PoliSeer: A Tool for Managing Complex Security Policies*

Daniel Lomsak and Jay Ligatti
University of South Florida
Department of Computer Science and Engineering
{dlomsak, ligatti}@cse.usf.edu

November 25, 2009

Abstract

Complex security policies are difficult to specify, understand, and update. The same is true for complex software in general, but while many software-engineering tools exist for decomposing complex general software into simpler reusable modules (packages, classes, functions, aspects, etc.), few policy-engineering tools exist for decomposing complex security policies into simpler reusable modules. The tools that do exist for modularizing policies either encapsulate entire policies as atomic, indecomposable modules or allow fine-grained modularization but are complicated and lack policy-visualization capabilities.

This paper presents PoliSeer, the first tool we are aware of that allows policy engineers to specify, visualize, modify, and enforce complex policies as compositions of simpler policy modules. We describe the design and implementation of PoliSeer, as well as a case study in which we have bootstrapped PoliSeer by using it to specify and enforce a policy on itself.

1 Introduction

Although complex security policies are difficult to specify, analyze, and update, they arise often in practice. For example, a system administrator or end user may wish to enforce a complex collection of security constraints on an untrusted application to limit its access to resources such as files, memory, and peripheral devices and to obligate it to audit security-relevant operations and employ appropriate cryptographic protocols and intrusion detection on network communications. In general, software-security policies tend to become more and more complex over time, due to the emergence of new attacks, users' demands for relaxations to overly tight policy constraints, and the development of new application areas, like medical databases, which require domain-specific security and privacy considerations [2].

1.1 Related Work

The trend of increasing complexity in software-security policies mirrors the trend of increasing complexity in general software applications; however, many tools and techniques exist to help software engineers specify, analyze, and update complex software applications. One of the most common techniques is modularization; engineers can modularize software into independent, reusable, individually testable components (packages, classes, functions, aspects, etc.) that can be parameterized by, and can communicate with, other

*USF technical report CSE-SSec-112509

components through well-defined interfaces. Decomposing complex software into simpler modules saves engineers from having to manage software as a single, indecomposable code block. Integrated development environments (IDEs) for software engineering typically provide good support for navigating and visualizing software modules [7, 19, 20, 22].

In contrast, we are not aware of any existing policy-engineering tools that enable users to navigate and visualize arbitrary policy modules, or even to build arbitrary policies straightforwardly as compositions of existing policy modules. Many policy-engineering tools do exist for specifying and managing—without visualizing—arbitrary but uncomposable policies (e.g., [14, 13, 6, 8, 9, 18, 21, 10]). Many other tools exist for specifying, visualizing, analyzing, and/or composing policies, but only in particular domains; e.g., Policy Visualization Analysis tool provides a GUI for managing SE Linux policies [23], Expandable Grid manages access-control policies [17], Policy Mapper manages location-based access-control policies [4], front ends for SPARCLE and PERMIS manage natural-language access-control and privacy policies [5, 11], and Fang and Firmato manage firewall policies [16, 1]. None of these tools allow policy engineers to specify, manipulate, compose, visualize, or enforce arbitrary (including cross-domain) policies.

Another related tool is Polymer, which does allow cross-domain policies to be specified and enforced as compositions of simpler policy modules [2]. Polymer users specify runtime policies on Java-bytecode applications in the Polymer programming language. Polymer policies exhibit *universal composability*; every policy can be composed arbitrarily with other policies. Polymer achieves universal composability by (1) making all policies first-class objects (i.e., objects that are treated like all other values, which can be passed as arguments to and returned as results from methods) and (2) requiring all policy objects to implement a standard interface. A Polymer policy P can be parameterized by another policy P', and when P has to decide whether and how to allow a security-relevant application event A to occur, P may query P' for a response to A and use that response to generate its own response. For example, a `Conjunction` policy might be parameterized by two policies P_1 and P_2; the overall policy can enforce the conjunction of P_1 and P_2 by always responding to security-relevant events with the most restrictive of the responses of P_1 and P_2. In this case `Conjunction` is a *superpolicy*, and P_1 and P_2 are *subpolicies*. As another example, an `Audit` superpolicy may be parameterized by a policy P and a string S; then `Audit` can blindly enforce P while logging all security-relevant events to file S. In this way, policy engineers can build arbitrary, complex policies as compositions of simpler subpolicies.

However, Polymer lacks policy-visualization capabilities, and policy engineers cannot specify, analyze, or update policies straightforwardly in Polymer due to its complicated static and dynamic semantics. The complications stem from Polymer's safeguards to ensure universal composability; policies must segregate effects (observable state updates and I/O operations) out of `query` methods and into `accept` and `result` methods. Segregating effects makes for complicated control flow between policy methods, and policy engineers must understand this control flow in order to specify, analyze, and update policies correctly. For example, Figure 1 contains an email-client policy with `query`, `accept`, and `result` methods, as specified in Polymer, while Figures 7 and 8 show how the same policy could be straightforwardly specified and visualized in PoliSeer.

1.2 Contributions

We present PoliSeer, the first tool we know of that enables complex policies to be straightforwardly specified, visualized, modified, and enforced as compositions of simpler subpolicy modules. PoliSeer users can import universally composable policies from a policy library, compose them in meaningful ways by declaring arguments to policies, visualize and update policies holistically, and generate code to enforce the composed policies. Our prototype implementation of PoliSeer uses Polymer as an underlying language of universally

```
package examples.mail;
import polymer.*;
import examples.*;
public class Email extends Policy {
  private Policy p;
  public Email(){
    p = new Audit(new Conjunction(new Dominates(new Dominates(new Reflection(),
          new Conjunction(new ClassLoaders(), new NoOpenClassFiles())),
          new Dominates(new DisSysCalls(), new InterruptToCheckMem(10.0, 4000))),
          new IsClientSigned(new Trivial(), new Dominates(new Dominates(new TryWith(new ConfirmAndAllowOnlyHTTP(),
          new AllowOnlyMIME()), new Attachments()), new Conjunction(new OutgoingMail("PoliSeer@cse.usf.edu"),
          new AllowInsertedActions(new IncomingMail())))))), "email.log");
  }
  public Suggestion query(Action a) { return p.query(a); }
  public void accept(Suggestion s) { p.accept(s); }
  public void handleResult(Suggestion s, Object result, boolean wasExnThn) {
    p.handleResult(s, result, wasExnThn);
} }
```

Figure 1: Email-client policy specified in Polymer (taken from [3]). The same policy can be specified and visualized in PoliSeer as shown in Figures 7 and 8.

composable policies (i.e., our PoliSeer implementation imports and exports Polymer policies); however, we have designed PoliSeer to be readily portable to any other system with universally composable (first-class and parameterized) policies.

Roadmap We proceed as follows. Section 2 describes the PoliSeer interface; Section 3 explains and evaluates our implementation of PoliSeer; Section 4 reports our experiences designing and implementing a case-study policy in PoliSeer; and Section 5 concludes.

2 The PoliSeer Interface

PoliSeer provides a straightforward interface for creating, visualizing, modifying, and enforcing complex policies.

2.1 The Main Window

The main PoliSeer window consists of two panels, as shown in Figure 2. The left is the *policy-selector panel*; the right is the *policy-tree panel*.

- The policy-selector panel allows users to navigate the file system to find existing composable policies. Our implementation begins by populating the policy-selector panel with all subdirectories and `.poly` files (i.e., Polymer policy files) in the user's home directory. The policy-selector panel uses a standard interface for navigating the file system: clickable areas expand and contract subdirectories. When a user expands a subdirectory, PoliSeer searches for, parses, and displays all policy files in the newly visible directory. PoliSeer parses the policy files so it can display the types of parameters each policy expects (in its constructor) next to that policy's name in the policy-selector panel, as shown in Figure 2 (when multiple constructors exist for the same policy, users select the desired constructor from a drop-down list before inserting the policy into a policy tree). Because computer users are accustomed to this sort of expand-and-contract navigation interface (e.g., the Windows file explorer and many application programs employ the same interface), navigating policy libraries is straightforward.

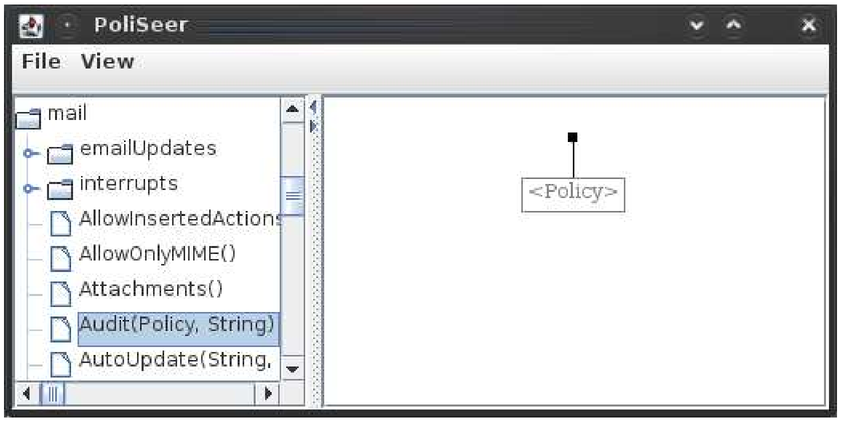


Figure 2: Main PoliSeer window divided into policy-selector and policy-tree panels. The policy-tree panel is displaying the default, empty policy.

- The policy-tree panel contains a visualization of the current policy. When PoliSeer begins executing, it displays the empty policy as shown in Figure 2. The empty policy consists of a single grayed-out node containing the text `<Policy>`, which indicates that PoliSeer expects that node to be filled in with a policy. In general, grayed-out nodes in a PoliSeer policy indicate incompletions in the policy; the text in a grayed-out node indicates the type of data that must be inserted into that node. In this way, PoliSeer communicates to the user whether, and in what ways, policies are incomplete. For example, Figure 3 shows a policy-tree panel for an incomplete, one-node `Audit` policy parameterized by another `Policy` and a `String`; the policy is incomplete until the user specifies a `Policy` and a `String` argument for `Audit`.

The only windows incorporated into PoliSeer besides the split main window are modal popups (for routine operations like selecting a location to load a policy from or save a policy to) and a window for viewing policy documentation and source code (described in Section 2.3).

2.2 Creating Policies

PoliSeer's basic interface for creating policies is simple. Users may select a policy in the policy-selector panel by left-clicking on the policy name. Having selected a policy P, the user may left-click on any *landing area* L in the policy-tree panel to insert P into L. Valid landing areas are grayed-out `<Policy>` nodes and *branch-insertion points* (BIPs) in the policy-tree panel. PoliSeer automatically displays BIPs as small black squares in the policy-tree panel on every branch into which a user could possibly insert a policy.

For example, Figure 2 shows PoliSeer as it begins, with an empty policy-tree panel. A user may add the `Audit` policy as the root of the policy tree by clicking on the `Audit` policy in the policy-selector panel and then clicking on the grayed-out `Policy` node in the policy-tree panel. The policy tree in Figure 3 results from this addition; `Audit` has been added as the root node of the policy, but two new grayed-out nodes have appeared because PoliSeer has parsed the `Audit` policy and determined that it is parameterized by another `Policy` and a `String`. The user may then insert a `String` as the right child of the `Audit` policy by clicking on the grayed-out `String` node and entering the string in a pop-up window, as shown in Figure 4. Users may enter other types of arguments to policies, such as `int`s, `float`s, `boolean`s, and `char`s, similarly to `String`s, but PoliSeer will first confirm that the user's entry can be tokenized as a literal of the correct

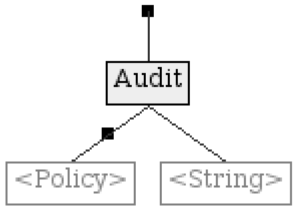


Figure 3: Policy-tree panel showing a root `Audit` policy parameterized by another `Policy` and a `String`, though no children have yet been specified.

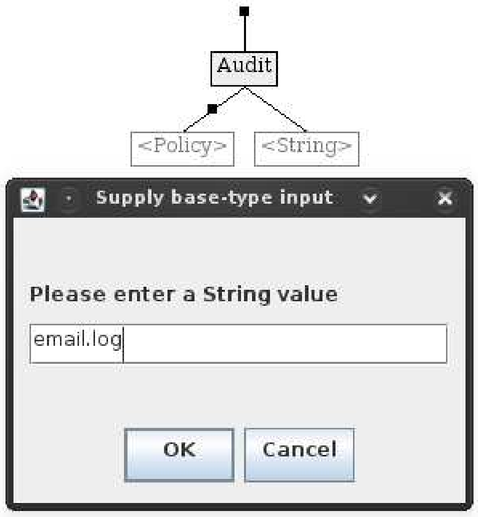


Figure 4: Policy-tree panel as the user enters a `String` argument for the `Audit` policy.

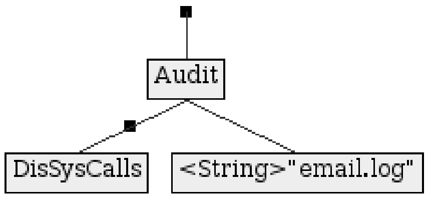


Figure 5: Policy-tree panel showing an `Audit` policy with subpolicy and string arguments. This complete policy disallows system calls (i.e., `java.lang.Runtime.exec` methods) at runtime while logging all policy decisions to a file named `email.log`.

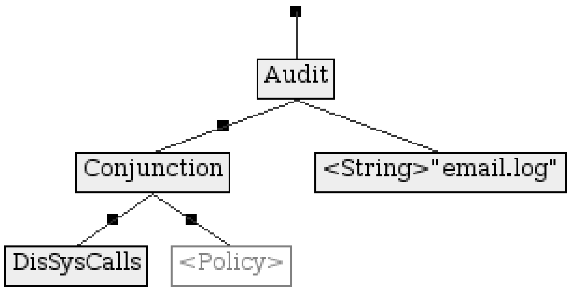


Figure 6: The same policy-tree panel shown in Figure 5, except that the user has now inserted a `Conjunction` policy between the `DisSysCalls` and `Audit` policies.

type. Currently, PoliSeer can only import and manipulate policies with primitive-type, `String`, and `Policy` parameters, but all the policies provided in the standard Polymer distribution [3] satisfy this constraint.

Continuing with this example, Figure 5 shows a complete policy tree that results from inserting a (childless) policy and a string into the grayed-out nodes of Figure 3. Two BIPs exist in Figure 5; a user may insert a policy node into this policy above the `Audit` root or above the `DisSysCalls` child of `Audit`. To insert a `Conjunction` policy between the `Audit` and `DisSysCalls` nodes, the user simply clicks on the `Conjunction` policy in the policy-selector panel and then clicks on the BIP between the `Audit` and `DisSysCalls` policies in Figure 5; the result is shown in Figure 6.

Having created a (complete or incomplete) PoliSeer policy, a user may save it to a `.psr` file (which is simply a serialization of the policy tree) with the `File -> Save Tree` menu option and may generate Polymer code to enforce the policy in a `.poly` file with the `File -> Generate Policy Code` option. Conversely, users may resume creating, viewing, or modifying a saved `.psr` policy with the `File -> Load Tree` option. When exporting an incomplete PoliSeer policy to a `.poly` file, PoliSeer automatically parameterizes the exported policy by all missing policy components (e.g., if the PoliSeer policy is complete except for missing one child of a `Conjunction` superpolicy, then the exported policy will have a single parameter, a policy that fills in for the missing child of `Conjunction`).

2.3 Visualizing Policies

As Figures 7–9 demonstrate, PoliSeer's policy-tree panel can provide a useful high-level visualization of complex policies decomposed into subpolicy modules. If PoliSeer's view of a policy tree is too high level, users can always choose the `View -> Policy Source` menu option to obtain the source-code-level details of the most recently selected policy. Examining a policy's source-code documentation can be helpful for PoliSeer users when figuring out which arguments to specify for that policy. For example, a PoliSeer user may see and be intrigued by the `Audit` policy, view the documentation in the `Audit` source code to understand what the policy does and which arguments it expects, and then include `Audit` in the policy tree.

As another aid for visualizing policy compositions, PoliSeer provides a toggleable menu option `View -> Show Non-Policy Nodes`. This option removes (or restores) non-policy nodes in the policy-tree panel. Removing non-policy nodes from a policy tree may simplify the user's view of a policy, as Figures 7 and 8 demonstrate. Non-policy nodes often clutter a policy tree without providing much insight into the policy's organization. For example, non-policy nodes may specify port-number, IP-address, or filename arguments to policies, which may be irrelevant for understanding the overall policy structure.

2.4 Modifying Policies

PoliSeer users may modify policy trees in three ways:

1. Users may swap sibling nodes (and their subtrees) by dragging and dropping one sibling node onto another. Swapping subpolicies can be useful when dealing with superpolicies that make semantic distinctions between the order of their children (e.g., Polymer's `Dominates` superpolicy gives priority to its left child [2]). Nodes must have the same type to be swapped, and PoliSeer currently does not support non-sibling node swapping due to policy-tree circularities that arise when swapping a node (and its subtree) with one of its descendants.

2. Users may replace a policy node P in the policy tree with a policy P' by left-clicking P' in the policy-selector panel and then left-clicking on P in the policy tree. PoliSeer only allows P' to replace P when the parameter types of P and P' are *well aligned*, meaning that PoliSeer must be able to assign each of P's nonempty children to be children of P' without introducing any type conflicts. If P and P' are well aligned, PoliSeer performs the replacement by making the P node be a P' node and then traversing P's children from left to right and reassigning each nonempty child of P to the leftmost child of P' with the same type. In this way, PoliSeer attempts to allow policy replacement in all cases in which it could possibly make sense.

3. Users may delete a policy-tree node by right-clicking on it. Before deleting any nonempty policy-tree node, PoliSeer confirms the deletion with a popup dialog box. When deleting a policy node N, the leftmost policy-child of N takes the place of N in the policy tree, and PoliSeer discards all other children of N. Because users might not expect this deletion semantics, PoliSeer highlights all the about-to-be-discarded nodes and displays a confirmation window before actually discarding the highlighted nodes, as shown in Figure 9.

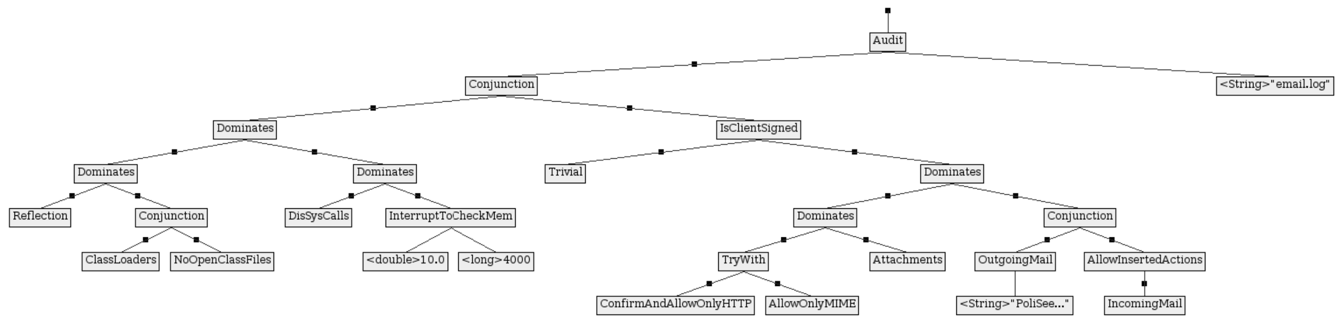


Figure 7: Full tree for an example email policy (taken from [2]).

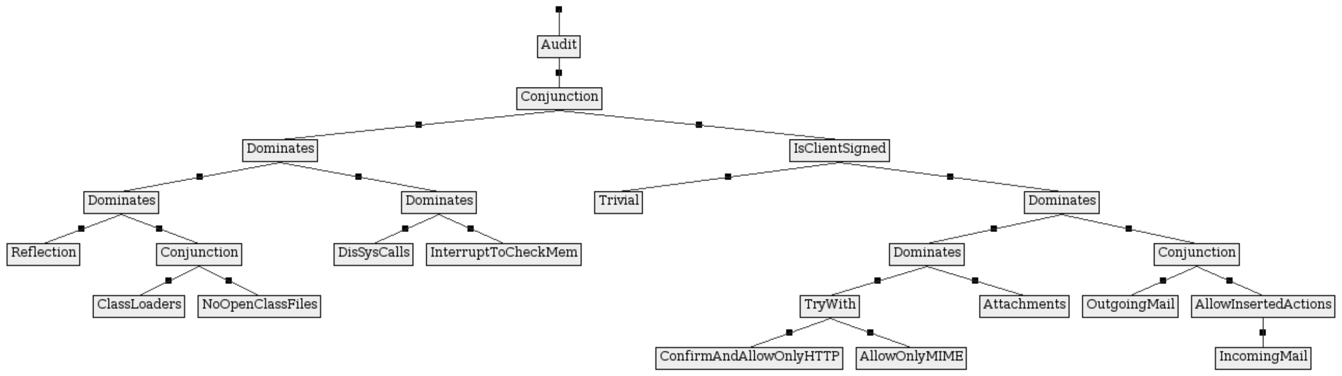
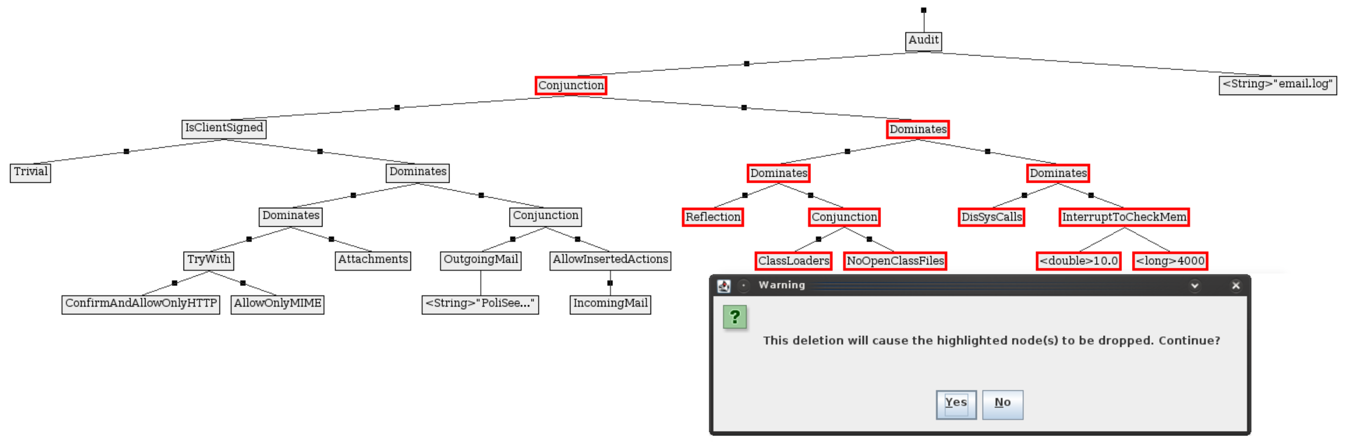


Figure 8: The same policy tree shown in Figure 7 but simplified by hiding non-policy nodes.



This deletion will cause the highlighted node(s) to be dropped. Continue?

Figure 9: Warning displayed before deleting a node with multiple policy children. PoliSeer has highlighted the descendants that will be discarded along with the node being deleted.

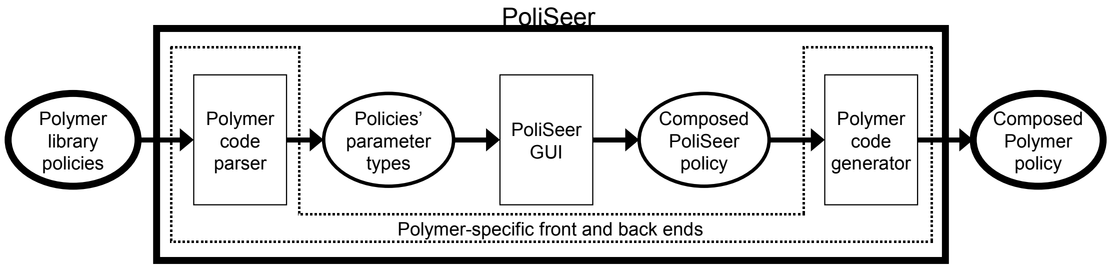


Figure 10: Architectural overview of PoliSeer.

When combined with the ability to insert policy nodes into any landing area in a policy tree, these three operations provide users a complete palette of basic policy-specification, -visualization, and -modification tools.

3 Implementation

We have implemented PoliSeer as an open-source Java application [15]. The implementation is 3351 lines of code in 12 source-code files.

3.1 Architectural Overview

Our PoliSeer implementation consists of three high-level modules:

1. The front end, which reads and parses Polymer files for input into PoliSeer. When a user opens a directory in the policy-selector panel, PoliSeer parses the Polymer files in that directory to determine which types of arguments the policies' constructors expect to receive. PoliSeer uses this type information to prepare grayed-out children in the policy tree, as discussed in Section 2.2. Our front end parses Polymer files with a parser generated by JavaCC, a recursive-descent parser generator [12].

2. The PoliSeer GUI, which contains all the code to implement PoliSeer's interface.

3. The back end, which generates Polymer code for the policy tree being viewed. Users can input the code that this module generates directly into the Polymer system to enforce the specified policy on untrusted Java-bytecode applications.

Figure 10 summarizes PoliSeer's implementation architecture. The Polymer-dependent front and back ends are distinctly separated from the Polymer-independent GUI, so developers can change PoliSeer's underlying policy-specification language from Polymer to a language L by writing and plugging in new front and back ends for L.

3.2 Performance

We have tested our implementation's performance on a Sony Vaio laptop with Intel Core2 Duo 1.73GHz CPUs and 1GB of RAM, running Kubuntu 8.10. For all tests, we report running times obtained by averaging real execution times over ten executions.

Our first test measured the time taken for PoliSeer to start up, build its GUI, and exit at the end of PoliSeer's `main` method. This time included the virtual-machine start-up time and was just 830ms on average.

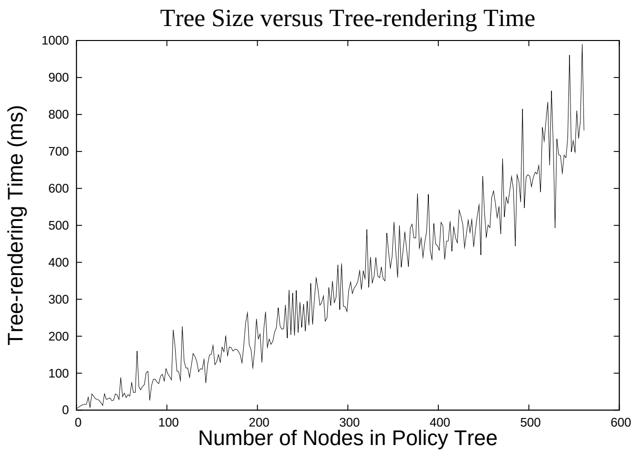


Figure 11: PoliSeer performance rendering policy trees during node insertion.

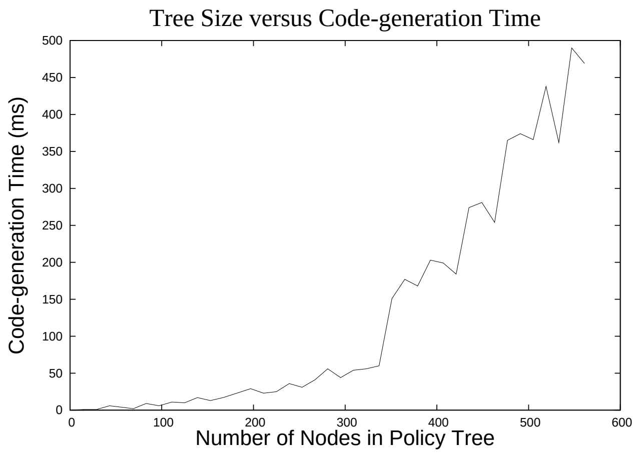


Figure 12: PoliSeer performance generating Polymer code.

Next, we measured the time taken for PoliSeer to parse the Polymer files in the policy library to determine the types of parameters in the policies' constructors. With an average Polymer-policy-file size of 84.5 lines of code, PoliSeer parsed the Polymer files in only 3.1ms on average.

Our third test measured the amount of time PoliSeer took to render a policy tree during insertion of nodes into the tree. This rendering time dominates the amount of time it takes for PoliSeer to insert new nodes into policy trees; node-insertion time is O($\lg n$) for finding where to modify the tree data structure, O(1) for modifying the tree at that point, and O(n) for rendering the new tree with the inserted node (where n is the number of nodes in the policy tree). Figure 11 confirms the linear growth of policy-tree rendering. All tree-rendering-intensive operations in PoliSeer (i.e., node insertion, swapping, replacement, and deletion) exhibit performance similar to that shown in Figure 11. Although tree-rendering times never exceeded one second, even for trees with hundreds of nodes, a good optimization to consider in the future would be to only re-render modified portions of trees.

Finally, we measured the time taken for PoliSeer to generate Polymer code files from policy trees. We implement code generation by (recursively) traversing the policy tree with a preorder traversal, while concatenating strings to construct every policy-tree node. As Figure 12 illustrates, PoliSeer's code-generation time remains low (less than a second) even for policies with many hundred nodes.

In summary, all the basic PoliSeer operations have tolerable performance, and performance delays do not even become noticeable until the user manipulates policy trees containing several hundred nodes.

4 Case study

We have designed and enforced a complex case-study PoliSeer policy that restricts the runtime behavior of PoliSeer itself; that is, we have bootstrapped PoliSeer.

4.1 Policy Overview

Figure 13 displays our case-study policy tree. This policy has a `Conjunction` as its root, so it constrains the untrusted application (PoliSeer in this case) by always responding to a security-relevant action with the most restrictive of its two subpolicies' responses to the same action.

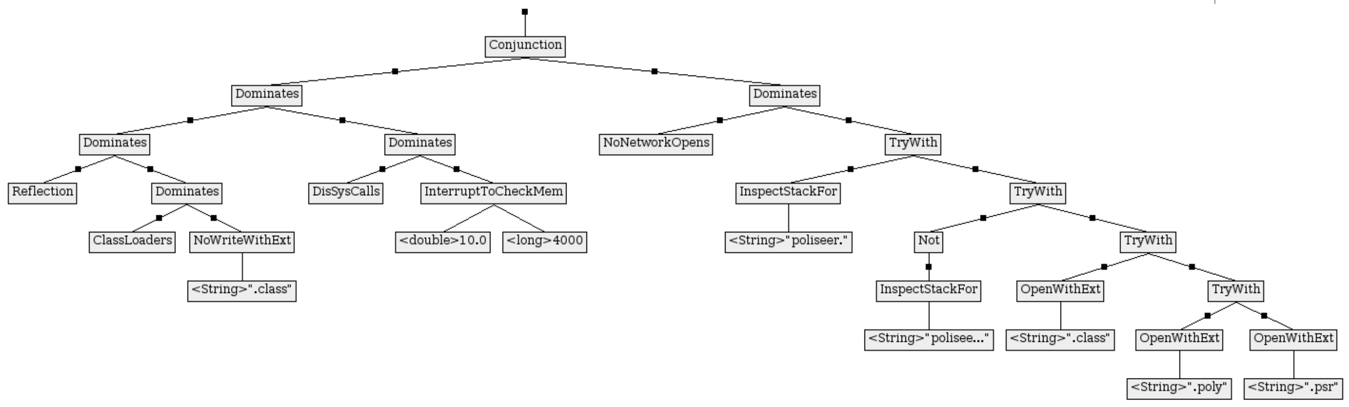


Figure 13: Case-study policy, specified in PoliSeer, that constrains the PoliSeer application itself.

The left high-level subpolicy contains standard policies that should be included in all Polymer policy trees to prevent untrusted applications from using reflection, constructing class loaders, writing `.class` files, or invoking system-level functions (with `java.lang.Runtime.exec` methods) [2]. This branch also includes the `InterruptToCheckMem` policy, which notifies the user if the virtual machine's memory consumption exceeds a specified threshold. All of these subpolicies are conjoined by `Dominates` superpolicies, which act as short-circuit `Conjunction` policies in our case study.

The right high-level subpolicy specifies constraints that we particularly wanted to enforce on PoliSeer; something would be wrong if PoliSeer violated any of these constraints. We call this branch of policies the PoliSeer-specific policy. Like the entire case-study policy, the PoliSeer-specific policy decomposes into two branches, joined with a `Dominates` superpolicy (again acting as a short-circuit `Conjunction`).

The left branch of the PoliSeer-specific policy is the `NoNetworkOpens` policy, which disallows the untrusted application from opening any network sockets. The right branch of the PoliSeer-specific policy restricts the PoliSeer application (i.e., code in the `poliseer` package)—but not the case-study policy we are enforcing on PoliSeer (i.e., code in the `poliseer.policy` package)—from opening files with extensions other than `.psr`, `.poly`, or `.class`. Intuitively, although the case-study policy may open other types of files (perhaps because, e.g., we later extend the policy with auditing capabilities that necessitate opening log files), the PoliSeer application itself should have no effect on the file system except for reading and writing PoliSeer (`.psr`), Polymer (`.poly`), and Java-bytecode (`.class`) files. The application will actually not be able to *write* `.class` files (because the `NoWriteWithExt` policy already disallows it), but the case-study policy does allow the application to *read* `.class` files (because Java compilers often optimize benign operations like initializing nested classes by having the outer class's initializer read the bytecode of the inner class).

The right branch of the PoliSeer-specific policy contains four subpolicies:

1. `TryWith` combines two subpolicies by first obtaining its left child's response to the security-relevant action A that the untrusted application is attempting to execute. If the left child allows A to execute unconditionally then so does the `TryWith` superpolicy; otherwise, `TryWith` responds to A with whatever response its right child returns for A.

2. `InspectStackFor` takes a `String` argument S and inspects the runtime call stack for a method called from package S. More specifically, the policy traverses the call stack from the most recent to the oldest method invocation and disallows the security-relevant action the untrusted application is

about to execute if and only if the traversal reaches a call from S before reaching a `doPrivileged` call.

3. `Not` inverts the response of its subpolicy. For example, if the subpolicy responds to a security-relevant action by halting, the `Not` superpolicy will respond by allowing the action.

4. `OpenWithExt` takes a `String` argument S and allows file-open actions to execute if and only if they access files with names that end with file-extension S.

The case-study policy enforces the desired file-open subpolicy by chaining together `TryWith` combinators. Every child of the `TryWith` policies allows one type of file-open action to execute; a file open is only disallowed when none of those children allow it. In turn, the `TryWith` children allow actions from outside the `poliseer` package, actions from within the `poliseer.policy` package (technically, from `Not` outside the `poliseer.policy` package), actions that open `.class` files, actions that open `.poly` files, and actions that open `.psr` files.

4.2 PoliSeer Performance While Being Monitored

We measured the performance of PoliSeer executing in the Polymer system while enforcing the case-study policy. The measurements provide some insight into the runtime overhead induced by enforcing Polymer policies.

In general, runtime-policy-enforcement overheads depend on the complexity of the policy being enforced and the number of times that policy must respond to security-relevant actions. One important consideration in this regard is that our case-study policy only considers security relevant methods that open files, open network sockets, make system-level calls, implement reflection, or construct class loaders. Enforcing the case-study policy induces runtime overhead only when one of these security-relevant methods gets invoked. Therefore, operations like inserting nodes into policy trees, which do not execute security-relevant methods, run equally quickly regardless of whether the case-study policy is being enforced.

The average time for PoliSeer to start up in the Polymer system was 3.7s, much higher than the 0.83s start-up time we measured when the case-study policy was not being enforced. This significant start-up overhead is common in Polymer due to the large number of files that get opened as the virtual machine loads classes during application startup (recall that our policy considers file openings security relevant).

Besides the start-up overhead, none of the overheads induced by enforcing the case-study policy on PoliSeer were noticeable. The average time to parse Polymer files increased by 0.8ms, and the average time to generate Polymer code increased by 1.3ms (regardless of the policy size), when enforcing the case-study policy. The case-study policy has to respond once to each Polymer-file parsing and code-generation operation because these operations each entail one security-relevant file-open action. Hence, we infer that enforcing the case-study policy only added about 1ms to the execution time of every security-relevant action.

4.3 Experiential Observations

Designing, specifying, and enforcing the case-study policy was a valuable experience for us. We found that PoliSeer made it convenient—but not a foolproof process—to specify and enforce the case-study policy.

Summary of Experiences Designing Policies

Policy modularization was immensely helpful for designing the case-study policy because it enabled us to decompose high-level policy goals into simpler subgoals. For instance, we wished to enforce that PoliSeer code could only open `.psr`, `.poly`, and `.class` files. It was natural for us to decompose this high-level goal into several subgoals, and specifying each subgoal in PoliSeer required just a few mouse clicks (to incorporate `TryWith` and `OpenWithExt` policies). This was a convenient way to design and specify a complex policy.

It is important when designing policies in PoliSeer to have a full and robust policy library available, as users who are unfamiliar with the underlying policy language will have difficulty constructing and importing their own (Polymer) policies. Although PoliSeer by itself is a Turing-incomplete policy-specification tool and lacks the expressiveness of Turing-complete languages like Polymer, PoliSeer users can always specify and import Polymer policies when greater expressiveness is required. Fortunately, existing policies comprise the vast majority of the policies we have constructed. For example, the `Conjunction`, `Dominates`, `TryWith`, and `Not` superpolicies enable subpolicies to be composed in a rich variety of ways. Other common policies perform auditing or enforce access controls on network sockets, files, and other objects. We implemented the entire case study with only a standard library of Polymer policies, though we did have to add five policies (`Not`, `InspectStackFor`, `NoNetworkOpens`, `NoWriteWithExt` and `OpenWithExt`) to the nascent Polymer library before specifying the case-study policy in PoliSeer. We believe each of these five policies is generic and makes a useful addition to the standard Polymer library.

Similarly, it was important for the Polymer policies we chose to include in our case study to be well documented, so we could understand their high-level semantics during composition. We often found ourselves examining policy documentation because a policy's name alone did not provide enough information to understand which arguments the policy expected. In all cases, though, we obtained a sufficient understanding of a policy's semantics by quickly opening and reading its documentation in PoliSeer.

Summary of Experiences Using PoliSeer

Overall we found the PoliSeer interface a great aid for policy specification and visualization. Nonetheless, a couple limitations become apparent during the case study. First, PoliSeer lacks a way for users to select an entire subtree of nodes. Subtree selection would be useful for flexibly moving, replacing, or deleting entire subtrees. Subtree selection could also make it convenient to generate policy code for just one subtree within the overall policy-tree panel; for example, a user could modularize and simplify one part of a complex policy by selecting a subtree of the complex policy, naming that subtree P, generating code for P, and then replacing the selected subtree with P.

Another difficulty related to scaling and whitespace in the policy-tree panel. We found that the default policy-tree scaling made the organization of small- and medium-sized policies clear, but it became more difficult to get a good high-level view of policies as they grew larger. A potential improvement to the PoliSeer GUI would be a mechanism for scaling policy trees and/or the whitespace between nodes in policy trees. Such scaling would enable users to view more policy-tree information on one screen without scrolling; this idea is similar to PoliSeer's existing ability to toggle between showing and hiding non-policy nodes.

5 Conclusions and Future Work

PoliSeer is the first tool we know of to allow policy engineers to specify, visualize, modify, and enforce complex policies as arbitrary compositions of simpler subpolicies. We have successfully used PoliSeer to design and implement complex email-client and bootstrap policies (Figures 7 and 13). In our experience, PoliSeer has been a great aid for quickly specifying, visualizing, and generating code to enforce complex policies. We view PoliSeer as an IDE for security policies, providing policy engineers the same sorts of benefits that traditional IDEs provide software engineers (convenience of creating high-level specifications and visualizations to minimize errors in, or totally avoid, low-level programming tasks).

There are many opportunities for future extensions to our prototype PoliSeer implementation. For example, we would like to consider additions to PoliSeer's interface for:

- Selecting groups of nodes to enable moving, saving, replacing, and deleting entire subtrees of policies
- Scaling the whitespace between nodes in a policy tree, or even scaling the size of the entire policy tree
- Commenting on parts of policy trees, for example, to allow the user to draw a border around a group of nodes, possibly shade the space within that border, and add textual comments to document the purpose and behavior of nodes within that border

We hope that with continued research and development, policy IDEs will be as helpful for managing complex security constraints as standard IDEs have become for managing complex software.

Acknowledgments This research was funded in part by National Science Foundation grants CNS-0742736 and CNS-0716343.

References

[1] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. *ACM Trans. Comput. Syst.*, 22(4):381–420, 2004.

[2] L. Bauer, J. Ligatti, and D. Walker. Composing expressive run-time security policies. *ACM Transactions on Software Engineering and Methodology*, 18(3):1–43, 2009.

[3] L. Bauer, J. Ligatti, and D. Walker. Polymer: A language for composing run-time security policies, 2008. `http://www.cs.princeton.edu/sip/projects/polymer/`.

[4] R. Bhatti, M. Damiani, D. Bettis, and E. Bertino. Policy mapper: Administering location-based access-control policies. *Internet Computing, IEEE*, 12(2):38–45, March-April 2008.

[5] C. A. Brodie, C.-M. Karat, and J. Karat. An empirical study of natural language parsing of privacy policy rules using the SPARCLE policy workbench. In *Proceedings of the second symposium on Usable privacy and security*, pages 8–19, 2006.

[6] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. *Lecture Notes in Computer Science*, 1995:18–39, 2001.

[7] S. Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software.* Springer-Verlag, Berlin, 2007.

[8] Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2000.

[9] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Security and Privacy*, 1999.

[10] K. Havelund and G. Roşu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):158–173, 2004.

[11] P. Inglesant, M. A. Sasse, D. Chadwick, and L. L. Shi. Expressions of expertness: the virtuous circle of natural language for access control policy specification. In *Proceedings of the 4th symposium on Usable privacy and security*, pages 77–88, 2008.

[12] JavaCC, 2008. `https://javacc.dev.java.net/`.

[13] C. Jeffery, W. Zhou, K. Templer, and M. Brazell. A lightweight architecture for program execution monitoring. In *Program Analysis for Software Tools and Engineering (PASTE)*, 1998.

[14] Y. Liao and D. Cohen. A specificational approach to high level program monitoring and measuring. *IEEE Trans. Softw. Eng.*, 18(11):969–978, 1992.

[15] D. Lomsak and J. Ligatti. *PoliSeer*, 2009. `http://www.cse.usf.edu/~ligatti/projects/poliseer/`.

[16] A. Mayer, A. Wool, and E. Ziskind. Fang: a firewall analysis engine. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 177–187, 2000.

[17] R. W. Reeder, L. Bauer, L. Cranor, M. K. Reiter, K. Bacon, K. How, and H. Strong. Expandable grids for visualizing and authoring computer security policies. In *CHI 2008: Conference on Human Factors in Computing Systems*, pages 1473–1482, Apr. 2008.

[18] W. Robinson. Monitoring software requirements using instrumented code. In *HICSS '02: Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9*, 2002.

[19] N. Saigal and J. Ligatti. Inline Visualization of Concerns. In *Proceedings of the International Conference on Software Engineering Research, Management, and Applications (SERA)*, Dec. 2009.

[20] T. Schäfer, M. Eichberg, M. Haupt, and M. Mezini. The SEXTANT software exploration tool. *IEEE Transactions on Software Engineering*, 32(9):753–768, 2006.

[21] K. Sen, A. Vardhan, G. Agha, and G. Rosu. Efficient decentralized monitoring of safety in distributed systems. In *26th International Conference on Software Engineering (ICSE'04)*, pages 418–427, 2004.

[22] M. Shonle, J. Neddenriep, and W. Griswold. AspectBrowser for eclipse: A case study in plug-in retargeting. In *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, 2004.

[23] W. Xu, M. Shehab, and G.-J. Ahn. Visualization based policy analysis: case study in selinux. In *SACMAT '08: Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 165–174, New York, NY, USA, 2008. ACM.