October 2015

Developing a Compiler for a Regular Expression Based Policy Specification Language

Cory Michael Juhlin
University of South Florida, cory.juhlin@gmail.com

Developing a Compiler for a Regular Expression Based Policy Specification Language

by

Cory Juhlin

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Date of Approval:
October 20, 2015

Keywords: Security, access control, parsing, regular expression, policy composition

DEDICATION

To David, the love of my life, for supporting me in every way possible while I work on my studies.

To my mother, Robin, who not only supported me but has made me the person I am today.

To my stepfather, Bruce, who supports and encourages me as he would his own son.

ACKNOWLEDGMENTS

I would like to thank Dr. Ligatti for encouraging me to take on this project and giving me great feedback every step of the way. This thesis was not part of my plan, but I am so grateful for you nudging me to do it. I would also like to thank Yan Albright and Danielle Fergusson, two fellow graduate students and research partners. Your input and advice was invaluable.

TABLE OF CONTENTS

LIST OF FIGURES ii

ABSTRACT iii

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 RELATED WORK 4
- 2.1 XACML 4
- 2.2 AspectJ 6
- 2.3 Polymer 7
- 2.4 LoPSiL 9
- 2.5 ANTLR 10

CHAPTER 3 POCO SYNTAX OVERVIEW 12
- 3.1 Policy Structure 12
- 3.2 Example Policy - Attachments 13
- 3.3 Example Policy - Audit 17
- 3.4 Example Policy - Root 20

CHAPTER 4 PARSER IMPLEMENTATION 21
- 4.1 Parser Implementation Issues 21
- 4.2 Lexing Rules 23
- 4.3 Parser Grammar 24

CHAPTER 5 STATIC ANALYSIS SCANNER TOOL 27
- 5.1 First Scanner Tool: Execution Pointcuts in AspectJ 28
- 5.2 Second Scanner Tool: Call Pointcuts in AspectJ 29
- 5.3 Scanner Tool Implementation Details 29

CHAPTER 6 IMPLEMENTATION 32
- 6.1 Implementation Strategy 32
- 6.2 Compilation Process 34
  - 6.2.1 Closure Generation 34
  - 6.2.2 Pointcut Extraction 35
  - 6.2.3 AspectJ Generation 35
  - 6.2.4 Java File Generation 37
- 6.3 PoCo Runtime System 39

CHAPTER 7 CONCLUSION AND FUTURE WORK 41

LIST OF REFERENCES 43

LIST OF FIGURES

Figure 3.1	Example PoCo policy for monitoring email attachments.	14
Figure 5.1	Initial setup screen of the PoCo scanner tool.	30
Figure 5.2	Results screen of the PoCo scanner tool.	31
Figure 6.1	Control flow once an AspectJ pointcut is triggered at a security-relevant method.	33

ABSTRACT

Security policy specification languages are a response to today's complex and vulnerable software climate. These languages allow an individual or organization to restrict and modify the behavior of third-party applications such that they adhere to the rules specified in the policy. As software grows in complexity, so do the security policies that govern them. Existing policy specification languages have not adapted to the growing complexity of the software they govern and as a result do not scale well, often resulting in code that is overly complex or unreadable. Writing small, isolated policies as separate modules and combining them is known as policy composition, and is an area in which existing policy specification languages have a number of drawbacks. Policy composition is unpredictable and nonstandard with existing languages. PoCo is a new policy specification language that uses signed regular expressions to return sets of allowed and denied actions as output from its policies, allowing policies to be combined with standard set operations in an algebraic way. This thesis covers my contribution to the PoCo project in creating a formal grammar for the language, developing a static analysis tool for policy designers, and implementation of the first PoCo language compiler and runtime for the Java platform.

CHAPTER 1

INTRODUCTION

As computer software grows increasingly more complex, so do the security issues involving the software. Security policy specification languages allow different organizations to define and enforce rules relevant to their interests and often apply that to untrusted third-party software. These languages give users the power to restrict, prevent, and modify actions they deem security relevant.

Software codebases tend to grow and change over time, which means that the policies that monitor them must grow and adapt as well. Just as general-purpose programming languages have evolved to deal with this growing complexity by offering a variety of methods and best practices to modularize and organize code, so too must policy specification languages else they risk becoming complex at best or unreadable at worst. Existing policy specification languages have a large problem that is exacerbated by the ever-growing complexity of the software they monitor: they do not scale well. Changing parts of existing policies may cause unintended changes elsewhere in the policy, leading to a distortion of the policy designer's original intention. As a result, the policy designer must have a macro-level understanding of how all parts of a set of policies work together.

A modular policy specification language is not new (see [27, 5, 22]). Creating small, self-contained, modular policies focused on a single concern and combining them to create a larger security policy is known as policy composition. The goal is that each piece of a modular policy can be modified and addressed in isolation within a larger, combined policy. While individual policies might be easily-understood, the policy combination mechanisms provided by existing policy specification languages are not easy to reason about or behave in unpredictable ways, sometimes requiring the policy designer to understand and anticipate specific nuances of the language's execution.

In some policy specification languages that are strictly limited to access control (i.e. simply permitting or denying a action made by a monitored program), policy combination is more straightforward and even has the potential to be automated. More expressive policy specification languages that can enforce more than just the set of access control policies (e.g. these policies may modify actions, promote unrelated actions, or a combination of the two) present a large problem for policy composition.

My work builds on the PoCo policy specification language proposed by Daniel Lomsak [17]. PoCo is a general-purpose policy specification language with some unique qualities that make it especially suited to tackling the policy composition issues described previously. Unlike other policy-specification languages, PoCo makes use of regular expressions to express sets of actions during the decision process. The key feature of PoCo that makes it unique among policy specification languages is that policy composition obeys many algebraic properties. This allows the language to be predictable and allows the language to be statically analyzed at compile time for a number of code quality metrics.

The PoCo language uses regular expressions to represent abstract sets of actions. The language is structured around sequences of input and output events, where each input is matched against a regular expression and each output is a signed regular expression (SRE). A signed regular expression is simply a regular expression prefixed with either a positive (+) or negative (-) sign to denote sets of promoted (desired) or denied actions, respectively. Regular expressions are an expressive and concise way to denote sets of actions. Because all PoCo policies return a single SRE for each queried event, policy composition takes advantage of the regular expression's algebraic properties. For example, a policy may combine two SREs, either from subpolicies or different code paths within the policy, to form a new SRE with set operators such as union, intersection, negation, and disjunction.

At a high level, a PoCo policy will make one of three decisions for each queried event:

- Allow original event to proceed.
- Execute an alternative action if the queried event is an action, or return an alternative result if the queried event is a result.
- Deny queried event and halt execution.

Conceptually, the root policy is the first policy queried with a security-relevant event. This policy receives the ouput of all subpolicies and determines what result to return to the PoCo runtime. In PoCo, the graph of policies is directed and acyclic (DAG). The root policy has no incoming edges and the leaves are the policies with no outgoing edges. When a security-relevant event occurs, all policies in the DAG are queried by the PoCo runtime, starting with the leaves. Each policy's output is retained during the query such that each policy is queried once and only once regardless of how many parent policies use its output.

The central thesis presented here is that an implementation of an expressive, general-purpose policy specification language that uses regular expressions as a means for conveying decisions is possible on the Java platform. This thesis describes my work on the PoCo project including formalization of the language and creating a working implementation of a compiler and runtime to prove the concepts of PoCo are viable. My contributions to the PoCo project are as follows:

1. Formalization of PoCo syntax and creation of a grammar for parser generation.
2. Creation of a static analysis policy "Scanner Tool" to analyze the number of method calls in a monitored program that will be intercepted by a PoCo policy.
3. Creation of the PoCo runtime and compiler implementation alongside research partners Yan Albright, Danielle Fergusson, and Donald Ray.

The remainder of this paper is organized as follows. Chapter 2 describes existing policy specification languages and their limitations. Chapter 3 describes the PoCo language and its syntax in detail. Chapter 4 covers the creation of the PoCo language parser and lexer grammar. Chapter 5 details the creation of the PoCo Scanner Tool, a static-analysis tool for exploring how many methods in the monitored program a policy interacts with. Chapter 6 details the implementation of the PoCo compiler. Finally, Chapter 7 concludes this paper with some observations on the project, and future work.

CHAPTER 2

RELATED WORK

Policy-specification languages are an active area of research, with many different approaches to solving the same problem of having a policy-specification language that is expressive, well-defined, and scalable. Much of the existing work involves creating algebras for access control policy composition ([8, 7, 21, 9]). Access control (i.e. denying or allowing an action) is the most basic functionality a policy-specification language can expose and is universally supported by the discussed languages, and by PoCo. Further still are efforts to introduce temporal aspects or state to the policy model ([6, 20, 25]).

This section primarily focuses on existing policy-specification languages. The strengths and weaknesses of each language are presented, along with their implementation details. The other category of related work are the tools used to implement the PoCo language described later in the paper. There are a number of common yet difficult tasks in language compilation that are automated by these tools so that focus can shift to implementing the unique features of the language.

2.1 XACML

Extensible Access Control Markup Language, commonly known as XACML, is an open standard for an XML-based access control language. As it is widely used in industry, it serves as a benchmark for newer policy-specification languages [3]. XACML policies are written in XML and can be combined using a small number of built-in policy combinators. Policies written in XACML can specify an arbitrary number of conditions that must be satisfied to grant an access request. A mechanism called an obligation is provided, allowing a policy to suggest arbitrary actions be taken in the event that its decision is followed. The standard itself does not provide a formal semantics for the language [19], creating several ambiguities regarding the execution of XACML policies (e.g. a policy's obligations may or may not be executed depending on decisions further up the policy tree) [22]. The standard also fails to provide any means of writing custom policy combinators in the XACML language itself. Instead, these custom combinators must be implemented in the software that interprets and enforces a XACML policy and is therefore written in whatever programming language was used to implement the interpreter.

The XACML standard has three key weaknesses that limit its expressiveness. First, XACML is only capable of specifying access control in practice. A policy limited to just access control only has the power to permit or deny an action. Other usage scenarios such as prompting the user for feedback before making a decision or modifying the return values of the requested action are all beyond the scope of simple access control policies. While the standard does provide support for suggesting actions be taken via the obligation mechanism, the execution of these obligations cannot be guaranteed [22]. The obligation mechanism in XACML is typically ignored in most attempts to analyze or extend the language [4, 16, 9] as obligations are treated as an *implementation detail* at the point of enforcement rather than a core part of the specification. Alqatawna et al. [2] propose a generalized obligation combination framework, although it is unable to address the core inadequacies of XACML's obligations such as the inability to query policies on an obligation's results or consider obligations during policy execution [19]. As there is no sanctioned language in the XACML specification for writing a policy's obligations, the authors of [2] recommend that more robust support for obligations be added to the specification itself.

The second weakness is that XACML is a stateless architecture [3]. Any scenario where a policy would need to reference past events would therefore be impossible. Maintaining state is a key feature for a policy-specification language else the set of enforceable policies is greatly reduced [12]. The set of enforceable policies for XACML is therefore a proper subset of PoCo's. Simple scenarios would be impossible such as limiting access to a file after some number of read operations, or preventing writes to a file if another has been read.

Finally, XACML only provides eight built-in combinators but is lacking standard algebraic combinators such as conjunction and disjunction. Some of the provided combinators, such as first-applicable, lack algebraic properties. For example, the first-applicable combinator's result depends on the order in which it queries its subpolicies [22]. Having a limited set of built-in combinators is further compounded by the lack of any formal language for policy designers to specify their own [16]. Some other common combinators that cannot be created with XACML are consensus and majority. Combination of obligations in XACML similarly lacks algebraic properties. Obligations are only promoted from those policies that came to the same decision as the root-level policy [22]. Policies will therefore not be able to know when or if their obligations are executed. Similarly, there is no guarantee that the obligations will be executed in a certain order. For a policy designer, algebraic properties for policy combination simplify policy development because the execution of their policy is predictable even when combined with other policies.

2.2 AspectJ

AspectJ is an extension to the Java programming language that adds support for aspect-oriented programming (AOP). AOP allows a developer to create encapsulations of state and behavior that exist outside the traditional class hierarchy of a program but instead crosscut the hierarchy in some way. For example, an aspect might log to a file whenever ports are opened at any time during the execution of a program. Aspects are the basic functional unit of an aspect-oriented program. The two essential parts of an aspect are pointcuts and advice. Advice are simply method declarations defined against one or more pointcuts. Pointcuts define *where* a piece of advice will execute (e.g. when a certain method is called). One common use of pointcuts is to match a specified method call [27]. A pointcut is a powerful tool because it allows for wildcard matching of method signatures (e.g. "int *(..)" matches any method that returns an integer value). In AspectJ, one pointcut has the potential to match many different method signatures.

AspectJ's ability to monitor method calls within a program lends itself to security-policy specification. Another policy-specification language discussed later, LoPSiL [11], utilizes AspectJ in its implementation. One drawback to using AspectJ solely as a policy-specification language is that AspectJ is a general purpose programming language. As a superset of the Java language's syntax, AspectJ does not enforce any particular structure or organization for policies beyond the fundamental aspect, advice, and pointcut patterns. [27]. It is therefore up to the policy designer to enforce a design methodology that will ensure policies are still able to be reasoned about as they grow in complexity. This makes AspectJ more suitable as an intermediate layer; a policy-specification language can compile down to AspectJ, allowing a programmer to code in an environment tailored to runtime security policy design.

Although AspectJ is not a policy-specification language, it is used for this purpose. As mentioned above, the drawback of the language is that is provides no framework or structure for policy composition. Assume that we wish to combine two premade aspects, each representing a self-contained security policy. If both aspects operate on a shared set of pointcuts, both aspects will execute sequentially in arbitrary order. This is called *aspect interference* [26]. There is not a lot a policy designer can do to combat aspect interference other than specify priority to enforce an ordering of execution for the aspects. A proposed extension to the AspectJ language would make aspects more intelligent about these conflicts and give them the ability to modify their behavior when interference occurs [26], while the authors of [14] created a static-analysis algorithm to detect conflicts when multiple aspects operate on a pointcut.

The process of instrumenting a program with AspectJ is called *weaving*. There are two basic ways in which AspectJ constructs can be weaved into an existing program:

1. At compile time. If the source of the programs you wish to instrument are available to you, you can compile the AspectJ code alongside it and the outputted bytecode will have the appropriate AspectJ hooks weaved into it. Security policies would be of limited use if we were required to obtain the source code to all programs we wished to monitor. Commercial software or software of unknown origin require another weaving method [27].

2. Bytecode (or binary) weaving. Compiled Java programs come as class files (e.g. those in a JAR file) containing Java bytecode. An aspect can be weaved into this bytecode. Typically this can be done prior to running the program on the JVM using tools provided with the AspectJ installation. The bytecode weaving can also be performed at run-time using custom Java class loaders. Either way, the end result is the same: a compiled program without its source code available will be instrumented with the provided aspect [27].

2.3 Polymer

Polymer is a policy-specification language that both addresses the limited functionality and ambiguous obligation propagation of XACML while maintaining a programming methodology designed for security policies, unlike AspectJ. Sharing much of the same syntax as Java, Polymer allows for arbitrary logic to be inserted into policies, providing a greater level of flexibility than in

XACML. A policy in Polymer returns one of six suggestions for any queried action. The suggestions are named as such because a policy has no guarantee that its suggestion will be followed at the root policy. Of particular interest are the insert and replace suggestions. These allow a policy to suggest performing an arbitrary action prior to making a decision (e.g. prompt the user for permission) or replace the return value of an action. Polymer is therefore capable of specifying more than just strict access control policies [5].

The basic unit of Polymer is the policy, which contains a query method, accept method, and a result method. The query method is how the Polymer runtime obtains a policy's response to an action. This query method must not alter a policy's state because there is no guarantee that whatever response a policy provides will be followed. For this reason, a policy has a separate accept method that is called when the runtime is taking the action suggested by a policy. The final method, result, is called after an insert or replacement suggestion is followed and alerts the policy of the return value of that action [5].

Polymer is implemented as a bytecode rewriter that inserts execution hooks around security-relevant methods in compiled programs and libraries, which are then used by the Java virtual machine. In addition to providing the compiler Polymer policy files, the policy designer must enumerate all security-relevant methods in another file. The compiler uses this list of security-relevant methods to know where to weave its functionality in the monitored application. The team also created a class loader that rewrites bytecode as classes are loaded, but not all libraries can be rewritten this way. System libraries, for example, have to be ready upon loading the virtual machine. Polymer's implementation uses methods similar to those used by AspectJ, in which it compiles and weaves code. The Polymer team was aware of this similarity but decided to roll their own implementation for the finer level of control it would grant them [5].

Polymer's structure still relies on the policy designer to adhere to its design patterns for it to be effective. For example, nothing other than convention prevents a policy's query method from altering state, although it must not for correct behavior. While there are benefits to having a syntax based on a general-purpose language like Java, it also means that convention must be followed for the language to behave as advertised, instead of the language itself enforcing that structure [5].

Policy composition in Polymer is flexible, allowing the policy designer to create their own policy combinators if they wish. Polymer's combinators can be powerful but also suffer from the same lack of structural enforcement as mentioned earlier. The built-in conjunction combinator lacks algebraic properties, for example. The high level goal of the conjunction combinator is to take the most restrictive decision from the subpolicies. In Polymer, the conjunction operator lacks associative and commutative properties because of its handling of “insert” suggestions. These insertion suggestions will be executed in the order that the sub policies were specified. The language’s adherence to complete mediation means that each policy is also queried on each insertion suggestion, which means that it is possible a policy’s decision could change as sibling policies execute their insertions [5].

2.4 LoPSiL

LoPSiL is a location-based policy-specification language. As a result of its limited scope compared to the more general Polymer language, LoPSiL’s syntax and structure are tailored to policies relating to the location of the user. Policies in LoPSiL have the ability to react to varying geolocation accuracies and polling frequencies as well as standard location coordinates. The language is of particular use in the mobile computing space, where location services are nearly ubiquitous. Using LoPSiL, a policy designer may restrict an application from acquiring location information when the device is in a certain region (e.g., near the user’s home) or during certain timeframes (e.g., prevent company-mandated applications from tracking the user during non-work hours) [11].

Despite LoPSiL’s novelty, it lacks any sort of support or framework for composing policies. This is in part a side effect of the language design, where a policy specifies parameters (e.g. location granularity assumptions and update frequency assumptions) that are managed by the LoPSiL runtime environment. LoPSiL policies are as a result limited in scope as there is no provided method for handling the complexity involved with combining separate policies [11].

AspectJ is utilized extensively in LoPSiL’s implementation, and the practices used by its creators inspire the work described later in this paper. LoPSiL’s syntax is based off of Java’s so that the process of integrating a LoPSiL policy into AspectJ code is straightforward and allows the LoPSiL implementation to avoid many of the low-level details that would be required when creating a custom bytecode rewriter. Similar to Polymer, the LoPSiL policy designer must supply, in addition to their policy, a text file listing all security-relevant method calls their policy should be aware of. The method calls specified in the aforementioned text file are written in AspectJ pointcut syntax. The compiler will use the method list to generate AspectJ code that forwards method call information to the LoPSiL policies during runtime. The LoPSiL policy (which is already Java syntax) is converted to Java code by simply importing the precompiled LoPSiL runtime classes. The final stage of compilation involves weaving the AspectJ files with the Java programs and libraries to be monitored, which is accomplished using AspectJ's weaving tools [11].

2.5 ANTLR

The projects discussed later in this paper utilize the ANTLR parser generator to create the code required to lex and parse a policy-specification language. ANTLR is an open-source lexer and parser generator written in Java [23]. A lexer converts a stream of characters (typically source code) into a series of tokens, which are the smallest logical groupings for the characters. Given the stream of tokens generated by the lexer, a parser will obtain the structure of the document according to some reference grammar. Parsing and lexing are typically the initial steps when compiling a program.

Another common parser generator is Bison, which is itself based on an even older generator known as YACC [13]. Both ANTLR and Bison require a *context free grammar* in Backus-Naur form (BNF) [23, 13]. A context free grammar specifies parser rules in a recursive way with each rule defined by the tokens and other parser rules it expands to [18].

ANTLR and Bison attack the same problem in different ways. Whereas Bison generates an LALR parser, ANTLR generates LL(*) parsers. LR parsing, used in Bison, is also known as bottom-up parsing. This form of parsing is able to recognize more languages than an LL parser with the same number of lookahead characters [18]. LL(k) parsing is known as top-down parsing. In this scheme, the parser must figure out what nonterminal it is looking at with only k lookahead characters before it begins parsing that nonterminal. An LL(*) parser is not limited to a fixed number of lookahead characters, having instead the ability to match based on regular expressions. Therefore, a LL(*) parser should be able to match any nonterminal that can be matched with a regular expression [18]. One significant drawback of LL parsers is that they are unable to parse grammar rules that have left-recursion, while LR parsers are able to parse both left and right recursion. ANTLR rewrites left-recursive grammars behind the scenes, allowing the grammar to maintain the clarity that left-recursion provides (e.g. `expr : expr + expr`). ANTLR's parsing engine is actually an extension to LL(*) parsing called ALL(*) or adaptive-LL parsing. ALL(*) parsing allows the parser engine to examine the input stream of characters during runtime, which allows the parser to adapt without the need to statically predict all possible parsing outcomes. This means that writing a grammar for both Bison and ANTLR is similar, with ANTLR making sure to correct for any issues specific to LL parsing (e.g. left recursion) behind-the-scenes [23].

ANTLR automatically creates a parse tree from a provided grammar. A very common task in compiler design is to traverse a parse tree, performing specific actions at each node. ANTLR frees the compiler designer from creating a parse tree traversal method by creating a *visitor* class. The visitor class is a parse tree walker that automatically visits each node in the tree. By subclassing the generated visitor class, a programmer can override this default behavior in an ad-hoc manner, customizing the parse tree traversal only where required (e.g. to parse two tokens of an expression in a specific order) [23]. This is a convenience not found in the Bison tool.

CHAPTER 3

POCO SYNTAX OVERVIEW

PoCo's parser and lexer are made to function correctly despite the language's peculiarities. Before we can discuss how we managed to create a parser for PoCo, we need to explain how the language works in detail. In this chapter I will go over the PoCo syntax via examples to illustrate how the various pieces of a policy fit together. In doing so, the reader will have a better understanding of how PoCo's seemingly simple syntax creates a number of problems when implementing the parser because of its use of regular expressions.

3.1 Policy Structure

This section will provide a brief and high-level overview of key portions of PoCo's syntax and structure to aid discussion of the language's implementation details. A PoCo policy at its most basic consists of a single sequence of events, called an *execution*. An execution can contain other executions (grouped in parentheses) or an *exchange*. Exchanges are the basic building blocks of a PoCo policy. Each exchange is a self contained action/reaction pair for a set of events. An exchange is composed of two parts: a matching regular expression and a resulting SRE response. When a policy is queried with an action at runtime, that action is matched against the regular expression in the exchange. If the match is successful, the SRE of that exchange is propagated upwards through the PoCo policy [17].

As stated in Chapter 1, much of PoCo's functionality and logic relies on and interacts with signed regular expressions or SREs. A SRE is simply a regular expression with either a positive or negative sign in front of it (e.g. `` +`(Open|Close)' ``). Positive SREs denote sets of permitted or promoted actions while negative SREs denote sets of denied actions. PoCo's power and expressiveness is derived in large part from the usage of SREs as policy outputs. Since SREs represent sets of actions, SREs can be manipulated and combined in predictable, algebraic ways [17]. PoCo's syntax uses two symbols to indicate the beginning (backtick - `` ` ``) and end (apostrophe - `'`) of a regular expression string. PoCo SREs also use the `%` character as a wildcard. It is similar to the regular expression `.*` that matches any sequence of characters, and is used instead of the period character so that writing function call match strings is more straightforward (i.e. the policy designer may write `` `File.open' `` instead of `` `File\.open' ``).

A PoCo policy has five parts, three of which are optional. The elements of a PoCo policy are listed below:

1. Policy name
2. Variable declarations (optional)
3. Function declarations (optional)
4. Primary execution
5. Transactions (optional)

The primary execution contains the body of the policy. All of a policy's logic is defined within its primary execution. The examples that follow in this section will further elaborate on the primary execution's syntax and structure.

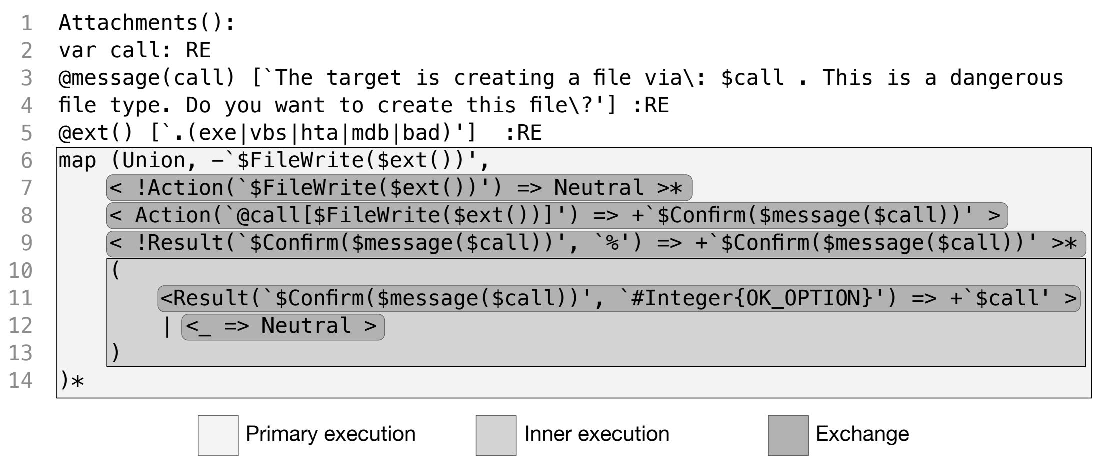
One of the optional components, transactions, are pieces of arbitrary code written in the target language (in the case of our implementation of PoCo, the target language is Java). Transactions can be used as promoted actions from within a PoCo policy, allowing for complex operations to be carried out in the native language of the monitored program.

The rest of this chapter will be composed of a series of three example PoCo policies to illustrate the language's syntax and semantics.

3.2 Example Policy - Attachments

Figure 3.1 shows a PoCo policy that is intended to monitor the types of files (attachments) downloaded by an email client and warn the user when the attachment is potentially dangerous (e.g. it is an executable or script). The high-level behavior of the policy is as follows:

1. If a file write operation is initiated and the extension matches one of the values in the provided "dangerous" extensions, open a dialog box asking the user to permit the action.



```
 1  Attachments():
 2  var call: RE
 3  @message(call) [`The target is creating a file via\: $call . This is a dangerous
 4  file type. Do you want to create this file\?'] :RE
 5  @ext() [`.(exe|vbs|hta|mdb|bad)']  :RE
 6  map (Union, -`$FileWrite($ext())',
 7      < !Action(`$FileWrite($ext())') => Neutral >*
 8      < Action(`@call[$FileWrite($ext())]') => +`$Confirm($message($call))' >
 9      < !Result(`$Confirm($message($call))', `%') => +`$Confirm($message($call))' >*
10      (
11          <Result(`$Confirm($message($call))', `#Integer{OK_OPTION}') => +`$call' >
12          | <_ => Neutral >
13      )
14  )*
```

Figure 3.1. Example PoCo policy for monitoring email attachments. The primary execution is highlighted in the lightest gray while the inner execution is highlighted in a darker shade of gray. Each individual exchange is shaded with the darkest shade of gray.

2. Wait for the user to select an action in the dialog box.

3. If the user elects to proceed, continue with the attempted file write action, otherwise deny the action.

4. Repeat from Step 1.

Line 1 contains the policy name, `Attachments`. Note that policies can be parameterized with other policies. This allows policies to use the results of other policies as part of their decision-making process. An example of such a use would be a policy parameterized on two subpolicies P1 and P2, where P2 is a more restrictive version of P1. The policy may decide to promote P1's permissive output only when a program has a valid signature, otherwise P2's output is promoted.

Line 2 contains a variable declaration. In PoCo, all variables used in a policy must be declared after the policy name. The format is `var ID: TYPE` where ID is the variable name and TYPE indicates whether the variable is a regular expression (RE) or SRE. A variable's value is bound at runtime during a policy's execution.

Lines 3-5 contain two function definitions. Like variables, functions can be declared as either an RE or SRE type. Functions can also be parameterized. The syntax for a function definition is `@ID(PARAMS)[RE or SRE] : TYPE` where ID is the function name, PARAMS is a (possibly empty) comma-separated list of parameter names, and TYPE is either RE or SRE and matches the value inside the square brackets. Line 3 also contains the first example of a variable usage within a regular expression using the dollar sign syntax (`$call`). Any time a dollar sign symbol appears in an RE or SRE, PoCo will expand the value of the function or variable into the string at runtime. For variables, the dollar sign simply prefixes the variable name. For functions, the parameters must also be supplied (an example of that can first be seen on Line 7).

Also note that this policy makes use of globally defined functions `Confirm` and `FileWrite`. `Confirm` is defined as a function call to open a dialogue box with a supplied message (e.g. `JOptionPane.showMessageDialog()` in Java). `FileWrite` is defined similarly to the example below:

```
@FileWrite(ext) [`File.open($ext)'] : RE
```

The `FileWrite` function takes one argument to specify a filename extension of a file opened by the `File.open()` method.

Line 6 marks the start of the policy's primary execution (shaded in the lightest gray), which continues until Line 14. All executions are denoted with opening and closing parentheses; however these parentheses can be omitted for single-element executions (i.e., executions containing a single exchange). The execution on Line 6 is a specialized form of a PoCo execution called a `map`, which is similar to the map functionality in many functional languages. The `map` function takes three arguments: a set operation function (e.g. union, disjunction), an SRE, and an execution. The third argument, a child execution, is wrapped such that the SRE returned by the map is a product of applying the set operator (argument 1) to the SRE (argument 2) and the value returned by the child execution (argument 3).

Executions are by default evaluated sequentially such that each time the policy is queried, the next item in the policy's primary execution is queried. An execution can be made alternating such that all elements of an execution are queried at the same time by placing a bar symbol `|` between items. Similarly, an execution can be postfixed with two operators that modify the sequential execution. The first operator, `*`, means that an execution can be queried zero or more times. The second operator, `+`, means that an execution can be queried one or more times. These operators are intended to emulate the behavior of the same operators in regular expressions.

Line 7 begins the first exchange within the primary execution. The exchanges, shaded in the darkest gray, begin with a `<` symbol and end with a `>` symbol. The first part of the exchange after `<` is the *match* portion. The match is typically a regular expression but can also be a built-in PoCo operator that returns an RE. The arrow symbol (`=>`) is not only the syntactical divider between the match and the response portions of an exchange but a visual cue that a match on the left will promote the response on the right. The response section of an exchange on the right side of the arrow symbol is always an SRE.

A PoCo policy executes by iterating over its primary execution. Imagine a hidden index variable is managed by the policy which identifies the current element in the execution. When an exchange is queried, it will return its SRE only if there is a match. It is therefore important for policy designers to make sure a policy will be able to act on the set of all possible events at each step in its execution.

The exchange on Line 7 uses a specialized match function `Action`, which will only match on action attempts. The opposite of `Action` is the `Result` function, which will match only on a completed action. Line 7 is also an implicit single-element execution with a `*` operator, which means that this exchange can execute an unlimited number of times, or not at all. *If the action is not a file write operation, this policy does not consider it relevant and will return a neutral SRE.*

Line 8 matches any call to write a dangerous filetype. There is a new syntax used here to bind the value of the captured event to the previously-declared `call`. The syntax for variable binding is `@ID[RE]` where ID is the variable name and RE is the regular expression to match to an event. Note that the combination of Line 7 and 8 ensures that that the policy can respond to the set of all possible operations at this stage in the policy's execution. This exchange will promote via a positive SRE a dialog box asking the user to proceed. *If the action is a file write operation, present a dialog box alerting the user to the action.*

Line 9 is another exchange with a `*` modifier. The purpose of this call is to make sure that the dialog box is actually presented. A PoCo policy has no guarantee that a promoted action will actually be acted upon when composed with other policies, so this exchange will repeatedly promote the dialogue box action until it sees a result. *If not the result of the dialog box, promote the dialog box again.*

Line 10 begins an inner execution that alternates between the exchanges on Lines 11 and 12. The exchange on Line 11 checks if the return value of the dialog is in the affirmative and promotes the file write action. Line 12 is a catch-all exchange using the wildcard match `_` that promotes neutral for all other possible events. PoCo's object syntax appears in the SRE of Line 11. The object syntax lets the PoCo runtime match specific values. The syntax is `#ID{VALUE}` where ID is a name of a class in the source language (e.g. the `Integer` Java class) and VALUE is the value to assign the instance of that object. *If the user wishes to proceed, allow the file write. Otherwise deny.*

In this example it is important to note that the `map` operator will make all `Neutral` values deny the file write action. Also, one might wonder why a union of a permit and deny action (i.e. the `map` will union a deny SRE with the permit SRE from Line 11) is possible. As explained in chapter 1, the union operator in PoCo is by default an *optimistic union* meaning positive values are favored over identical negative values.

Finally, the primary execution ends on Line 14. The primary execution has a `*` modifier, so it will repeat itself as many times as necessary.

3.3 Example Policy - Audit

Listing 3.1 contains the code for the Audit policy. The Audit policy wraps another PoCo policy and logs the decisions made by that policy to a file while at the same time propagating those decisions upward.

Line 1 contains the policy name and its parameters, a child `Policy p`, and a filename of the log file `String f`. A child policy can be used just like any other variable within a PoCo policy.

Lines 2-3 contain variable definitions. Variable `act` will be used to store the attempted action, `out` will store the child policy's output on the queried action, and `ps` will store a reference to the output file.

The Audit policy is designed to run in a loop, endlessly logging to a file once started. The exchanges on Lines 5-8 however run only once during the lifetime of the Audit policy and are used for one-time setup tasks. Line 5 binds any action query to the `act` variable and simultaneously stores the child policy's output in the `out` variable. Note the use of the binary operator `&&` in the match sequence of this exchange. Variable binding always returns a true value so the `&&` operator is a shortcut to performing multiple bindings in a single match. This exchange promotes a transaction, `fopen`, to open the file `f`. *On any action, query the subpolicy while promoting the file open action for the log file.*

```
1  Audit(Policy p, String f):
2  var act : RE
3  var out : RE
4  var ps : RE
5  <Action(`@act[%]') && @out[`$p'] => +`fopen($f)'>
6  <!Result(`fopen($f)', `%') => +`fopen($f)'>*
7  <Result(`fopen($f)', `@ps[%]') => +`log($ps, $out, $act)'>
8  <!Result(`log($ps, $out, $act)', `%') => +`log($ps, $out, $act)'>*
9  (
10         <!Infinite(Conjunction(Positive($out), Complement(+`$act'))) => $out>
11         |<Subset($out, +`$act') => +`$act'>
12         |<!Infinite(Positive(Results($out))) => $out>
13         |<!Subset($out, -`$act') && !Subset($out, +`$act') => +`$act'>
14         |<_ => $out>
15         <Result(`%', `%') => $p>*
16         <Action(`@act[%]')&&@out[`$p'] => +`log($ps, $out, $act)'>
17         <!Result(`log($ps, $out, $act)', `%') => +`log($ps, $out, $act)'>*
18 )*
19
20 transaction private static PrintStream fopen(String fn) {
21     return new PrintStream(
22        new BufferedOutputStream(new FileOutputStream(fn)),
23        true
24     );
25 }
26
27 transaction private static void log(PrintStream ps, SRE s, Action a) {
28     ps.println("On trigger action " + a.toString());
29     ps.println("Subpolicy output: " + s.toString());
30 }
```

Listing 3.1. Audit Policy

Line 6 is another example of an exchange that runs indefinitely to ensure the action promoted in Line 5 is actually executed. *Until the log file is opened, promote the file open action for the log file.*

Line 7 binds the result of the file open action to the `ps` variable and promotes another transaction, `log`, to write the queried action and subpolicy's output to a file. *When the log file is opened, write the action and subpolicy's output to the log file.*

Similar to Line 6, Line 8 ensures the logging action from Line 7 is executed.

On Lines 9-18 is a child execution that executes indefinitely. Lines 10-14 alternate between five different exchanges. These exchanges represent the following five possibilities of subpolicy output:

1. Subpolicy has permited/promoted a finite set of actions where the queried action `act` may or may not be a proper subset.

2. Subpolicy has permitted actions that include the queried action `act`.

3. The subpolicy has permitted a finite number of results.

4. The queried action `act` is neither permitted nor denied by the subpolicy.

5. All other scenarios.

The exchange on Line 10 represents the first scenario. A number of new boolean match operators are introduced here. First, we use the unary `Complement` operator to get the set of all possible actions excluding `` +`$act' ``. The unary `Positive` operator returns only the set of positive values from the subpolicy's output. These two values are provided to the binary `Conjunction` operator which returns the set of intersecting actions from two sets. Finally, the unary `Infinite` operator returns true if a set covers an infinite number of actions. Since we're negating the result of `Infinite`, we want to know if the policy has promoted an action outside of the queried action. The net result: *if the subpolicy's output contains a finite set of actions excluding the queried action, promote the subpolicy's output.*

Line 11 represents the second scenario. Here, we use the binary `Subset` operator to check if the queried action is part of the actions permitted by the subpolicy. *If the subpolicy permits the queried action, permit the action.*

Line 12 represents the third scenario. Here we check that the results permitted by the subpolicy is a finite set (i.e. the subpolicy has done more than return a wildcard). *If the subpolicy has a finite set of permitted actions, promote the subpolicy's output.*

Line 13 represents the fourth scenario. Here, we check to see that the queried action is neither permitted nor denied by the subpolicy. We interpret this as the subpolicy not considering the action relevant and will therefore permit the action. *If the subpolicy does not care about the queried action, permit it.*

Line 14 represents the final scenario and defaults to promoting the subpolicy's output if none of the other scenarios are true. Line 15 is also a catch-all that simply defers any result events to the subpolicy.

Line 16 will capture the next queried action event and output it to the log file similar to Lines 5 and 7. Line 17 is identical to Line 8.

Lines 20-30 contain transaction methods written in native Java code. These methods are attached to the generated PoCo policy object and can be called from within the policy. As the name implies, transactions are treated as atomic actions in PoCo.

3.4 Example Policy - Root

The PoCo language also supports a slightly different and more restricted syntax for declaring a root policy. PoCo policies are intended to be self-contained and modular. To combine various policies together, a root policy can define a policy tree describing how each policy is queried from the root.

```
import DenyEmails
import NoOpenPorts

Main():
tree rootNode = Union(DenyEmails(), NoOpenPorts())
```

Listing 3.2. PoCo Root Policy

Listing 3.2 is an example of a small root policy that combines the results of two policies: `DenyEmails` and `NoOpenPorts`. A root policy is structured similarly to a normal PoCo policy, but the policy's body may only contain tree definitions. Each tree may be defined in terms of an SRE operator (e.g. `Union` or `Conjunction`), another PoCo policy, or another tree. The first tree in the root policy becomes the root node of the entire policy tree.

The syntax for a tree definition is as follows:

```
tree NAME = ID ( PARAMS )
```

where `NAME` is the name of the tree while `ID` can either be a policy name, name of another tree, or an `SRE` operator. `PARAMS` is a comma-separated list of parameters to the tree node.

The PoCo policy examples presented in Sections 3.2, 3.3, and 3.4 have demonstrated the PoCo syntax and structure in enough detail for our discussion of the parser implementation to proceed. While the syntax of PoCo is limited to more general-purpose programming languages, its expressiveness and power are derived from the regular-expression syntax of its SREs.

CHAPTER 4

PARSER IMPLEMENTATION

The first step in implementing a complete PoCo compiler is to implement the parser. The parsing stage of compilation processes the input source file not at a character by character level, but as a series of predefined language-specific tokens. This series of tokens is obtained by running the source program through a separate program called a lexer. At this level, only basic errors will be caught such as invalid characters in variable names. Once the lexer has output a stream of tokens, the parser groups the tokens into semantic units according to the rules specified in a provided grammar file. Parsing will catch even more errors in the source program because the grouping of tokens must now have meaning (e.g. `new new` would trigger a parser error in Java). As stated in Chapter 2, the PoCo parser and lexer implementation uses the ANTLR generator [23].

This chapter will go over the process of implementing the PoCo parser. First, a number of the difficulties associated with parsing PoCo will be discussed. Then, a discussion of the lexer rules and parser grammar will follow.

4.1 Parser Implementation Issues

There are a few unique challenges related to parsing a language such as PoCo. Languages such as C or Java have string literals (usually a sequence of characters between a set of quotes), but they don't have to parse what's inside of the strings. PoCo's regular expressions are similar to string literals in the way they are used as data in the policy evaluation process, but they can also have structures within that are semantically relevant to compilation.

During compilation of a PoCo policy, the PoCo-specific structures within the regular expressions must be parsed and translated while at the same time leaving the raw regular expression intact to be used for matching at runtime. Typically, when parsing a string literal in another language, the lexer would treat all characters between the start and end symbols (typically quotes) as part of the literal and output a single token. For PoCo we do not have the luxury of treating all characters between the opening and closing symbols of a regular expression (` and ') as part of a regular expression literal — we need to know the details of what happens inside the regular expression while treating the bulk of it as a string literal. This required the use of ANTLR's lexer modes. Lexing modes allow us to switch the lexer to a new scope when a certain token is reached, where each mode has completely separate and distinct set of token definitions. These modes can be popped and pushed off a stack so that we can switch between modes at any time during the lexing process. This gives us the power to use a completely different set of tokens while inside certain PoCo language constructs than the ones used while outside.

For PoCo, we triggered a lexing mode shift whenever the opening character of a regular expression (`) was encountered, and popped the mode off of the stack when unescaped end character (') was encountered. A regular expression is a specialized string and can contain an arbitrary sequence of characters whereas the PoCo code outside of a regular expression has a very limited set of acceptable tokens. Within the regular expression we also check for PoCo-specific syntax (e.g. variable binding `` `@call[%]' ``) which can also trigger additional lexing modes. The power of lexing modes does come at a cost, however, as distinct entry and exit tokens are needed to trigger the switching of the various lexing modes. This led to some last-minute PoCo syntax tweaks (e.g. the backtick character used to begin regular expressions). The flexibility and power afforded by lexing modes also lends itself to a more complicated grammar. Similarly, since each mode is completely separate, any overlap in token scope requires the tokens be declared in each mode. This creates complexity and makes the grammar more prone to errors.

Some of the PoCo-specific structures encountered within the regular expressions cannot be statically resolved. For example, a matching regular expression might be written as `` `$call' ``, which means that the variable `call` should be inserted into this string. If `call` is assigned during runtime, then the value of this regular expression cannot be determined at compile time. Similarly, there exist PoCo statements such as variable binding and SRE operators that change the value of a string at runtime. To address this issue, the PoCo parser "slices" up regular expression literals into a series of string literals and PoCo syntax. This way, the parser can generate code to reassemble the string at runtime while integrating the PoCo-specific functionality into the string.

```
17  TRANS:      'transaction' -> pushMode(TRANSACTIONS) ;
18  NEUTRAL:    'Neutral' ;
19  MAIN:       'Main' ;
20  VAR:        'var' ;
21  IMPORT:     'import';
22  TREE:       'tree';
23  DOT:        '.' ;
24  LPAREN:     '(' ;
25  RPAREN:     ')' ;
26  LBRACE:     '{' ;
27  RBRACE:     '}' ;
28  MAP:        'map' ;
29  COMMA:      ',' ;
30  AT:         '@' ;
31  RBRACKET:   ']' ;
32  LBRACKET:   '[' ;
33  DOLLAR:     '$' ;
34  LTICK:      '`' -> pushMode(INSIDERE) ;
```

Listing 4.1. PoCo Lexer Modes Lines 17-34

4.2 Lexing Rules

The PoCo syntax has been discussed in detail via the examples presented earlier in this chapter. Similarly, the reasoning behind some of the design decisions for our parser grammar have been presented in the previous section. What follows is an overview of the PoCo lexer rules.

Listing 4.1 shows a portion of the PoCo lexer modes. This portion of code contains many of the PoCo syntax related to policy declaration, executions, and exchanges. Recall from the previous section that PoCo's unique situation requiring PoCo syntax to be parsed within regular expression literals requires us to use lexer modes. Lines 17 and 34 are examples of pushing a new lexing mode onto the stack when a certain character is encountered (e.g. when the opening backtick `` ` `` of a regular expression is found).

Listing 4.2 contains the next portion of PoCo lexer modes. Note that on Line 58, a new mode named `INSIDERE` begins. This mode specifies the tokens within a regular-expression literal. The `SYM` token catches all string literal characters except those reserved for PoCo-specific syntax. We also transition to other modes when we encounter a start symbol for a PoCo language structure (e.g. the `@` symbol for variable binding).

Listing 4.3 contains the final portion of the PoCo lexer modes. In this listing four additional modes are declared, `REVAR` for the `$` variable syntax, `OBJECT` for the `#` object syntax, and

```
58 mode INSIDERE;
59
60 INIT:       '<init>' ;
61 SYM:        ('\\' [`\[\]\?@%$\\\*\+:,\(\){}<>#\'\|] | ~[`\[\]\?@%$\\\*\+:,\(\){}<>#\'\|])+ ;
62 RELPAREN:  '(' -> type(LPAREN) ;
63 RERPAREN:  ')' -> type(RPAREN) ;
64 RELBRACE:  '{' -> type(LBRACE) ;
65 RERBRACE:  '}' -> type(RBRACE) ;
66 REDOLLAR:  '$' -> type(DOLLAR), pushMode(REVAR) ;
67 REAT:       '@' -> type(AT), pushMode(REBIND) ;
68 APOSTROPHE: '\'' -> popMode ;
69 VARCLOSE:  ']' -> type(RBRACKET) ;
70 REASTERISK: '*' -> type(ASTERISK) ;
71 REPLUS:     '+' -> type(PLUS) ;
72 REQUESTION: '?' -> type(QUESTION) ;
73 REBOP:      '|' -> type(BAR);
74 REPOUND:    '#' -> type(POUND), pushMode(OBJECT) ;
75 RECOLON:    ':' -> type(COLON) ;
76 RECOMMA:    ',' -> type(COMMA) ;
77 REWILD:     '%' ;
78 NEST:       '`' -> type(LTICK), pushMode(INSIDERE) ;
```

Listing 4.2. PoCo Lexer Modes Lines 58-78

TRANSACTIONS for code written in the Java language. The REVAR and OBJECT modes are all used to obtain a token for a variable identifier. Since variable usage in PoCo does not have a distinct end symbol (e.g. $call), all possible end symbols must be enumerated in this mode. This involves duplication of most of the values in the INSIDERE mode from Listing 4.2. The TRANSACTIONS mode is designed to take the native Java code verbatim and contain it within a single token, TRANSCONTENT.

4.3 Parser Grammar

The entire PoCo parser grammar is designed to identify all PoCo related structures in a policy whether they exist inside or outside of a regular expression. While most of it mirrors the syntax and semantic examples presented earlier in the chapter, there are some parts of the grammar worth discussing in detail: regular expression grammars.

Line 173 of Listing 4.4 is of particular interest. This marks the start of the grammar rule re for PoCo-style regular expressions. Within this rule are the multiple items that can appear within a PoCo regular expression. Previously, it was mentioned that the PoCo parser maintains the string literal tokens of the regular expression alongside the syntactically-relevant parts. This rule makes it possible. Note that re is defined recursively (see Lines 183-184). The rebop (RE binary operator)

```
84  mode REVAR;
85
86  ID3:       [a-zA-Z][a-zA-Z0-9_\-]* -> type(ID) ;
87  DOT2:      '.' -> type(DOT) ;
88  REVARAPOSTR:'\'' -> type(APOSTROPHE), popMode, popMode ;
89  WS2:       [ \t\r\n] -> type(SYM), popMode ;
90  EXITVAR1:  '(' -> type(LPAREN), popMode ;
91  EXITVAR2:  ')' -> type(RPAREN), popMode ;
92  EXITVAR3:  '{' -> type(LBRACE), popMode ;
93  EXITVAR4:  '}' -> type(RBRACE), popMode ;
94  EXITVAR5:  ']' -> type(RBRACKET), popMode ;
95  EXITVAR6:  '*' -> type(ASTERISK), popMode ;
96  EXITVAR7:  '+' -> type(PLUS), popMode ;
97  EXITVAR8:  '?' -> type(QUESTION), popMode ;
98  EXITVAR9:  '|' -> type(BAR), popMode ;
99  EXITVAR10: ':' -> type(COLON), popMode ;
100 EXITVAR11: ',' -> type(COMMA), popMode ;
101 ESCAPED:   ('\\' [`\[\]\?@%$\\\*\+:,\(\){}<>#\'\|]) -> type(SYM), popMode ;
102
103 mode OBJECT;
104
105 ID4:       [a-zA-Z][a-zA-Z0-9_\-]* -> type(ID) ;
106 DOT3:      '.' -> type(DOT) ;
107 OBJLBRACE: '{' -> type(LBRACE), popMode ;
108
109 mode TRANSACTIONS;
110
111 ENDTRANS:     'end transaction';
112 QUOTEDCONTENT: '"' ( '\\"' | . )*? '"' ;
113 TRANSCONTENT:  ((~('"')) | QUOTEDCONTENT)+? ENDTRANS -> popMode;
```

Listing 4.3. PoCo Lexer Modes Lines 84-113

```
173 re:        rewild |
174            DOLLAR qid LPAREN opparamlist RPAREN |
175            DOLLAR qid |
176            function |
177            object |
178            LPAREN re RPAREN |
179            LPAREN RPAREN |
180            AT id LBRACKET re RBRACKET |
181            INIT |
182            SYM+ |
183            re reuop |
184            re rebop re ;
185
186 function:  fxnname INIT LPAREN arglist RPAREN |
187            fxnname LPAREN arglist RPAREN ;
188
189 fxnname:   SYM+ |
190            object |
191            object SYM+ ;
192
193 arglist:   re |
194            arglist COMMA re |
195            ;
196
197 rebop:     BAR |
198            ;
199
200 reuop:     ASTERISK |
201            PLUS |
202            QUESTION ;
203
204 rewild:    REWILD ;
```

Listing 4.4. PoCo Parser Grammar Lines 173-204

rule on Line 198 allows itself to be empty. This means that `re` rules can be combined without any separators. Similarly, the rule on Line 198 allows the `re` node to absorb as many syntactically-irrelevant characters as needed — this accomplishes the stated goal of preserving the string literal tokens alongside the PoCo-specific structures.

CHAPTER 5

STATIC ANALYSIS SCANNER TOOL

One of my contributions to the PoCo project was to create a static analysis tool that gives policy writers a visual mapping between their policy and the methods it will interact with [15]. A single PoCo policy and any number of compiled Java classes or JAR (Java archive — a zip package of compiled Java classes) files may be input into the tool. The tool extracts all regular expressions from the PoCo policy that are used to match action events. With this list of regular expressions, the compiled Java code and referenced libraries are scanned for all method calls. Each method call is mapped to zero or more of the regular expressions. The final output is a table listing all methods that will be matched by each regular expression in a policy.

Our PoCo compiler implementation outputs a combination of AspectJ and Java code to implement PoCo policies. Recall from Section 2.2 that AspectJ uses *pointcuts* to define where to execute an advice's code [27]. Pointcuts define a method signature pattern. Any method that matches this pattern will trigger that pointcut. While there are a number of pointcut types, there are two that are applicable to the PoCo project: *execution* and *call* pointcuts. An *execution* pointcut triggers a piece of advice at the execution point of a method, while a *call* pointcut triggers advice at the place a call to the specified method is made.

To illustrate the difference between *execution* and *call* pointcuts, consider a pointcut that matches the Java standard library method `System.out.println()`. Also assume that this AspectJ code is only instrumenting a third-party application that we intend to monitor. If we declare the pointcut as an *execution* pointcut, no advice using this pointcut would trigger even when the monitored program uses `System.out.println()`, while a *call* pointcut would trigger execution as expected. The difference is that an *execution* pointcut will only function properly if the Java code containing the method is also instrumented by the AspectJ compiler. In this case, the Java standard library (rt.jar — over 60 MB on a Java 8 JRE) would need to be instrumented. However, using a *call* pointcut would result in the expected behavior and only need to instrument the monitored Java program.

The scanner tool has two iterations that correspond to changes in the underlying PoCo compiler implementation. Midway through development, the PoCo compiler switched from using *execution* pointcuts to using *call* pointcuts. Both iterations will be described in the following sections starting with the initial premise and ending with the current functionality.

5.1 First Scanner Tool: Execution Pointcuts in AspectJ

Initially, the PoCo compiler was outputting AspectJ code using *execution* pointcuts. The advantage to using *execution* pointcuts is that monitored programs cannot use reflection to bypass policies. *Execution* pointcuts trigger at the execution site of a method and are triggered regardless of the mechanism by which a method is invoked. As demonstrated in the previous section's example, the disadvantage of using *execution* pointcuts is that all code and libraries used by the monitored program must also be instrumented by the AspectJ code.

The scanner tool was initially envisioned as both a static analysis tool to assist policy designers by visualizing the scope of their PoCo policy and also perform a required task for PoCo policy compilation. While AspectJ supports some wildcard features and variable numbers of arguments in pointcut declarations, they are not as expressive as regular expressions. The scanner tool's most important purpose was to translate the regular expressions from a PoCo policy into concrete lists of methods that can be translated into AspectJ pointcuts during compilation.

Performance was a major concern with the initial iteration of the scanner tool. All libraries used by the monitored program must be input into the tool. At the very least this means that Java's standard libraries must be input. When testing on a Java 8 Update 31 JRE, the rt.jar containing most of the Java standard library had over 179,000 methods signatures. Each regular expression from the PoCo policy must iterate over the set of extracted method signatures, creating a significant processing bottleneck. By splitting up the matching task into four threads, average runtime was decreased from an average of 8152 milliseconds to just 3382 milliseconds — an almost 2.5 times speedup. The test involved 30 regular expressions containing known Java Standard Library method names (e.g. `println`) and a method signature pool of 221,833 methods for a total of over 6.5 million regular expression comparisons.

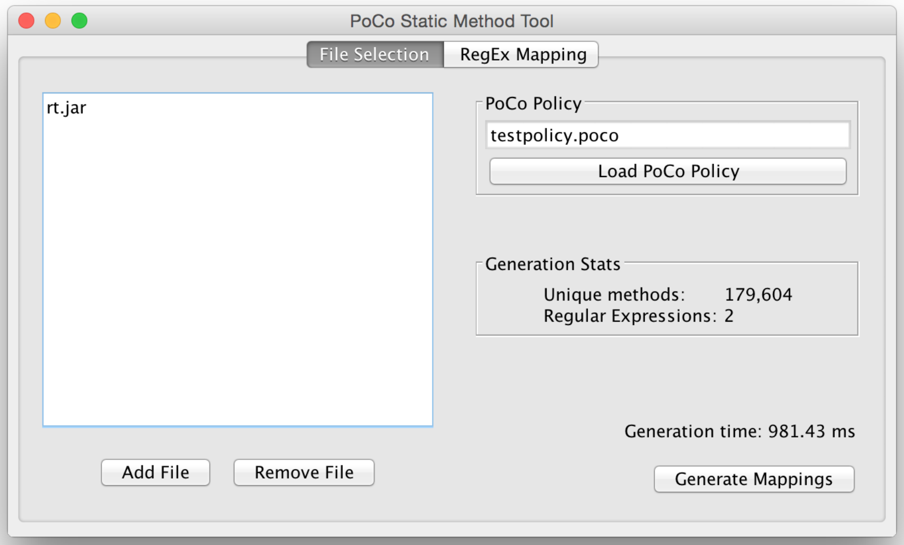
5.2 Second Scanner Tool: Call Pointcuts in AspectJ

During the development of the PoCo compiler, the research team collectively decided that we should be using *call* pointcuts instead of *execution* pointcuts in our generated AspectJ code. When using *execution* pointcuts, the requirement that a policy designer input all libraries and code into the scanner tool places an extra burden on the policy designer as he or she must know the location of all code used by the monitored program. It also creates an opportunity for error and a false sense of security: if the policy designer forgets to instrument a library used by their monitored program, their policy will be unaware of any code that is executed within that library. Using *call* pointcuts obviates the need to instrument the entire runtime environment. Note that *call* pointcuts will not capture calls to monitored methods from methods declared outside the monitored program (e.g. library methods). We can get around this limitation by instrumenting the libraries used by the monitored program, however in PoCo we leave this as the responsibility of the policy designer to know what methods have security-relevant side-effects.

As previously mentioned, *call* pointcuts are unable to trigger when a monitored function is invoked via an alternate means such as reflection. To address this issue, the initial PoCo implementation will monitor calls to Java's reflection APIs and allow the policy designer to restrict access to these language features. As a result of the switch, only the monitored program must be input into the scanner tool. This results in a smaller and more targeted set of method signatures that must be compared with the PoCo policy. The performance issues of the initial iteration will therefore be reduced, however the processing is still multithreaded for greater scalability.

5.3 Scanner Tool Implementation Details

The Scanner tool is implemented as a Java 8 Swing GUI program. The ASM 5 bytecode engineering library [10] is used to decode compiled Java programs and extract their method calls. ANTLR 4.5 [24] is used to parse the input PoCo policy.



rt.jar

PoCo Policy: testpolicy.poco

Generation Stats
Unique methods: 179,604
Regular Expressions: 2

Generation time: 981.43 ms

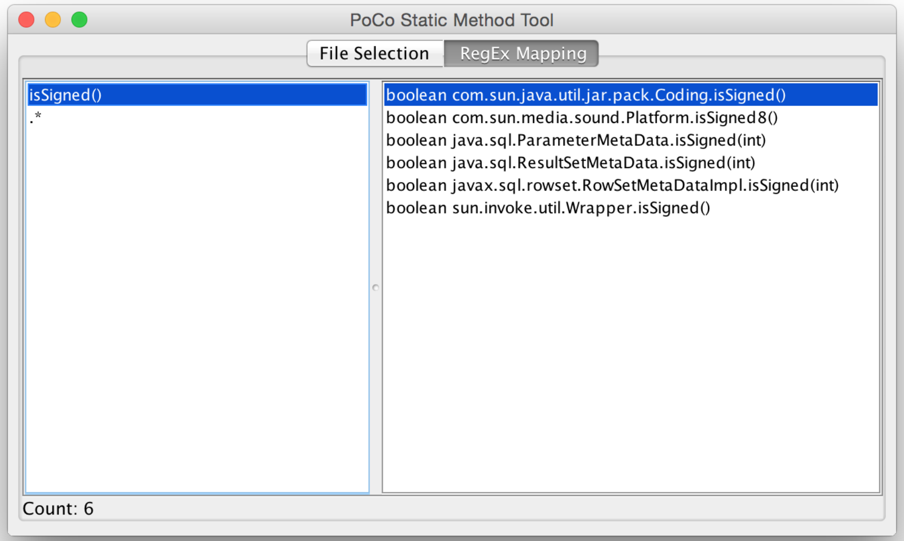
Figure 5.1. Initial setup screen of the PoCo scanner tool.

The initial screen of the PoCo scanner tool is shown in Figure 5.1. Any number of compiled Java files may be input for scanning; however, the typical usage scenario only requires the monitored Java program to be input. When the user clicks on the “Generate Mappings” button, the tool begins a series of operations, detailed below:

1. Create parse tree. At this stage a parse tree of the input PoCo policy is constructed using the ANTLR library.

2. Parse variable and function information. The tool walks the PoCo policy parse tree and creates a catalog of the declared variables and functions. Each variable or function’s name, type, and contents are stored in the catalog.

3. Parse PoCo policy body. The tool walks the parse tree and evaluates each action-matching regular expression. These regular expressions are then *resolved* so that all variable and function references are evaluated and expanded. If at any point during this process a regular expression depends on a variable that is bound at run time, the matching expression is ignored and a warning is issued to alert the user of the potential incompleteness of the scan.



```
isSigned()
.*
```

```
boolean com.sun.java.util.jar.pack.Coding.isSigned()
boolean com.sun.media.sound.Platform.isSigned8()
boolean java.sql.ParameterMetaData.isSigned(int)
boolean java.sql.ResultSetMetaData.isSigned(int)
boolean javax.sql.rowset.RowSetMetaDataImpl.isSigned(int)
boolean sun.invoke.util.Wrapper.isSigned()
```

Count: 6

Figure 5.2. Results screen of the PoCo scanner tool.

4. Scan compiled Java files. Obtains a list of method call signatures from the compiled Java bytecode.

5. Map expressions obtained in Step 3 to the signatures obtained in Step 4.

Once the mapping process is complete, the user can navigate to the second tab of the tool's interface seen in Figure 5.2. Each regular expression obtained from the PoCo policy is listed in the left pane, and a list of all method calls the regular expression matches is listed in the right pane.

The PoCo scanner tool doubles as a static analysis tool for policy designers and an integral step in the compilation of the PoCo policy. Translating between the regular expressions of a PoCo policy to concrete method signatures is further complicated by the addition of PoCo-specific syntax which must be parsed and evaluated to convert to a canonical regular expression in the Java language. Policy designers may use the tool to evaluate whether their matching expressions are too broad (which may negatively impact performance, due to unnecessary policy queries) or too narrow (which may negatively impact security due to some security-relevant actions not being monitored).

CHAPTER 6

IMPLEMENTATION

My final contribution to the PoCo project was to work on an implementation and compiler of the PoCo language to demonstrate PoCo's feasibility. This project was a joint effort between myself, Yan Albright, Danielle Ferguson, and Donald Ray [1]. Our implementation is Java based, allowing PoCo policies to monitor arbitrary Java programs by inlining hooks into the program's bytecode. We use the following open-source tools in our PoCo compiler:

- ANTLR 4 [24] for parser and lexer generation.
- ASM [10] for analyzing compiled Java bytecode.
- AspectJ [27] for inlining policy code into compiled Java programs.

The following sections will cover the high level structure of our PoCo implementation, challenges and solutions for implementing PoCo, and an example of a compiled PoCo policy.

6.1 Implementation Strategy

The implementation of the PoCo compiler is influenced by the work done on the LoPSiL language [11]. LoPSiL's implementation uses AspectJ as an intermediate language to intercept security-relevant method calls and query a policy object. PoCo's implementation is similar. As discussed in Section 2.2 and Chapter 5, the PoCo implementation outputs AspectJ code that uses *call* pointcuts to intercept security-relevant methods. A piece of *around* advice executes at the call site for each security relevant method. "Around" advice effectively replaces the method body with our own code that can yield control to the original method if needed. Within this advice, we forward the details of the action to the PoCo policy runtime, which will query its policies and return a result.

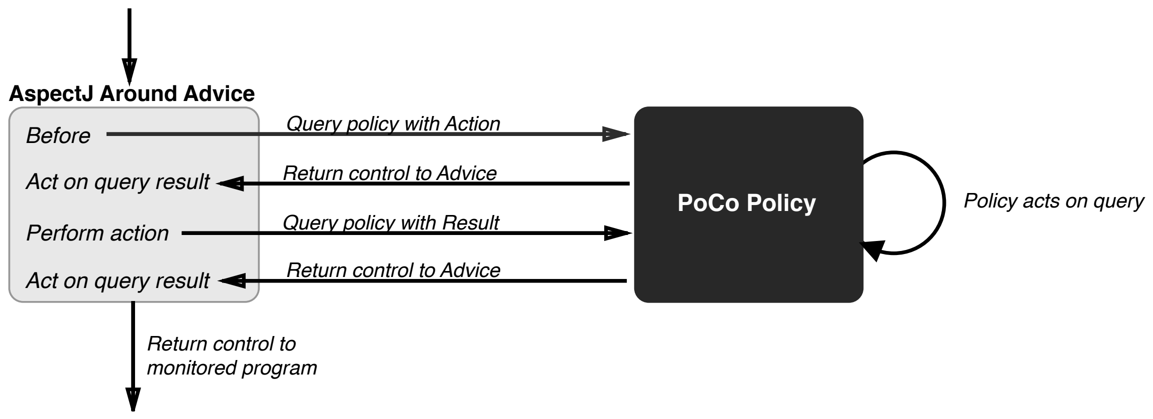


Figure 6.1. Control flow once an AspectJ pointcut is triggered at a security-relevant method.

Figure 6.1 shows how the execution flow of a monitored program is modified once an AspectJ pointcut is triggered. The compiler creates pointcuts for all methods considered security-relevant by its policies. Once one of these methods is called, the AspectJ Advice associated with it executes. Once the advice has control, it queries the PoCo runtime about the attempted action.

Once the PoCo runtime and policies are queried, the following scenarios may occur:

1. The set of permitted actions includes the queried action. In this case, the action is allowed to execute.

2. Neither the set of permitted nor the set of disallowed actions contain the queried action. If the set of permitted actions is finite, an alternate action is performed at random. Otherwise, this is treated as a neutral result and the queried action is performed.

3. The set of disallowed actions includes the queried action, and the set of permitted actions is finite and not empty. In this case the runtime promotes an action at random as in the previous case.

4. The set of disallowed actions includes the queried action, and the set of permitted actions is not finite or empty. In this case, the action is prevented and the program will halt.

5. The sets of permitted and disallowed actions are both empty. This is a neutral result and the monitor will by default allow the queried action to execute.

The following sections will detail the compile-time and runtime aspects of our PoCo implementation.

6.2 Compilation Process

The PoCo compiler inputs one or more PoCo policy files and a compiled Java program to monitor. The compiler outputs an AspectJ file and compiled PoCo runtime that are then input into the AspectJ compiler to instrument the monitored program. This section will focus on the steps involved in creating the AspectJ file while the next section will detail how the PoCo runtime is constructed.

There are three modules that make up the PoCo compiler: the parser, the static analysis scan, and the code generation. The parser was detailed in Chapter 4. The static analysis is a wholly optional module that gives a policy designer feedback about potential ambiguities and undefined behavior within their policies. Because this module was completed independently by another member of the group, it will not be detailed here. The code generation module outputs the AspectJ and Java source files from the provided PoCo policies. The steps taken by the code generation module are as follows:

1. Generate closure. The compiler walks the PoCo parse tree and generates a data structure to hold information about all variable and function declarations within the PoCo policies.

2. Extract pointcuts. The compiler walks the entire PoCo parse tree to acquire signatures of all security-relevant methods for AspectJ pointcut generation.

3. Generate AspectJ file. The AspectJ file contains all pointcuts and advice for instrumenting the monitored program as well as hooks to the PoCo runtime.

4. Generate Java files. The logic of each inputted PoCo policy is reimplemented as a Java class that interacts with the PoCo runtime and AspectJ advice.

Each of these steps will next be described in greater detail.

6.2.1 Closure Generation

PoCo policies declare global and policy-scope variables and functions before the body of a policy begins. These variables, which are bound at runtime, must be considered by the code generation module later in the compilation process. Functions may or may not be parameterized however their bodies are static at compile time.

The process of Closure generation follows similarly to the variable and function parsing performed in the Scanner Tool in Chapter 5. The parse tree of the policy is traversed up to and including the variable and function definitions. Each variable and function's return type, value, and parameters are then saved into the closure. During this step the compiler will also discover and output errors related to duplicate variable and function names.

6.2.2 Pointcut Extraction

AspectJ advice is only inlined into the monitored program at the locations specified by pointcuts, therefore the PoCo compiler has to know at compile-time all methods considered security-relevant by the inputted PoCo policies. To generate a list of the security-relevant methods, the PoCo compiler again uses the same process employed in the Scanner Tool in Chapter 5.

The compiler will step through the parse tree and monitor the `match` structures within each PoCo `exchange` (see Chapter 3). Recall that a PoCo exchange can match either actions, results, or both. Within the `match` structures are regular expressions (called REs in PoCo). The PoCo compiler analyzes and parses these regular expressions to create pointcut signatures compatible with AspectJ. While AspectJ's pointcut notation has some wildcard and variable-length argument list features, it is not as expressive as regular expressions [27]. Special care must be taken by the PoCo compiler to translate the regular expressions into AspectJ pointcut notation without changing the original expression's scope.

Consider a PoCo regular expression that matches events of the form `` `Open(Port|Socket)' ``. This expression could match both a function call to `OpenPort()` and `OpenSocket()`. This would require two separate pointcut declarations in AspectJ. Our compiler uses the same process used in the Scanner Tool in Chapter 5 to bridge the gap between PoCo's expressive syntax and AspectJ. To generate all the pointcut expressions for a single PoCo regular expression, the compiler extracts a list of all method codes from the monitored Java program. The PoCo regular expressions are then matched against this list to generate a mapping from regular expression to actual method calls.

6.2.3 AspectJ Generation

Our compiler outputs a single AspectJ file per monitored program. Recall from Section 2.2 that pointcuts specify where advice will execute in AspectJ. Using the extracted pointcut signatures from

```
1 AllowOnlyMIME():
2 @ports()[!`#int{143|993|25|110|995}']: RE
3 <_ => -`java.net.ServerSocket.<init>($ports)'>*
```

Listing 6.1. AllowOnlyMIME PoCo Policy

the previous section, the compiler outputs all pointcuts in the first part of the AspectJ file. The second part of the file is composed of advice that executes for each pointcut. The advice that is generated will execute before and after each security-relevant method is invoked. Within the advice, our PoCo policy and runtime are queried with information about that specific method invocation. A simple example PoCo policy is presented in Listing 6.1 to illustrate the generation of AspectJ pointcuts and around advice.

Listing 6.1 contains a small PoCo policy that prevents a monitored program from opening sockets to ports other than those specified on Line 2. Looking closely at Line 2, there is a `!` symbol at the start of the function definition, meaning `ports` defines the set of ports excluding the ones listed. Line 3 is the only exchange of the policy and simply returns a negative SRE for all queries. Negative SREs express that a policy wishes to deny an action. Whether or not the action is denied is not known here—the root policy decides what result will ultimately be returned to the runtime.

Listing 6.2 contains the first part of the code generated from the PoCo policy in Listing 6.1. This part contains the generated AspectJ code and excludes the Java code implementing the policy's logic.

Line 7 contains the root policy declaration. For simple situations where a single policy is being compiled, the PoCo compiler will generate a default root policy that forwards the results of the inputted policy (in this case, the AllowOnlyMIME policy).

The AspectJ-specific syntax begins on Line 9 and ends on line 19. Lines 9-10 contains the policy's first and only pointcut definition. Here the pointcut is declared as a `call` pointcut (i.e. our advice is inlined at the call site, not the execution site), on all methods matching `java.net.ServerSocket.new()`, with a single integer parameter.

Line 12 begins the first and only advice. This advice is declared as `around` advice, which means that the code is inlined in such a way that the advice itself absorbs the method call. Inside this advice, we compare the method call's single integer argument (its value is in the variable `value0`)

```
1  import com.poco.PoCoRuntime.*;
2  import java.lang.reflect.Method;
3
4  import java.lang.reflect.Constructor;
5
6  public aspect AspectAllowOnlyMIME {
7     private RootPolicy root = new RootPolicy(new AllowOnlyMIME() );
8
9     pointcut PointCut0(int value0):
10       call(java.net.ServerSocket.new(int)) && args(value0);
11
12    Object around(int value0): PointCut0(value0) {
13       if (RuntimeUtils.StringMatch(new Integer(value0).toString(),
14             "[^143|993|25|110|995]")) {
15          root.queryAction(new Action(thisJoinPoint));
16          return proceed(value0);
17       } else
18          return proceed(value0);
19    }
```

Listing 6.2. AllowOnlyMIME AspectJ code

with set of disallowed values. If it matches, we query the policy in Line 15, otherwise we let the method continue in Line 18.

6.2.4 Java File Generation

The final step of the compilation process involves generating a Java representation of the inputted PoCo policy. The code outputted in this step effectively *rebuilds* the PoCo policy's structure as a collection of Java objects that mimic the PoCo language structure at runtime. The code outputted in this step is fairly simplistic, as the actual behavior of the objects is fleshed out in the PoCo runtime. Only the declared variables, functions, and structure of the inputted policy are translated to Java code, not the functionality. The policy class created in this step extends a policy class that exists in the runtime. This outputted Java file derives much of its functionality from the included PoCo runtime, discussed in the next section.

This AspectJ file contains the blueprint for how we'd like to inline our code in the monitored program. The Java file contains the policy itself. We use the AspectJ compiler to perform the code inlining task. The generated AspectJ and Java files are input along with the monitored program. The AspectJ compiler then weaves the advice from the AspectJ file into the monitored program and outputs a new, instrumented binary.

```
    class AllowOnlyMIME extends Policy {
        public AllowOnlyMIME() {
            try {
                SequentialExecution rootExec = new SequentialExecution("none");
                SequentialExecution exec0 = new SequentialExecution("*");
                Exchange exch0 = new Exchange();
                Match match0 = new Match(
                        "java.net.ServerSocket.new(#int{[^143|993|25|110|995]})"
                );
                exch0.addMatcher(match0);
                SRE sre0 = new SRE(null, null);
                sre0.setNegativeRE(
                        "java.net.ServerSocket.new(#int{[^143|993|25|110|995]})"
                );
                exch0.setSRE(sre0);
                exec0.addChild(exch0);
                exec0.setHasExch(true);
                rootExec.addChild(exec0);
                rootExec.getCurrentChildModifier();
                setRootExecution(rootExec);
            } catch (PoCoException pex) {
                System.out.println(pex.getMessage());
                pex.printStackTrace();
                System.exit(-1);
            }
        }
    }
}
```

Listing 6.3. AllowOnlyMIME Java code

Listing 6.3 contains the generated AspectJ and Java code from the PoCo compiler. Line 21 begins the Java representation of the original PoCo policy. On Line 24 begins the process of recreating the input policy as a collection of PoCo runtime objects. Notice that the flow of the code follows the traversal of a parse tree. The compiler creates runtime objects as they are encountered, fleshes them out, parses objects contained within, and connects the objects together as it navigates back up the tree. The sole execution of the policy is created in Line 25 (the root execution created on Line 24 is an implementation detail), with a `*` modifier indicating that the execution can repeat zero or more times. The policy's sole exchange is built up in Lines 26-35, with a match object and negative SRE created in Lines 27 and 32 respectively. A policy object always needs a root execution and as a result Line 40 is always required in any generated PoCo policy.

While the AspectJ and Java code are presented here in a single file for convenience, all compiled PoCo policies follow the same structure: a series of pointcuts inline advice at security-relevant method calls, and the advice query the PoCo root policy for a decision each time they are triggered.

6.3 PoCo Runtime System

The PoCo runtime is actually three parts: 1) the inlined AspectJ advice, 2) the Java policy class outputted by the compiler, and 3) the PoCo runtime support library. Much of the Java code outputted by the PoCo compiler relies on the PoCo runtime library for its functionality. This support library is static — it does not have to be recreated or recompiled for each new PoCo policy or monitored program. It can be thought of as a packaging of all the PoCo-specific boilerplate code that each policy requires.

PoCo semantics significantly differ from Java's, so our approach relies on a layer of abstraction where PoCo structures (e.g. exchanges, executions, SREs) are created as Java objects that encapsulate PoCo-specific functionality. The PoCo runtime implements abstract, queryable `policy` classes that are forwarded events from the inlined AspectJ advice. The steps involved in querying the PoCo runtime are as follows:

1. When a pointcut is triggered, the advice constructs an `Event` object that contains the method signature, details about its arguments, and whether this is an *Action* or *Result*.

2. The root PoCo policy is queried with the `Event` object from the AspectJ advice.

3. The root PoCo policy queries **all** subpolicies in the DAG, starting with the leaves. Each policy's result is also saved so that each policy is only ever queried once per event.

4. The advice receives the result from the root policy and reacts accordingly, either halting execution or allowing the action to proceed.

The PoCo runtime is simulateneously the most critical part of the PoCo implementation as it provides most of the PoCo functionality in Java, and the most unremarkable as it is simply a Java implementation of the control flows and logic outlined in the PoCo specification [17]. The noteworthy portions of the PoCo implementation are the pieces outlined earlier in this chapter that convert a dynamic and expressive language like PoCo to an AspectJ and Java program.

CHAPTER 7

CONCLUSION AND FUTURE WORK

The PoCo language is unique among existing policy specification languages because of its use of signed regular expressions (SREs) to denote sets of allowed and denied actions. This use of SREs as the output of policies lends itself to predictable policy composition that exhibits some algebraic properties because SREs can be combined with traditional set operations such as union, intersection, and disjunction [17].

My work's central thesis is that an implementation of an expressive, general-purpose policy specification language that uses regular expressions as a means for conveying decisions is possible on the Java platform. My contributions to the PoCo project have made this a reality with a working implementation of the PoCo language compiler and runtime for the Java platform. First, I implemented a fully-functional formal grammar for the PoCo language, allowing us to generate a parser and lexer for the language. Second, I developed a static analysis "Scanner Tool" that provides PoCo policy designers with insight into how their policy will interact with a monitored program's method calls. Finally, I initiated and contributed towards development of the PoCo compiler and runtime on the Java platform.

The completion of this thesis proves the viability of the PoCo project, but there is still more work to be done to take advantage of the language's set-based policy composition. One interesting area of further development of the language is porting it to a popular mobile platform that is based on Java: Android. Porting PoCo to Android would allow smartphone users to feel at ease installing applications that come from untrusted sources, or applications that ask for more permissions than are seemingly necessary. PoCo on Android would also provide invaluable insights into the language's shortcomings and inefficiencies on a mobile platform where inefficient code can quickly cause performance or battery life problems. Having PoCo on such a vibrant and growing

Java-based platform would both benefit those who work on the language with experience and test data, but would also further the language's reach beyond desktop Java.

There are some significant hurdles to implementing PoCo on Android smartphones, some technical and some practical. One technical hurdle is that PoCo requires the original binary of an application to be rewritten with PoCo policy code weaved into it, which will break any cryptographic signatures the original application had. The Android OS is by default restrictive to applications that do not have a valid signature. A practical hurdle is the issue of having to instrument an application with a policy on a computer prior to uploading it to an Android device. There may be the possibility of bundling a PoCo compiler application onto the device itself to bypass this issue.

Another future work item for PoCo is generalization of the language to other platforms outside of the Java ecosystem, such as C# and Microsoft's .NET. Porting PoCo to C# is a logical leap from the Java platform because .NET and Java share many similarities, from syntax to the use of bytecode and a virtual machine. Integrating PoCo with a Microsoft platform would further expand its reach to an extremely vast software ecosystem centered around Windows applications.

It is my feeling that the implementation of the PoCo compiler and runtime is an important step in the language's journey from concept to practical tool. Expanding the language to other platforms while refining and improving its performance will be key to expanding its reach even further. We have shown that PoCo's innovative concepts are viable and can assist policy designers in creating more modular, self-contained policies so that they can be combined in predictable and standard ways without added complexity from language peculiarities.

LIST OF REFERENCES

[1] Yan Albright, Cory Juhlin, Danielle Ferguson, and Donald Ray. *PoCo Compiler and Runtime Source Code.* University of South Florida, 2015. https://github.com/Corjuh/PoCo-Compiler.

[2] Ja'far Alqatawna, Erik Rissanen, and Babak Sadighi. Overriding of access control in XACML. In *Eighth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY '07)*, pages 87–95, June 2007.

[3] Claudio A. Ardagna, Sabrina De Capitani di Vimercati, Gregory Neven, Stefano Paraboschi, Franz-Stefan Preiss, Pierangela Samarati, and Mario Verdicchio. Enabling privacy-preserving credential-based access control with XACML and SAML. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on Computer and Information Technology*, pages 1090–1095, June 2010.

[4] Jayalakshmi Balasubramaniam and Philip W.L. Fong. A white-box policy analysis and its efficient implementation. In *Proceedings of the 18th ACM Symposium on Access Control Models and Technologies*, SACMAT '13, pages 149–160, New York, NY, USA, 2013. ACM.

[5] Lujo Bauer, Jay Ligatti, and David Walker. Composing expressive runtime security policies. *ACM Trans. Softw. Eng. Methodol.*, 18(3):9:1–9:43, June 2009.

[6] Elisa Bertino, Claudio Bettini, Elena Ferrari, and Pierangela Samarati. An access control model supporting periodicity constraints and temporal reasoning. *ACM Transactions on Database Systems*, 23(3):231–285, September 1998.

[7] Piero Bonatti, Sabrina De Capitani di Vimercati, and Pierangela Samarati. An algebra for composing access control policies. *ACM Transactions on Information and Systems Security*, 5(1):1–35, February 2002.

[8] Glenn Bruns and Michael Huth. Access control via belnap logic: Intuitive, expressive, and analyzable policy composition. *ACM Transactions on Information and Systems Security*, 14(1):9:1–9:27, June 2011.

[9] Jery Bryans. Reasoning about XACML policies using CSP. In *Proceedings of the 2005 Workshop on Secure Web Services*, SWS '05, pages 28–35, New York, NY, USA, 2005. ACM.

[10] OW2 Consortium. ASM bytecode engineering library. http://asm.ow2.org, 2015.

[11] Joshua Finnis, Nalin Saigal, Adriana Iamnitchi, and Jay Ligatti. A location-based policy-specification language for mobile devices. *Pervasive and Mobile Computing*, 8(3):402 – 414, 2012.

[12] Philip W.L. Fong. Access control by tracking shallow execution history. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 43–55, May 2004.

[13] Free Software Foundation, Inc. *GNU Bison 3.0.2 Manual*, Oct 2013. https://www.gnu.org/software/bison/manual/bison.html.

[14] Micah Jones and Kevin W. Hamlen. Disambiguating aspect-oriented security policies. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pages 193–204, New York, NY, USA, 2010. ACM.

[15] Cory Juhlin. *PoCo Scanner Tool Implementation*, May 2015. https://github.com/Corjuh/PoCoScanner.

[16] Ninghui Li, Qihua Wang, Wahbeh Qardaji, Elisa Bertino, Prathima Rao, Jorge Lobo, and Dan Lin. Access control policy combining: Theory meets practice. In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*, SACMAT '09, pages 135–144, New York, NY, USA, 2009. ACM.

[17] Daniel Lomsak. *Toward More Composable Software-Security Policies: Tools and Techniques.* PhD thesis, University of South Florida, 2013.

[18] Kenneth C. Louden. *Compiler Construction Principles and Practice.* Cengage Learning, 1997.

[19] Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi. Formalisation and implementation of the XACML access control mechanism. In Gilles Barthe, Benjamin Livshits, and Riccardo Scandariato, editors, *Engineering Secure Software and Systems*, volume 7159 of *Lecture Notes in Computer Science*, pages 60–74. Springer Berlin Heidelberg, 2012.

[20] Till Mossakowski, Michael Drouineaud, and Karsten Sohr. A temporal-logic extension of role-based access control covering dynamic separation of duties. In *Proceedings of the 10th International Symposium on Temporal Representation and Reasoning and Fourth International Conference on Temporal Logic.*, pages 83–90, July 2003.

[21] Qun Ni, Elisa Bertino, and Jorge Lobo. D-algebra for composing access control policy decisions. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pages 298–309, New York, NY, USA, 2009. ACM.

[22] OASIS. *eXtensible Access Control Markup Language (XACML) Version 3.0*, January 2013. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pdf.

[23] Terence Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Bookshelf, 2013.

[24] Terence Parr. ANTLR tool homepage. http://www.antlr.org, May 2015.

[25] François Siewe, Antonio Cau, and Hussein Zedan. A compositional framework for access control policies enforcement. In *Proceedings of the 2003 ACM Workshop on Formal Methods in Security Engineering*, FMSE '03, pages 32–42, New York, NY, USA, 2003. ACM.

[26] Fuminobu Takeyama and Shigeru Chiba. An advice for advice composition in AspectJ. In Benoît Baudry and Eric Wohlstadter, editors, *Software Composition*, volume 6144 of *Lecture Notes in Computer Science*, pages 122–137. Springer Berlin Heidelberg, 2010.

[27] Xerox and Palo Alto Research Center. *The AspectJ Programming Guide*, 2014. http://eclipse.org/aspectj/doc/released/progguide/index.html.