e_2) : \text{TypedVal}} \quad (\text{invoke})
$$

$$
\frac{\Lambda, \Gamma \vdash e : \tau}{\Lambda, \Gamma \vdash \{ e \}_{s(v)} : \tau} \quad (\text{endLabel})
$$

Figure 6.2: Static semantics (rules for actions, results, events, CFGs, labels, and function calls).
6.3 Dynamic Semantics

To express the runtime behavior of monitored applications, PoCo’s dynamic semantics are defined using small-step operational semantics with a left-to-right, call-by-value evaluation order.

One interesting part of PoCo’s dynamic semantics is the set of rules for function calls. As shown in Figure 6.3, five rules are included to handle function calls. Aside from all adding a begin and an end labels to an execution trace before and after the evaluation, the five rules are used to determine the behavior of a function call based on different situations. Specifically, 1) for an onTrigger or onObligation function call, the rule callOb resets the current result trace before evaluation; 2) for a security-relevant function call (i.e. included in the list of monitored functions, F) that is originated from an obligation, the rule callFromObligation will append the evaluation result of the call to the current result trace; 3) for a security-relevant function call that is originated from the target application, the rule callFromApplication will evaluate the call to the final PoCo monitor response to it; 4) for the monitor function to be called on a security relevant event, the rule callMonitor sets the flag inOb to true before evaluation to indicate the current executing context; 5) for a security-irrelevant function call, callNonMonitoredFunction directly evaluates the call and attaches a begin and an end labels to a function call’s execution trace.

6.4 PoCo Language Properties

Using these semantics, we have proven that PoCo language is type safe via standardtype-preservation and progress lemmas. We have in addition proven that the PoCo Language supports atomic obligations, conflict resolution between policies with obligations, and allows all policies to react to the obligations of other policies.
6.4.1 Type Safety

The PoCo language is type safe via the standard Preservation and Progress Lemmas [70]. Type safety guarantees that well-typed PoCo programs will never get stuck (i.e., well-typed expressions are either values or can be further evaluated). The theorem is stated below. The proof of type safety appears in the companion technical report [19].

**Theorem 1** (Type-safety).

\[(C, e) : \tau \land (C, e) \rightarrow^* (C', e') \Rightarrow (C', e') : \tau \land (\exists v : e' = v \lor \exists C''', e''' : (C', e') \rightarrow (C''', e''')).\]

6.4.2 Atomicity of Obligations

We have proven that all obligations in PoCo are executed atomically—once an obligation begins executing, no other obligation code executes until that obligation has finished executing. This does not guarantee that the executing obligation will terminate. We have
also proven that PoCo supports specifying and enforcing pre-, post-, and ongoing- obligations by making use of both pre-on-action and pre-on-result obligations.

6.4.3 Conflict Resolution

We have proven that PoCo resolves conflicts between policies and obligations by allowing each policy to vote to approve or deny each obligation immediately prior to its execution. This vote is guaranteed to be provided as input to the vote combinator which may or may not use the value to determine the final vote. Since this vote combinator is specified by the policy architect, policies have as little or as much decision-making power as is desired.

6.4.4 Relevance Resolution

We have proven that PoCo enables policies to react to obligations of other policies by allowing each policy to propose a new obligation in response to an executed obligation that contains security-relevant events. This property shows that after each obligation containing security-relevant events ends, a CFG is created based on querying the onObligation function of each policy.

6.4.5 Policy Permutability

We have proven that it is possible to design a PoCo monitor (i.e., VC and OS pair) such that the order in which the policies are declared does not affect the outcome. This is a desirable feature because it allows for true modularity of policies and makes it simpler to test sets of policies in isolation. In order to prove this, we must first define what it means for the outcome to be unaffected. In general terms, this means that regardless of the order that policies are input, identical obligations should be executed in the same order, and the same output event should be decided.
6.5 Summary

This chapter summarizes the formal syntax and semantics that highlight the key features of the PoCo Language and enable formal type-safety reasoning. The primary purpose of these semantics, which includes all of the core features of the PoCo Language, is to express the workings of these features in a precise and unambiguous manner. 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 is proven to be type safe through standard type-preservation and progress lemmas.
PoCo policies are designed to be granular pieces of logic that execute obligations based on security-relevant input events. Each security-relevant action the target application attempts to execute, and each security-relevant result the underlying system attempts to return, is broadcast to all the policies registered with the monitor.

Designed to provide the safe executions of policy obligations while maintaining their atomicity, a PoCo policy includes components to prevent the execution of obligations that could violate security concerns as well as react to the executed obligations. Specifically, a PoCo runtime policy is comprised of three components: onTrigger, vote, and onObligation. Among these components, the onTrigger specifies a desired response to a security-relevant event’s execution request; the response may include obligations or, more precisely, pre-on-action obligations. The vote component ensures that only obligations that do not violate security concerns will be executed. In reacting to the executions of permitted obligations, the onObligation component specifies the desired responses, namely pre-on-result obligations, that need to be carried out upon the executions.

In the following sections, we discuss these three components in detail to show how PoCo allows executed obligations to maintain their atomicity and enable policies to interact meaningfully with the obligations of other policies. To illustrate the core features of PoCo policies, we use six simple, but representative, example policies to serve as running examples throughout this chapter:

- $P_{file}$ disallows users from opening the secret.txt file

\footnote{Parts of this chapter is published in the International Conference on Software and Computer Applications\cite{20}. Permissions to use the materials are provided in Appendix C.}
• \( P_{\text{dis}}(\text{Action } d) \) disallows the specified action \( d \)

• \( P_{\text{postlog}} \) requires every file-open action to be logged after the action has occurred

• \( P_{\text{prelog}} \) requires every file-open action to be logged before the action has occurred

• \( P_{\text{confirm}} \) requires each file-open action attempted by the target to be confirmed through a pop-up window

• \( P_{\text{time}} \) disallows popups unless at least 100 seconds have passed since the last popup

7.1 The onTrigger component

The first component of a PoCo policy, called \texttt{onTrigger}, is a function that responds to input events by specifying an obligation to execute before the input event. These obligations are atomic; every \texttt{onTrigger} method implicitly obtains a mutex lock that gets released when the method finishes.

Given a trigger event, \texttt{onTrigger} defines an obligation that may specify an \textit{output event}, which is the final PoCo response to the input (trigger) event. Ultimately, PoCo must relinquish control to enable the application or system to continue executing. If no policy specifies an output event, the PoCo monitor will cede control by allowing the trigger event to be executed, that is, by outputting the input event.

For example, \( P_{\text{file}} \)'s \texttt{onTrigger} examines the trigger event \( e \). If \( e \) is \texttt{fopen(secret.txt)} then \( P_{\text{file}} \)'s \texttt{onTrigger} sets \texttt{exit} as the output event, meaning that the monitor should cede control to the system to execute the \texttt{exit} action. \( P_{\text{file}} \) does not specify an output event when \( e \) is not \texttt{fopen(secret.txt)}, thus allowing irrelevant events to execute normally. No additional events are executed prior to the monitor ceding control. Hence, \( P_{\text{file}} \)'s \texttt{onTrigger} is defined as follows.
Output events must be treated specially because the monitoring system must reach agreement in how the transfer of control occurs. Therefore, one of the primary objectives of any system for composing runtime policies must be to determine the singular output event for each trigger event. Output events that are actions cede control to the underlying system, and output events that are results cede control to the application. Prior work has defined models of monitors that operate in this way, interposing between applications and executing systems and responding to trigger events with output events [36].

As seen in $P_{file}$’s onTrigger, changing the output event is accomplished with the setOutput method. Calling this method commits the monitor to using that event as the output event. Once setOutput has been called for a given trigger event, additional calls by any policy, for the same trigger event, result in an error state indicating that the output event cannot be overwritten. To avoid this error, a policy may first call getOutput or outputNotSet to confirm that an output has not yet been set; policy logic may then determine what happens if the output event has already been set.

$P_{confirm}$’s onTrigger method tests whether the trigger event, $e$, is a file-open action. If it is and no output event has been set, then onTrigger specifies an obligation to confirm $e$. Based on the result of the confirmation, onTrigger sets the output event to $e$ (indicating that the file open must be executed) or nil (indicating that an empty result must be returned to the application in lieu of opening the file). If $e$ is not a file-open action, or an output event is already set, $P_{confirm}$ inserts no additional logic. Hence, $P_{confirm}$’s onTrigger is defined as follows.

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

P_{confirm}'s onTrigger method tests whether the trigger event, e, is a file-open action. If it is and no output event has been set, then onTrigger specifies an obligation to confirm e. Based on the result of the confirmation, onTrigger sets the output event to e (indicating that the file open must be executed) or nil (indicating that an empty result must be returned to the application in lieu of opening the file). If e is not a file-open action, or an output event is already set, P_{confirm} inserts no additional logic. Hence, P_{confirm}'s onTrigger is defined as follows.
The ability to permanently set the output event is required for $P_{\text{confirm}}$’s correctness. If it were possible for the output event not to execute due to other policies’ obligations, then the user could opt to allow the file open before the monitor chooses not to allow it, or the user could opt to disallow the file open before the monitor executes it anyway. This level of control also enables policies to self-manage in instances where they conflict with other policies.

### 7.2 The vote Policy Method

The second component of a PoCo policy is a method, called `vote`, that takes an obligation $o$ and returns a vote indicating approval or disapproval of $o$. To enable the `vote` method to analyze obligations, PoCo represents them as Control Flow Graphs (CFGs). Hence, the `vote` method takes the CFG of an obligation being considered for execution and returns a boolean vote. For example, the goal of $P_{\text{file}}$ is to prevent the secret.txt file from being opened, even by other policies’ obligations. Therefore, when examining an obligation, $P_{\text{file}}$ looks for fopen(secret.txt) in the obligation’s CFG. If $P_{\text{file}}$ finds that action, it votes to disallow the obligation. Otherwise, $P_{\text{file}}$ votes to allow it. Hence $P_{\text{file}}$’s vote is:

```plaintext
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, PoCo policies analyze obligations before they execute—specifically, policies vote on candidate obligations based on their statically-generated 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 of actions invoked in obligations, it is necessary to allow unresolved arguments, which are parameters to a security-relevant action that could not be determined statically. The CFG of an obligation defines unresolved arguments as such, and policies may specify how to handle unresolved arguments. For example, $P_{\text{file}}$ conservatively votes against obligations known to open the secret.txt file and also obligations containing file opens with unresolved arguments. $P_{\text{file}}$’s vote is therefore:

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

7.3 The onObligation Policy Method

The third component of a policy is an obligation that may be executed in response to other obligations, in order to inject additional actions after the triggering obligations. This is necessary to achieve the goal of policies reacting to other policies’ obligations. The onObligation method responds to obligations by analyzing the results of all security-relevant actions performed during an obligation’s execution, that is, a result trace (rt). For example, $P_{\text{postlog}}$ proposes an obligation that logs each file open in another obligation. This new obligation is specified in the policy’s onObligation as follows:
PoCo cannot insert obligations before the execution of a triggering obligation because doing so may create inconsistency in the execution. Prior to executing an obligation $o_1$, the monitor decides whether $o_1$ should be executed. Execution of another obligation, $o_2$, prior to $o_1$ may cause policies to vote differently than they did originally, when deciding to permit $o_1$. If PoCo were designed to re-query policies after $o_2$ was inserted, and the new decision was to not execute $o_1$, it is possible that $o_2$ should not have been proposed in the first place. To have reliable behavior, the voting on and execution of a given obligation must therefore be treated as an atomic unit. For this reason it is not possible in $P_{prelog}$ to log events in obligations prior to their execution, though it is possible to do so in $onTrigger$.

To summarize, there are two ways PoCo policies specify obligations: $onTrigger$ specifies an obligation in response to a trigger event, and $onObligation$ specifies an obligation in response to other obligations (which may be defined by $onTrigger$ or $onObligation$). The $vote$ method enables policies to indicate approval or disapproval of obligations.

### 7.4 Parameterized Policies

To aid code reuse, policies may be parameterized. For example, there are many policies that might disallow one particular event. PoCo therefore allows policies to abstract over common patterns. For example, $P_{dis}$ is the generalized version of $P_{file}$. This policy has a parameter, $d$, that is the disallowed action.
Policy parameters are not limited to actions. Other uses of parameters could be to specify directory paths, port numbers, or any other data that may be relevant to a specific policy.

### 7.5 Local Policy State

PoCo policies are stateful, which means they can maintain state information for future decisions. It has been noted that restricting a monitor’s access to state information can have a significant impact on the policies that can be enforced [21]. For example, policies of the simple form “no \(a\) after \(b\)” require state. PoCo utilizes member variables and constructors to store and manage state information. Consider a policy \(P_{\text{time}}\) which disallows successive popups if the interval between them is less than 100 seconds. This policy requires state to record the last popup occurrence and approve of future popup-action attempts. Relevant parts of \(P_{\text{time}}\) are shown below.

```haskell
fun disallow (x:Act) : Pol = 
  name = disx,
  onTrigger = (fun ot(e:Event):Unit =
    case e of
      act a =>
        if a.name == x.name ∧ a.arg == x.arg
        then setOutput(event(act("exit", makeTypedVal(Unit, unit))))
        else unit
      | res r => unit),
  onObligation = (fun oo(rt: ResList):Unit = unit),
  vote = (fun vt(cfg: CFG):Bool =
    ¬call(containsAct, cfg=cfg, name=x.name, arg=(inArg(x.arg):(arg:TypedVal+none:Unit), count=1))
```

Policy parameters are not limited to actions. Other uses of parameters could be to specify directory paths, port numbers, or any other data that may be relevant to a specific policy.
The $P_{time}$ policy above keeps a state variable, $t$, that keeps track of the time of the last popup. The onTrigger watches for a popup event and then checks to see if the current time is less than one hundred seconds after $t$. If it is, the policy suggests that the application be terminated. If at least one hundred seconds have passed, the policy only updates its local variable, $t$, to the new value. The vote for this policy examines other obligations for popup actions. If it finds a single popup action, it performs the same time check as the onTrigger method. If it finds more than one, the policy disallows the obligation.

For the interested reader, Appendix A presents complete specifications of six example policies. Their construction follows directly from the policy components that have been described in this chapter.
The PoCo monitor handles composition of policies by resolving conflicts, scheduling obligations, dispatching the agreed-upon output event, and handing control back to the application or system. A PoCo monitor is configured with three parameters: a set of policies to be enforced, a Vote Combinator, which specifies desired resolution logic for conflicts between obligations and policies, and an Obligation Scheduler, which specifies resolution logic for conflicts between obligations. The following sections explain each of these parameters in turn.

8.1 Policies

The first configuration parameter is a list of policies to be enforced by the overall monitor. Each policy is registered to receive all security-relevant events that the monitor captures and broadcasts. The list of policies does not, necessarily, indicate any sort of priority or order of the policies, which may be more or less important depending on the other parameters supplied to the monitor.

8.2 Vote Combinator

The Vote Combinator or VC is the second parameter and is responsible for combining the policies' votes into a permit/deny decision for each obligation that wants to execute. Because the policy author may wish to combine votes based on information such as policy

\(^1\)Parts of this chapter is published in the International Conference on Software and Computer Applications[20]. Permissions to use the materials are provided in Appendix C.
name, the Vote object was introduced to hold a boolean vote as well as relevant information about the policy that generated it. The VC’s single method, called `evaluate`, takes a list of policies with their votes and returns a boolean decision, thus making the VC’s type as 

\[(\text{name} : \text{String} \times \text{vote} : \text{Bool})_{\text{List} \rightarrow \text{bool}}.\]

The VC resolves conflicts between obligations and policies by requiring each obligation that wants to execute to be voted on and only executing those which are approved. This approval decision can be based on any combining logic that is desired. For example, the VC below approves an obligation only if a policy \(P_1\) does not veto it.

```haskell
fun VCvote(votes:(name:String \times vote:Bool)_{\text{List}}):\text{Bool}=
  let output = ref true in
  let rvotes = ref votes in
  while(!empty(!rvotes)) {
    case head(!rvotes) of
      some v =>
        if v.name == "P1"
          then output := v.vote
          else unit
      none unit => unit;
    rvotes := tail(!rvotes)} end; !output end
```

The PoCo implementation includes several built-in VCs that can be used in their entirety or as building blocks to create other VCs. For instance, a VC that allows an obligation to execute if either \(P_1\) or all other policies allow it can be specified with built-in `Disjunction` and `Conjunction` VCs as demonstrated below.

```haskell
fun VCoverride(votes:(name:String \times vote:Bool)_{\text{List}}):\text{Bool}=
  call(VCdisjunction,call(VCconjunction,tail(votes))::head(votes))
```

A convenient side effect of PoCo’s event-broadcasting and voting mechanism is that policy conflicts are obvious during execution of the VC’s `evaluate` method; any votes to disallow an obligation or any vote that gets overruled by the `evaluate` method are conflicts between policies. It is, therefore, straightforward to detect and act on these conflicts dynamically by adding additional logic to `evaluate`. This enables logging information about the conflict so that it can be used to troubleshoot or make improvements to the affected policies.
8.3 Obligation Scheduler

The Obligation Scheduler or OS is the last parameter and is used to prioritize input policies based on specified criteria. The OS deals with conflicts that arise between obligations when the execution of one obligation would render the execution of another meaningless or detrimental. To resolve this type of conflict, the OS includes a single method named `prioritize` to schedule the execution of obligations. This scheduling is crucial because it determines the final output event for a given input event; the highest priority policy to call `setOutput` will be the one whose output event is returned to the application or system. Like the `vote` method for policies, the OS works with CFG representations of obligations, therefore the type for the OS is \((pol : Pol \times cfg : CFG)_{List} \rightarrow (pol : Pol \times cfg : CFG)_{List}\).

Obligation Scheduler can also implement any desired logic. Examples include prioritizing simpler obligations, weighting specific actions with more or less priority, or applying specific priorities to the policies generating the obligations. The PoCo implementation includes several OSs, such as the one below, which prioritizes the obligations in the same order as their associated policies were provided to the monitor.

\[
\text{fun } \text{OS}_\text{default}(\text{obs}: \text{CFG}_{\text{List}}): \text{CFG}_{\text{List}} = \text{obs}
\]

Another potentially interesting way to order obligations could be based on their complexity. Essentially, this would allow simple obligations that are less likely to cause conflicts to complete before dealing with more complicated obligations. This scheduler attempts to approximate the classic shortest-job-first scheduling algorithm.

\[
\text{fun } \text{OS}_\text{complexity}(\text{obs}: \text{CFG}_{\text{List}}): \text{CFG}_{\text{List}} = \\
\text{call}(\text{sort},(\text{list}=\text{obs}, \\
\text{comparator}=(\text{fun } c((o_1,o_2):(CFG \times CFG)): \text{Int} = \\
\text{call}(\text{length},o_1,\text{nodes})-\text{call}(\text{length},o_2,\text{nodes})) \\
))
\]
8.4 Monitor Operation

PoCo monitor can be viewed as an obligation dispatcher it decides which obligations to execute and in what order. When a security-relevant event occurs, the PoCo monitor first collects and prioritize CFGs of the enforced policies’ onTriggers, then PoCo monitor will process each obligation in order.

To process each obligation $o$, the PoCo monitor first collects votes on $o$ by querying all enforced policies’ vote components. The PoCo monitor then collects these votes and generates a single final decision whether or not the obligation $o$ should be executed. The obligation $o$ will be discarded if it is voted deny. Otherwise, the obligation $o$ will be executed.

Once the obligation $o$ finishes executing, the monitor collects any onObligations that may have been generated by it and adds them to the front of the list of obligations to be processed, thus ensuring that any new obligations triggered as a result of the current obligation are voted on and executed prior to moving on to any other obligations that may be waiting.

After processing all collected candidate obligations, the monitor checks if an output event was set by any of the policies. If one has been set, the monitor will dispatch this event in order to cede control back to the target application or the system. Otherwise, the monitor will use the trigger event as the output event. The PoCo monitor algorithm is presented in Appendix A.

8.5 summary

A PoCo monitor is configured with three parameters: a set of policies to be enforced, a Vote Combinator, which specifies desired resolution logic for conflicts between obligations and policies, and an Obligation Scheduler, which specifies resolution logic for conflicts that arise when a set of obligations does not execute in certain orders causes the execution of
another incorrect. Essentially, the PoCo monitor handles composition of policies by scheduling obligations, dispatching the agreed-upon output event, and handing control back to the application or system.
Chapter 9: PoCo Prototype Implementation\(^1\)

To evaluate and refine our design, we have implemented a prototype of PoCo’s policy composition and enforcement system. Implemented in Java and packaged as a Java library, the implementation is approximately 3,200 lines of code and is available online [citation anonymized]. In this section we will provide details of the compiler module.

9.1 PoCo Compiler Architecture

The PoCo compiler builds a trusted application by inlining security-enforcement code into the untrusted application. To inline the code, we used AspectJ [8], an aspect-oriented extension to Java. The AspectJ compiler inlines code, called advice, that executes before and/or after methods specified with one or more pointcuts [26]. The decision to use AspectJ over manual byte code re-writing was made largely for simplicity and because byte code re-writing to enforce runtime policies has already been accomplished by other projects [32] so there is no novelty in creating an additional implementation.

The PoCo compiler is made up of four modules: the pointcut extractor, policy converter, static analyzer, and AspectJ compiler, depicted in Figure 9.1. Following the flow of code translations, the PoCo compiler takes a list of policies specified in .pol files as input and uses the pointcut extractor to create an AspectJ (.aj) file including security-relevant methods monitored by the policies as the pointcut set. Next, the policy converter reconstructs the .pol files into Java (.java) files and creates a policy-scheduler file using the

\(^1\)Parts of this chapter is published in the International Conference on Software and Computer Applications[20]. Permissions to use the materials are provided in Appendix C.
specified obligation scheduler and vote combinator (in .os and .vc files respectively). Then the static-analyzer statically creates CFGs that represent the actions that may be invoked for each obligation. Finally, given the generated files and the information gathered for each obligation, the AspectJ compiler inlines the policy-enforcement code into the target application.

Figure 9.1: Overview of the PoCo compiler architecture. The compiler takes as input an untrusted application and outputs the same application with policy-enforcement code inlined before and after all security-relevant methods. Ovals are used to represent code files while rectangles represent processes executed during compilation.

9.2 Pointcut Extractor

The pointcut extractor obtains a policy’s events of interest by scanning the onTrigger method, discussed in Section 7.1; it locates all matches calls on the trigger action in the policy’s onTrigger method and extracts the arguments to create AspectJ pointcuts. It is always possible to manually modify the pointcuts after they are determined. This customization may be desirable in cases where complex logic makes it impossible for static analysis to determine the security relevant events or in cases where the writer wishes to manually restrict the events being monitored to improve system performance.

Let’s consider the policy $P_{\text{DisSysCalls}}$ [32] which prevents a target application from exploiting java.lang.Runtime.exec methods. This policy’s onTrigger calls the matches
method to inspect the trigger action using the wildcard ∗ to avoid listing all six overloaded exec methods. If a trigger action matches the java.lang.Runtime.exec methods, the policy attempts to halt the target application by changing the output event to null as the following example illustrates.

```java
public void onTrigger(Event e) {
    String acts = "java.lang.Runtime.exec(*)";
    if(e.matches(new Action(acts)))
        setOutput(new Result(e, null));
}
```

Taking this policy as its input, the pointcut extractor locates the matches method and learns that all of the exec methods are security relevant. The extractor then creates an AspectJ file with a pointcut defined to intercept all of the overloaded java.lang.Runtime.exec methods.

```java
pointcut DisSysCalls(): call(* java.lang.Runtime.exec(*));
```

Once the pointcut has been defined, the pointcut extractor defines advice to execute policy-enforcement code whenever the pointcut is triggered.

### 9.3 Policy converter

Given a list of policies, the policy converter copies relevant sections of the .pol file into a .java file template that includes statements to import required libraries. The policy converter is also responsible for creating a policy scheduler Java source file (described further in Section 9.5), using the specified obligation scheduler and vote combinator or the default PoCo scheduler and combinator. The default obligation scheduler, named OrderAsListed, preserves the original order of input policies by directly returning the input list.

```java
public OrderAsListed extends OS {
    public ArrayList<CFG> prioritize(ArrayList<CFG> policies) {
        return policies;
    }
}
```
The default vote combinator, Conjunction, performs a logical AND operation on the policies’ votes to get its result. This combinator is restrictive and can be used for composing unanimous decision-making policies.

```java
public Conjunction extends VC {
    public boolean evaluate(ArrayList<boolean>) {
        for (Boolean vote : votes)
            if (!vote) return vote;
        return true;
    }
}
```

9.4 Static Analyzer

The PoCo static analyzer utilizes two libraries, Java Compiler Tree [52] and ASM5 [14], to generate CFG representations of each obligation. First, Java Compiler Tree (included in the com.sun.source package) is used to visit an obligation’s abstract-syntax tree (AST) and obtain information about the method calls. Let’s consider the example policy $P_{\text{confirm}}$ from Section 7 which requires every file-open operation attempted by a target application to first be confirmed through a pop-up window.

```java
public void onTrigger(Event trigger) {
    if (trigger.matches(fileOpenAct) && outputNotSet()) {
        if (JOptionPane.showConfirmDialog(null, msg, "Security Question", 0) == JOptionPane.YES_OPTION)
            setOutput(trigger);
        else
            setOutput(new Result(trigger, null));
    }
}
```

By scanning the policy’s onTrigger function with the Java Compiler Tree library, the static analyzer finds three distinct paths (we do not consider the short-circuit expression evaluation for simplicity). All paths first invoke e.matches and outputNotSet methods. After that, one path ends while the other two paths invoke JOptionPane.showConfirmDialog. Depending on the user’s selection in the confirmation dialog each path invokes setOutput with differing
parameters and then end their execution. The CFG representation of the obligation is shown in Figure 9.2. This information is insufficient to make decisions about an obligation’s relevance to the concerns of the implemented policies as it does not contain type details for variables and methods. For example, we only know that the value of the first argument of JOptionPane.showConfirmDialog is null, but because the value null can be assigned for variables of any non-primitive type, we are unable to precisely infer the argument’s type.

By using the ASM5 library, a bytecode manipulation tool that can be used to analyze Java programs, to read compiled policy code, we are able to obtain method signatures as well as the values for the arguments that are statically initialized. By analyzing Pconfirm’s class file, we can get the signature of the setOutput(trigger) method call and determine that the parameter to this method, trigger, is the trigger event which update dynamically at runtime. By mapping the detailed method information onto the control flow information, the static analyzer generates a detailed CFG for each obligation.

For this implementation, dynamic analysis of obligations is limited to changes in the trigger event; this could be extended in future work to include additional options for dynamic analysis. This primary reliance on static analysis leads PoCo policies to be more conservative than what might be accomplished by allowing additional dynamic analysis.

Once all policies have been converted into appropriate AspectJ advice and all obligations have been statically analyzed, PoCo relies on the AspectJ compiler to inline the desired policy-enforcement code into the target application.

9.5 Policy Scheduler

As mentioned earlier, the policy converter is responsible for generating a policy scheduler that uses the configured obligation scheduler and vote combinator. To respond to a security-relevant event, the policy scheduler must use the specified obligation scheduler to prioritize the list of policies and generate an ordered list of obligations. It then obtains the
statically-generated CFG of each policy’s **onTrigger** method and injects the trigger event into it. If the resulting obligation is non-empty, it is added to an obligation queue. Once all obligations have been added to the queue, the obligation scheduler pushes the queue onto a stack that holds all obligations waiting to execute.

To process an obligation, the policy scheduler removes the first obligation from the first queue on the obligation stack and collects votes from all policies on whether to allow the obligation. These votes are passed to the vote combinator to be composed into a single permit/deny decision. An obligation that is denied will be discarded and the scheduler will continue with the next obligation. An obligation that is permitted, is executed and its sequence of security-relevant actions and results, a **result trace**, is dynamically collected. In order to avoid time-of-check to-time-of-use (TOCTOU) vulnerabilities, the voting on and execution of an obligation needs to happen sequentially in a single thread.

Next, the policy scheduler uses the result trace combined with the **onObligation** of each policy to determine if the executed obligation triggers any additional obligations. As with the **onTriggers**, each new obligation is added to a queue and then the queue is pushed onto the obligation stack. This stack of obligation queues ensures that obligations generated by other obligations are executed as soon as possible after the execution of the triggering

---

Figure 9.2: The control flow graph of $P_{confirm}$'s **onTrigger** function.
Figure 9.3: Policy scheduler flow — obligations are generated by policies based on the trigger event, prioritized by the obligation scheduler and then voted on. Executed obligations can result in additional obligations. Once the new queue is added to the stack, the scheduler start the process over with the first obligation in the first queue on the obligation stack. If the obligation stack is empty, the scheduler has completed all obligations. Figure 9.3 illustrates this process.
Chapter 10: Case Studies

To demonstrate the expressiveness and analyze the performance of the PoCo system, we conducted two case studies. On a Windows desktop we replicated the case study that was used to validate Polymer [32], which is the most directly comparable previous work. To test PoCo’s feasibility on a contemporary mobile platform, we also implemented PoCo for the Android platform and tested it on WordPress – an open-source application for managing blogs and websites. The empirical results demonstrate that the PoCo system’s overhead is acceptable on both Windows and Android systems.

10.1 Case Study on Windows Platform

The windows case study is made up of ten policies originally presented in the Polymer [32] work that are designed to prevent unsafe behavior in an email client. Due to differences in the structure of the two systems, there are some differences in how these policies must be written in PoCo, but the goals and results of the policies are the same. All variances from the original case study will be noted for completeness. The policies implemented were:

- **IsClientSigned** - trusts a cryptographically signed application and ensures that an unsigned application is monitored with additional policies. In Polymer this policy takes two policy parameters; the PoCo version instead trusts the target application by setting the trigger event as the output event. By prioritizing this policy as the

---

1Parts of this chapter is published in the International Conference on Software and Computer Applications[20]. Permissions to use the materials are provided in Appendix C.
first policy via the obligation scheduler the PoCo policy serves the same purpose as Polymer’s version.

- **AllowOnlyMIME** - prevents connections other than POP and IMAP.

- **ConfirmAndAllowOnlyHTTP** - disallows non-http connections and opens a popup for user confirmation before allowing HTTP connections.

- **IncomingEmail** - logs incoming emails and flags emails from unknown addresses as SPAM. Our policy includes additional security-relevant methods due to an implementation change in the latest version of Pooka [58], an open-source email client.

- **OutgoingMail** - confirms recipients, adds a BCC and logs all outgoing email.

- **ClassLoaders** - prevents the target application from creating a custom class loader.

- **Attachments** - warns users about dangerous email attachments before creating them.

- **NoOpenClassFiles** - ensures that compiled Java code will not be executed by the target application.

- **DisSysCalls** - prevents the target application from executing system-level calls.

- **InterruptToCheckMem** - monitors the memory consumption of the target application.

- **Reflection** - prevents Java reflection methods from being used to call PoCo methods.

In addition to the slight variances noted in the policies above, PoCo’s flat policy structure, contrasts from the tree-like structure of Polymer. This contrast means that PoCo does not have a concept of superpolicies (policies parameterized by other policies [32]) as Polymer does; instead PoCo sorts and composes a list of policies using an obligation scheduler and a vote combinator. In order to achieve similar effects to those seen in the Polymer case,
we applied the default OS, \texttt{OrderAsListed}, and the default VC, \texttt{Conjunction}, to the policies in the order they are listed above.

Rewriting these policies in PoCo used approximately 1138 lines of code and allowed us to successfully enforce this composed email policy on the email client without modifying the application’s source code.

We analyzed PoCo’s performance by measuring the time overhead incurred by the system, since time overhead impacts user experience. Specifically, we measured the average timing overhead for loading both the application and details of a specific email. The application-loading time is measured from the time that the email client begins executing to when the user can view the inbox. The details-loading time, on the other hand, is calculated from the moment a user clicks on a specific email in the inbox to the moment when the user can view the details of the selected email. To take these measurements, we used AspectJ to intercept events from Pooka and record the time at relevant points during its execution.

To determine the timing overhead of the PoCo monitor only, we measured these two time vectors under three scenarios. First, to establish a baseline, we measured these times of Pooka without enforcing any policies. Then, to exclude the time taken to execute specific policy obligations, we measured these times with a single \texttt{Trivial} policy as well as a policy comprised of ten \texttt{Trivial} policies. These measurements were performed using the same pointcuts that were generated by PoCo for the composed email policy, ensuring that an equal number of events are considered security relevant across all three scenarios. This evaluation was conducted on a MacBook Pro laptop running the macOS Sierra version 10.12.4 with 8GB of memory and a 2.9GHz Intel quad-core i7 processor. For each scenario and time period, we collected the data by executing the required actions 100 times in a consistent university-network environment. The email account that was used to complete the testing contained 15 incoming emails.
Table 10.1: PoCo performance statistics on email client over 100 runs.

<table>
<thead>
<tr>
<th></th>
<th>Load Pooka</th>
<th>Load Email Details</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average(ms)</td>
<td>Median(ms)</td>
</tr>
<tr>
<td>No Policy</td>
<td>6075.851</td>
<td>6187.144</td>
</tr>
<tr>
<td>One Trivial Policy</td>
<td>6128.403</td>
<td>6230.362</td>
</tr>
<tr>
<td>Ten Trivial Policies</td>
<td>6169.487</td>
<td>6255.931</td>
</tr>
</tbody>
</table>

The empirical results demonstrate that the overhead of the PoCo monitor only is relatively small. As shown in Table 7.1, with one trivial policy and ten trivial policies, the average timing overheads for loading Pooka are approximately 0.865% and 1.541%, respectively. The overheads for loading a specific email are approximately 0.392% and 5.060%, respectively.

The overhead of PoCo policies, on the other hand, are dominated by policy obligations, which can vary significantly from one policy to another. We in addition measured the performance with the entire composed email policy; the run-time overhead in this scenario is 17.8s on average. The number seems relatively high when compared to other scenarios. However, during this time period, the PoCo monitor processes 130 security-relevant events in total and triggered not so simple obligations like incoming-email logging, spam-email marking, long-email-subject truncation, and etc. If excluding the overhead of PoCo monitor, the overhead per event is approximately 136.43ms.

10.2 Case Study on Android Platform

To test PoCo’s feasibility on a contemporary mobile platform, we also implemented PoCo under the Android platform and tested on WordPress – an open-source application that helps to create and manage blogs and websites. Ten security policies are enforced on WordPress. Aside from the Trivial policy, eight polices are composed to target some of the most common vulnerabilities of the Android system and the last one is composed to
target one specific vulnerability of WordPress. The empirical results demonstrate that PoCo system’s overhead on Android Systems is acceptable.

10.2.1 Security Policies for Android Systems

As an open-sourced mobile platform, Android has earned popularity as well as criticism for its insufficient security mechanisms ([22] [63] [53]). For instance, built upon a coarse-grained permission-based mechanism, Android platforms suffer from confused-deputy vulnerability [38]. Android platforms also suffer from malware threats. A study in 2015 revealed that around 3-4% of the total applications available in Google Play are malware [4], and 750,000 new Android malware applications have been discovered in just the first quarter of 2017 []. Malware not only can exhaust infected devices’ resources (e.g., CPU, memory, and battery) but also cause users financial losses. For example, a malware stole $140,000 in BTC just in one month by stealthily replacing the cryptocurrency wallet ID that received money with the attacker’s wallet ID [34] [62].

Many solutions have been proposed to alleviate the security challenges on Android platforms, and one common solution is to enforce runtime security policies ([22], [65], [60], [41], and [37]). For instance, sets of policies were proposed in [60] and [22] to secure target applications’ behavior. Among these policies, the privacy policy protects user privacy by returning false IMEI, IMSI and phone number when target applications try to access the phone information, and the SMS policy prevents SMS abuse by prohibiting target applications from sending the text message to prime numbers.

We have implemented eight policies that are proposed in [22] and [60] in this case study. Among these policies, CPUMonitor and MemoryMonitor are behavior-based policies which respectively monitor a target application’s CPU and memory usage to help detect malware. PhoneInfo, CallInfo, and BlockingLocation are privacy policies that restrict target applications from accessing sensitive user or device information. PreventSMSAbuse pol-
icy [22] ameliorate SMS abuse from malicious applications. SMS abuse is one of the most prevalent threats to Android smartphones. *Ad-blocking* policy blocks any API calls from advertisement libraries thereby prevent potential threats originating from the library by blocking any API calls from an advertisement library [28]. These details of these policies are explained in the following subsections.

10.2.1.1 Behavior-based Policies

*CPUMonitor* and *MemoryMonitor* are behavior-based policies which monitor a target application’s CPU, memory usage, respectively. Android platforms suffer from malware threats, and one common behavioral pattern of malware is to exhaust infected devices’ resources (e.g., CPU, memory, and battery). For example, coin-mining malware, one of the major threats to the Android platforms recently, hijacks devices’ resources to perform CPU-intensive operations to mine cryptocurrency without user awareness [68]. Thus, monitoring resource consumption of a target application could help detect malware. In our implementation, these two policies will check a target application’s CPU and memory usage once per second.

For instance, *CPUMonitor* policy monitors CPU consumption periodically. The policy checks CPU consumption every 10 seconds and if the consumption surpasses the preset threshold then only essential functions (e.g., Activity, Service, and Receiver) will be permitted to execute. Thus, an obligation that contains any un-essential Android functions may violate this policy’s security concerns. This policy’s *onTrigger* method is defined as below.

```java
public void onTrigger(Event e) {
    if (cupInfo > maxCPuPercent) {
        if (isEssentialFunction(e))
            setOutput(e);
        else
            setOutput(null);
    }
}
```
10.2.1.2 Privacy Policies

PhoneInfo, CallInfo, and BlockingLocation are privacy policies that restrict target applications from accessing sensitive user or device information. Nowadays mobile applications collect and store users’ information to offer better user experience. However, without proper protection, the information could be breached for illegal usage.

All these three policies include atomic obligations. For instance, a system application that can access a device’s International Mobile Equipment Identity (IMEI), international mobile subscriber identity (IMSI), and phone number could be subjected to identity-fraud threats since the information could identify a unique user. To prevent such a breach, the phoneInfo policy below provides a target application with fake IMEI, IMSI, and phone number information upon request. The policy also issues a warning when an application requests the phone number via a confirmation-dialog popup, where a user can choose to permit or deny the request by offering real or fake numbers, respectively. Because the obligation includes unessential Android functions, it may violate the behavior-based CPUMonitor policy.

```java
public void onTrigger(Event e) {
    if (e.matchesAction(getIMEI)) {
        notifyUser("The app requested your phone’s IMEI information.");
        return fake_IMEI;
    } else if (e.matchesAction(getIMSI)) {
        notifyUser("The app requested your phone’s IMSI information.");
        return fake_IMSI;
    } else if (e.matchesAction(getphoneNumber)) {
        if (showConfirmDialog("Allow the app to get your phone number?")
            == PERMIT_OPTION)
            return phoneNumber;
        else
            return fake_PhoneNum;
    }
}
public boolean vote(CFG cfg) {
    return !cfg.contains(getIMEI) && !cfg.contains(getIMSI) &&
        !cfg.contains(getphoneNumber);
}
```
Nonetheless, users’ contact and geolocation information could also contain sensitive
data. Malicious applications cannot only exploit the information for monetary gain but
also subject users to phishing threats. In this case study, we implemented callInfo and
blurLocation to protect users’ contact and geolocation information respectively. Specifically,
the callInfo policy [22] requests the user’s permission to allow an application to access call
logs. This policy also denies access to stored SMS messages as well as notifies the user of
the access attempt. The BlurLocation policy, on the other hand, protects users’ location by
obscuring the information before returning it to a target application.

10.2.1.3 PreventSMSAbuse Policy

The PreventSMSAbuse policy [22] ameliorates SMS abuse from malicious applications.
SMS abuse is one of the most prevalent threats to Android smartphones. A study revealed
that SMS abuse accounts for 8% of mobile-phone abuse [6]. By exploiting this vulnerability,
malicious applications are able to gain monetary benefit. For example, the FakeInst trojan,
a malware that silently sends SMS messages to premium rate numbers reportedly stole more
than $10 million from its users [75]. Worse, SMS messages can also be exploited to spread
viruses. For example, the Mazar [49], an SMS virus that can gain administrator permissions
on infected devices, is spread through text messages.

To alleviate the SMS abuse threat, PreventSMSAbuse policy monitors and controls
message-sending methods of a target application. Specifically, the policy verifies a message’s
receiver number before allowing the message to be sent in order to prevent text messages from
being sent to premium numbers. The policy is also able to limit the number of messages
a target application can send. In addition, when a target application tries to send text
messages to a number that is not in the user’s contact list, a confirmation-dialog popup
will be issued prior to granting the sending request. PreventSMSAbuse policy requires its
obligation to be carried out atomically.
10.2.1.4 Ad-blocking Policy

The Ad-blocking policy blocks any API calls from advertisement libraries. Designed as a tool to promote applications or make money, advertisement libraries have been intensively bundled by developers into their applications. Studies show that around half of free applications include at least one advertisement library [39] [54]. The inserted advertisements not only consume device power and data [5] but can also trick users into downloading malware such as Boxer SMS Trojan [61]. Worse, advertisement libraries inherit the same permission sets that their host applications have been granted, the libraries thus can be exploited to steal users’ private information (e.g., geolocation or contact information). As such, monitoring API calls of advertisement libraries necessitates sufficient security mechanisms. The ad-blocking policy prevents potential threats originating from the library by blocking any API calls from an advertisement library [28].

```java
public void onTrigger(Event e) {
    if (e.isAction()) {
        if (e.matches(openAd1) || e.matches(openAd2)) {
            setOutput(new Result(e, null));
        }
        else if (e.matches(reflection)) {
            Method mtd = (Method) e.getCaller();
            String clzName = mtd.getDeclaringClass().getCanonicalName();
            if (clzName.equals("com.google.android.gms.ads") ||
                clzName.equals("com.inmobi.androidsdk"))
                setOutput(new Result(e, null));
        }
    }
}

public boolean vote(CFG cfg) {
    if (cfg.containsIncludeUnreslv(openAd1) ||
        cfg.containsIncludeUnreslv(openAd2) ||
        cfg.containsIncludeUnreslv(reflection))
        return false;
    return true;
}
```
10.2.1.5 URL-blacklist Policy

The Internet is full of various malicious websites that try to harm anyone that visit them. One way to alleviate this threat is to enforce a policy that can prevent users from accessing a list of known malicious websites. There are many companies and organizations publish blacklists of URLs, and the blacklist used in this case study are obtained from [42]. Nonetheless, Android systems offered many means for users to surf the internet, and in this policy, all these methods are monitored and any attempts of browsing blocked websites will be logged for auditing purposes. Because the obligation includes unessential Android functions, it may violate the behavior-based CPUMonitor policy. The onTrigger method is defined as below.

```java
public void onTrigger(Event e) {
    if(e.isAction()) {
        String url = null;
        if(e.matches(loadWebview)) {
            url = (String) e.getArg(0);
        } else if (e.matches(openConn)) {
            url = ((URL) e.getCaller()).toString();
        } else if (e.matches(urlActivity) || e.matches(urlActivity2)) {
            Object arg = e.getArg(0);
            if(arg != null) {
                Intent intent = (Intent) arg;
                if(intent.getAction()!=null) {
                    if (intent.getAction().equals(Intent.ACTION_VIEW) &&
                        intent.getData() != null)
                        url = intent.getData().toString();
                    } else {
                        if(intent.getExtras() != null &&
                            intent.getExtras().containsKey("url_to_load"))
                            url = intent.getExtras().getString("url_to_load");
                }
            }
        }
        if (url != null && isInBlackList(url)) {
            Log.e("The target application is trying to browsing a website that is blocked", url);
            setOutput(new Result(e, null));
        }
    }
}
```
10.2.2 WordPress application and a theme policy

We tested our policies on *WordPress* [10], one of the popular open-source applications that helps to create and manage blogs and websites. *WordPress*, powers more than 30 percent of the web, has been used to create millions of websites [71]. To facilitate users to customize their websites, themes can be used. Themes, portraying the appearance of webpages, allows users to define how their materials are displayed in a unified way. According to the statistics from *Scepter Marketing*, by March 2019, there are 31010 themes are available for download or purchase [43].

However, WordPress themes can introduce threats to websites. Recent vulnerability report shows that WordPress suffers various security threats ranging from Brute Force, SQL Injection, Cross-Site Scripting to Malware attacks; and, about 11% of these threats are introduced via WordPress themes [69]. The causes of the vulnerabilities are wide-ranging, from the outdated version to programmers’ careless or malicious intent.

To alleviate the threats introduced by themes, we composed a theme policy that warns users before they activate one. Within the Wordpress app, a user can activate a selected theme by clicking either the “ACTIVATE” button that is displayed in the native UIs. Handling this situation, the theme policy issues a warning when a user clicks an “ACTIVATE” button.

In addition, the WordPress app offers a preview function of a selected theme by displaying its web content. Within this web view, a user can activate the displayed theme by clicking the “Activate this design” button that is embedded in the web page. Since different theme developers can encode the function of the button in different files, we choose to conceal this button from the web page by replacing the caption value of this button to empty. The theme policy includes an atomic obligation that may violate security concerns of *CPUMonitor* policy and is specified as below.
10.2.3 Performance test on WordPress

We analyzed PoCo’s performance on Wordpress by measuring the time overhead incurred by the system. Specifically, we measured the average timing overhead for loading the application, stats, blogs and themes. The application-loading time is measured from the time that the app begins executing to when the main interface is loaded. The blogs-loading time is calculated from the moment a user clicks on the “blog” button in the main interface to the moment when the user can view the list of the posted blogs. The themes-loading time, on the other hand, is computed from the moment a user clicks on the “theme” button in the main interface to the moment when the user can view a list of themes. To measure the performance, we calculated these time vectors under two scenarios: one to establish a baseline by measuring these times without enforcing any policies on WordPress and another one to measure the PoCo performance by enforcing the ten policies on WordPress.

Similar to the previous case study, these measurements are taken by using AspectJ to intercept events from the WordPress add and record the time at relevant points during its execution. Again, the decision to use AspectJ over Android package manipulation was made.
Table 10.2: PoCo performance statistics on WordPress over 150 runs.

<table>
<thead>
<tr>
<th></th>
<th>Loading the App</th>
<th>Loading stats</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average</td>
<td>Median</td>
<td>Overhead</td>
<td>Average</td>
<td>Median</td>
</tr>
<tr>
<td>No policy</td>
<td>478.08</td>
<td>490.5</td>
<td>-</td>
<td>230.46</td>
<td>232</td>
</tr>
<tr>
<td>Ten security policies</td>
<td>520.39</td>
<td>521</td>
<td>8.84%</td>
<td>241.14</td>
<td>237.5</td>
</tr>
<tr>
<td></td>
<td>Loading blogs</td>
<td>Loading themes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No policy</td>
<td>206.33</td>
<td>202</td>
<td>-</td>
<td>232.80</td>
<td>246</td>
</tr>
<tr>
<td>Ten security policies</td>
<td>210.93</td>
<td>205.24</td>
<td>2.23%</td>
<td>261.73</td>
<td>268.57</td>
</tr>
<tr>
<td></td>
<td>Viewing details of themes</td>
<td>Viewing support page of themes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No policy</td>
<td>1337.71</td>
<td>1317.5</td>
<td>-</td>
<td>4301.71</td>
<td>4354</td>
</tr>
<tr>
<td>Ten security policies</td>
<td>1483.73</td>
<td>1481</td>
<td>10.92%</td>
<td>4791.25</td>
<td>4960</td>
</tr>
</tbody>
</table>

largely for simplicity and because it has already been accomplished by other projects [55] so there is no novelty in creating an additional implementation.

This evaluation was conducted with WordPress Version WordPress-Android-11.1.1 on a Motorola Nexus with Android Version 7.1.1. For both scenarios, we collected the data by executing the required actions 150 times in a consistent university-network environment. The empirical results demonstrate that the overhead of the PoCo on Android is endurable. As shown in Table 10.2, with ten enforced policies, the average timing overheads for loading WordPress, stats, blogs and themes are approximately 8.84%, 4.63%, 2.23%, and 12.43% respectively. Comparing with the most relevant related works AppGuard [37], Aurasium [60], Dr. Android [27], and I-ARM-Droid [11] which introduce an runtime overhead of 6.4-21.4%, 14 - 35%, 10 - 50%, and 16.2% respectively, PoCo system’s overhead is endurable.
Chapter 11: Conclusion

Aiming to bridge the gap of lacking satisfactory solutions for composing atomic-obligation policies while maintaining their atomicity, this dissertation presents a policy-specification language and enforcement system, named PoCo, that enables principled composition of atomic obligations.

11.1 Summary

PoCo, short for Policy Composition, is a policy-specification language and enforcement system that enables composed atomic-obligation policies to interact meaningfully and provide provable guarantees while maintaining the policies’ atomicity. Specifically, PoCo has the following properties:

- PoCo supports composing atomic-obligation policies while maintaining their atomicity. To resolve the conflicts between obligations and policies, PoCo employs static analysis of obligations, based on their CFG representations, to allow policies to validate the obligations of other policies before they are executed. To properly account for security-relevant actions that occur during the execution of an obligation, PoCo enables policies to react to executions of other policies’ obligations by analyzing the result traces of an executed obligation.

\[^1\]Parts of this chapter is published in the International Conference on Software and Computer Applications[20]. Permissions to use the materials are provided in Appendix C.
• PoCo supports pre-, post-, and ongoing obligations. To obtain maximum expressiveness of an obligation-based policy enforcement mechanism, PoCo supports all types of obligations since one kind cannot always be substituted by another.

• PoCo supports custom policy-composition logic. PoCo users can customize their unique combining logic through Vote Combinator and Obligation Scheduler components. Vote Combinator specifies desired resolution logic for conflicts between obligations and policies; and, Obligation Scheduler prioritizes obligations for execution.

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. In addition to the design of the PoCo enforcement system, we have defined the formal syntax and semantics that highlight the key features of the PoCo Language and enable formal type-safety reasoning. With the defined semantics, we prove the PoCo language is type-safe through standard type-preservation and progress lemmas. We have also proved that PoCo enables composed atomic-obligation policies to interact meaningfully while maintaining the policies’ atomicity.

11.2 Future Work

This section addresses several possible research directions.

11.2.1 Extend Dynamic Analysis of Obligations

The current implementation relies heavily on the static analysis of obligations, causing PoCo policies to be conservative in their assessment of other policies’ obligations. This is because PoCo policies vote on candidate obligations based on their statically generated CFGs and dynamic analysis of an obligation is limited to only include the trigger event information.
In the future work, dynamic analysis can be employed to achieve more accurate and reliable enforcement results. For instance, data flow analysis can be employed to track other global variables and currently active local variables.

Besides, dynamic analysis also can help achieve better performance of enforcement systems by obviating redundant actions specified in different policy obligations. While simplifying the tasks of specifying and enforcing complex policies by breaking them into manageable modules, this modular design may introduce redundant actions. Information generated through static analysis are conservative approximations; utilizing only this type of information to identify redundant actions can be unreliable. Conversely, dynamic analysis can help locate redundant actions by providing more accurate information about policies and their obligations.

11.2.2 Extend to a general solution

Another possible avenue for future work is to extend our current work to offer a general solution that supports both atomic and non-atomic. Aiming to bridge this gap of lacking satisfactory solutions for composing atomic-obligation policies while maintaining their atomicity, our current solution mainly focused on specifying and enforcing atomic obligations. Thus, PoCo will not be suitable for enforcing the policies that do not require obligations to be carried out atomically.

11.2.3 Identifying conflict pairs of policies

When composing complex security polices, inevitable human error can introduce unwanted conflicts. Helping to locate these types of conflicts, a system can offer feedback information regarding the conflicts between one policy and the obligations of other policies as well as conflicts among a set of enforced policies. Relying on the static analysis, the current PoCo solution output warnings by checking whether any possible execution traces of
an obligation violates security-concerns of other policies. The information of an obligation’s all possible traces is a conservative approximation since it is generated statically.

To offer more accurate conflict information, we can include Log-based security audits to help in locating the sets of suspicious pairs of policies that either partially or entirely contradict each other. Much research has been established in this area. For instance, the authors of [59] created the clusters on extracted log sequences to locate possible conflict policy pairs.
References


Appendix A: The PoCo Monitor Algorithm

Figure A.1 presents the PoCo monitor algorithm. Situated between an untrusted application and the underlying executing system, the PoCo monitor interposes any attempt of executing security-relevant behaviors. Once an attempt is captured, the monitor performs 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) Allow policies to vote 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, return it. Otherwise, output the original input event

The monitor is given as an expression parameterized by $\tau$ and $config$. An instance $e_{\text{monitor}}(\tau,(evt = e,pols = pols,os = os,vc = vc))$ monitors the security-relevant event $e$—whose return type is $\tau$—using policies $pols$, obligation scheduler $os$, and vote combinator $vc$. 
\begin{figure}[h]
\begin{verbatim}
 εMonitor(r, c) :=
 let pols = ref c.pols in
 let obQueue = ref []:(pol:Pol × cfg:CFG)List in
 let obStack = ref []:(pol:Pol × cfg:CFG)List in
 while (~empty(pols)) {
   obQueue := !obQueue θ (pol=head(!pols), cfg=
     makeCFG(inact(evt=c.evt, onTrig=head(!pols).onTrigger):Obligation))::[]:(pol:Pol × cfg:CFG)List;
   pols := tail(!pols);
   let votingPollist = call(c.os, !obQueue) in
   obStack := votingPollist :: !obStack;
   while (~empty(!obStack)) {
     obQueue := head(!obStack);
     let ob = head(!obQueue).cfg in
     let votingPols = ref votingPollist in
     let votes = ref []:BoolList in
     if ~empty(tail(!obQueue)) then obStack := tail(!obQueue) :: tail(!obStack)
     else obStack := tail(!obStack);
     while (~empty(!votingPols)) {
       let pol = head(!votingPols).pol in
       votingPols := tail(!votingPols);
       votes := !votes @ call(pol.vote, ob) :: []:BoolList
     end;
     if call(c.cv, !votes) then
       case ob.\text{obligation} of ot o₁ ⇒ call(o₁.onTrig, o₁.evt)
         | oo o₂ ⇒ call(o₂.onOblig, o₂.rt)
       else unit;
     if ~empty(getRT()) then
       votingPols := votingPollist;
       obQueue := []:(pol:Pol × cfg:CFG)List;
       while (~empty(!votingPols)) {
         obQueue := !obQueue θ (pol=head(!votingPols).pol, cfg=makeCFG(inact
           (rt=getRT(), onOblig=head(!votingPols).pol.onObligation):Obligation))
           :: []:(pol:Pol × cfg:CFG)List;
         votingPols := tail(!votingPols);
       }
       obStack := !obQueue :: !obStack
     else unit
   end
 end
\end{verbatim}
\caption{The PoCo Monitor Algorithm}
\end{figure}
Appendix B: Example Policies

Figures B.1 - B.6 present the example policies used throughout this dissertation written in the PoCo language.

```ocaml
let name = pol_file,
  onTrigger = (fun ot(e:Event):Unit =
    case e of act a =>
      if a.name = "open" \(\land\) tryCast(String,a.arg) = "secret.txt"
        then setOutput(event(act("exit", makeTypedVal(Unit,unit))))
        else unit
      | res r => unit),
  onObligation = (fun oo(rt: ResList):Unit = unit),
  vote = (fun vt(cfg: CFG):bool =
    \neg call(containsAct, cfg = cfg, name = "open",
     arg = in_arg makeTypedVal(String,"secret.txt")(arg:TypedVal+none:unit,count=1))
    \land \neg call(containsAct, cfg = cfg, name = "open",
     arg = in_arg none unit:(arg:TypedVal+ none:unit,count=1)))
)

Figure B.1: pol_file disallows users and obligations from opening the secret.txt file

fun disallow (x:Act):Pol = (
  name = pol_disallow, 
  onTrigger = (fun ot(e:Event):Unit =
    case e of act a =>
      if a.name = x.name \(\land\) a.arg = x.arg
        then setOutput(event(act("exit", makeTypedVal(Unit,unit))))
        else unit
      | res r => unit),
  onObligation = (fun oo(rt: ResList):Unit = unit),
  vote = (fun vt(cfg:CFG):bool =
    \neg call(containsAct, cfg=cfg, name=x.name,
     arg=(in_arg(x.arg)):(arg:TypedVal + none:unit),count=1))
)

Figure B.2: pol_disallow fun disallow specifies a family of policies which disallow the action x
```
(name = polpostlog,
onTrigger = (fun ot(e:Event):Unit =
case e of
  act a => unit
  | res r => if r.act.name == "fopen" then call(log,e) else unit),
onObligation = (fun oo(rt: ResList):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),
vote = (fun vt(cfg: CFG):Bool = true)
)

Figure B.3: polpostlog logs all file-opens after they occur

( name = polprelog
onTrigger = (fun ot(e:Event):Unit =
case e of
  act a =>
    if a.name == "fopen" then
      if outputNotSet() then call(log,e); setOutput(e)
      else case o of
        event o =>
          case o of
            act a1 =>
              if a1.name == e.name
              then call(log,e) else unit
            | res r1 => unit
            | none n => unit
    | res r => unit ),
onObligation = (fun oo(rt: ResList):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),
vote = (fun vt(cfg: CFG):Bool = true)
)

Figure B.4: polprelog logs file-open actions before they are executed
Figure B.5: $pol_{confirm}$ requires all file-open attempts to be confirmed by the user through a pop-up window.

Figure B.6: $pol_{time}$ disallows pop-ups unless at least 100 seconds have passed since the last pop-up.
Appendix C: Copyright Permissions

The following permission is for to use the content in this dissertation.
I. Copyright Transfer, Reserved Rights and Permitted Uses

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

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

Reserved Rights and Permitted Uses

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

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

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

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

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

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

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

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

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

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

(ix) Use any Auxiliary Material independent from the Work. (x) If your paper is withdrawn before it is published in the ACM Digital Library, the rights revert back to the author(s).

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

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

**NOTE:** For authors using the ACM Microsoft Word Master Article Template and Publication Workflow, The ACM Publishing System (TAPS) will add the rights statement to your papers for you. Please check with your conference contact for information regarding submitting your source file(s) for processing.

---

*Please put the following LaTeX commands in the preamble of your document - i.e., before \begin{document}:

```latex
\copyrightyear{2020}
\acmYear{2020}
\setcopyright{acmcopyright}
\acmConference[ICSCA 2020]{2020 9th International Conference on Software and Computer Applications}{February 18--21, 2020}{Langkawi, Malaysia}
\acmBooktitle{2020 9th International Conference on Software and Computer Applications (ICSCA 2020), February 18--21, 2020, Langkawi, Malaysia}
\acmPrice{15.00}
\acmDOI{10.1145/3384544.3384585}
\acmISBN{978-1-4503-7665-5/20/02}
```

**NOTE:** For authors using the ACM Microsoft Word Master Article Template and Publication Workflow, The ACM Publishing System (TAPS) will add the rights statement to your papers for you. Please check with your conference contact for information regarding submitting your source file(s) for processing.

---

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.*
NOTE: Make sure to include your article's DOI as part of the bibstrip data; DOIs will be registered and become active shortly after publication in the ACM Digital Library. Once you have your camera ready copy ready, please send your source files and PDF to your event contact for processing.

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

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

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

II. Permission For Conference Recording and Distribution

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

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

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

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

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

III. Auxiliary Material

Do you have any Auxiliary Materials? ☐ Yes ☐ No

IV. Third Party Materials

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

V. Artistic Images
If your paper includes images that were created for any purpose other than this paper and to which you or your employer claim copyright, you must complete Part V and be sure to include a notice of copyright with each such image in the paper.
We/I do not have any artistic images.
We/I have any artistic images.

VI. Representations, Warranties and Covenants
The undersigned hereby represents, warrants and covenants as follows:

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

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

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

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

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

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

I agree to the Representations, Warranties and Covenants

DATE: 02/24/2020 sent to danid0729@gmail.com at 14:02:42