IPSJ Journal Vol. 52 No. 7 2126–2140 (July 2011)

Regular Paper

PoliSeer: A Tool for Managing Complex Security Policies

Daniel Lomsak[†1] and Jay Ligatti[†1]

Complex software-security policies are difficult to specify, understand, and update. The same is true for complex software in general, but while many tools and techniques exist for decomposing complex general software into simpler reusable modules (packages, classes, functions, aspects, etc.), few 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 modules that cannot be decomposed or allow fine-grained policy modularization but require expertise to use correctly. This paper presents PoliSeer, a GUI-based tool designed to enable users who are not expert policy engineers to flexibly specify, visualize, modify, and enforce complex runtime policies on untrusted software. PoliSeer users rely on expert policy engineers to specify universally composable policy modules; PoliSeer users then build complex policies by composing those expert-written modules. This paper describes the design and implementation of PoliSeer and a case study in which we have used PoliSeer to specify and enforce a policy on PoliSeer itself.

1. Introduction

Although complex software-security policies are difficult to specify, understand, and update, they arise often in practice. For example, a system administrator or end user may wish to enforce a complex collection of constraints (i.e., a policy) on an untrusted application to limit that application's access to resources such as files, memory, and peripheral devices, and to obligate the untrusted application to audit security-relevant operations and employ appropriate cryptographic protocols 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 Department of Computer Science and Engineering, University of South Florida

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 modify complex software applications. One of the most common techniques is modularization; engineers can modularize software into independent, reusable components (e.g., packages, classes, functions, aspects, etc.) that can be parameterized by, and can communicate with, other 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 software modules [8),22),23),25)].

In contrast, recent efforts at creating tools for helping policy engineers specify arbitrary runtime policies have only permitted the management of indecomposable policies [7),9)–12),15)–17),21),24)]. These tools enable engineers to specify an arbitrary runtime policy as an isolated and centralized policy module; however, the tools do not enable engineers to decompose that centralized policy into simpler subpolicy modules, which could be specified, analyzed, reused, tested, and modified in isolation.

Other related efforts do allow users to specify, visualize, analyze, and/or compose policies, but only in particular domains. For example, the Policy Visualization Analysis tool provides a GUI for managing existing SE Linux policies [26)]; the Expandable Grid manages access-control policies [20)] (which are a proper subset of runtime-enforceable policies [18)]); the Policy Mapper manages location-based access-control policies [5)]; front ends for SPARCLE and PERMIS manage natural-language access-control and privacy policies [6),13)]; and Fang and Firmato manage firewall policies [1),19)].

The Polymer project has attempted to address the lack of tools for managing compositions of arbitrary runtime policies [2),3)]. Polymer is a language and tool for specifying and enforcing runtime policies on untrusted Java-bytecode applications. Polymer policies exhibit *universal composability* (every policy can be composed 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. Hence, a Polymer policy P can be parameterized by another policy P'; 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 take two policy arguments P_1 and P_2 in its constructor; 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 we call `Conjunction` a *superpolicy* and P_1 and P_2 its *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 of P's responses to security-relevant events in a file named S. Using such techniques of parameterizing policies with other policies, engineers can use Polymer to build complex runtime policies as compositions of simpler subpolicy modules (Fig. 7 provides a high-level view of a complex Polymer policy that specifies runtime constraints on email clients and is built by composing simple subpolicy modules).

Although Polymer enables arbitrarily complex runtime policies to be specified more conveniently as compositions of subpolicies, Polymer achieves this convenience by requiring each individual policy module to be specified surprisingly inconveniently. The inconvenience stems from Polymer's requiring users to adhere to a complex programming discipline designed to partition policy code into effectless (i.e., free of state updates and I/O operations) and effectful methods. **Figure 1** (which is based on a policy downloaded from the Polymer project's website [4]) shows the convoluted way policy logic must be specified in Polymer in order to make policies safely composable. As this example illustrates, specifying policies in Polymer requires care and expertise.

1.2 Contributions

We present PoliSeer, a GUI-based tool designed to make it simpler and more convenient for non-experts to specify, visualize, modify, and enforce complex policies. As far as we are aware, PoliSeer provides the first GUI for composing arbitrary runtime security policies.

PoliSeer users import universally composable policies from a policy library,

```
public class ConfirmAllHTTP extends Policy {
    private boolean userCancel = false, noAsk = false;
    public Response query(Action a) {
        aswitch(a) {
            case ⟨abs void NetworkOpen(String addr, int port)⟩:
                if (port==80 || port==443) {
                    if (noAsk) return new OK(this);
                    if (userCancel) return new ExceptionResponse(this);
                    return new InsertResponse(this,new Action(null,
                        "JOptionPane.showConfirmDialog(Component, Object,
                            String, int)",
                        new Object[ ]{null, "Allow HTTP to "+addr+"?",
                            "Warning",
                            new Integer(JOptionPane.YES_NO_OPTION)}));
                }
        }
        return new IrrelevantResponse(this);
    }
    public void accept(Response r) {
        if (r.isExn()) userCancel = false;
        if (r.isOK()) noAsk = false;
    }
    public void result(Response r, Object rslt) {
        if (r.isIns() && ((Integer)rslt).intValue()==JOptionPane.NO_OPTION)
            userCancel = true;
        else if (r.isIns() && ((Integer)rslt).intValue()==JOptionPane.YES_OPTION)
            noAsk = true;
    }
}
```

Fig. 1 Convoluted but composable Polymer policy requiring user confirmation before making HTTP connections (taken from Ref. 4)).

compose them in meaningful ways by declaring arguments for all policy parameters, and generate code for the composed policy. We believe this process is straightforward enough that system administrators and even advanced end users can use PoliSeer to specify and enforce application-level policies; users simply customize (i.e., specify arguments for) expert-authored policies. Hence, a primary requirement for using PoliSeer is an ability to read and understand documentation for expert-authored policies.

Beyond specifying complex policies as compositions of simpler subpolicy modules, policy engineers can use PoliSeer to visualize complex policies holistically (as shown in Fig. 7). Such high-level policy visualizations may improve the engineers' understanding of large and complex policies and help engineers locate and isolate problematic policy modules.

Our implementation of PoliSeer uses Polymer as the underlying language of universally composable policies; in other words, Polymer is the language in which our PoliSeer implementation imports and exports policies. However, we have partitioned the implementation into Polymer-specific and non-Polymer-specific

modules to make PoliSeer readily portable to other policy-specification languages with first-class and parameterized policies.

PoliSeer is a declarative, Turing-incomplete policy-specification tool, so it lacks the expressiveness of imperative, Turing-complete policy-specification languages like Polymer [2),3)], Naccio [11)], and PSLang [10)]. Instead, PoliSeer users must rely on expert policy engineers to make useful universally composable policies available. This sort of tradeoff between expressiveness and usability is common with security-management tools. We believe that PoliSeer strikes a good balance between expressiveness and usability because, like standard IDEs, engineers may make use of PoliSeer's convenience when it does not impede expressiveness, but if greater expressiveness is required, PoliSeer users can always specify Polymer policies on their own (or have experts create new policy libraries for them) and then import those new policies directly into PoliSeer.

Because a non-expert is, by definition, unable to write complete policies in a complex language like Polymer, PoliSeer is designed to avoid presenting users with, and requiring knowledge of, language-specific details. Specifically, PoliSeer caters to non-experts by sparing them from having to understand the complex lexical, syntactic, and semantic rules of expressive policy-specification languages.

- Lexically, PoliSeer users do not need to understand the custom keywords of specialized languages like Polymer; the only tokens PoliSeer users need to be able to construct are for standard primitive values like integers and strings.
- Syntactically, PoliSeer users do not need to understand the grammatical rules of programming languages. Every policy-specification language we are aware of introduces specialized syntax for defining policies and the circumstances under which security-relevant actions should be allowed to execute. In PoliSeer, users define policies simply by selecting policies from an existing library and supplying arguments for those policies, resulting in a tree representation of policies. PoliSeer users may create, modify, and visualize policies without understanding programming-language syntax by dragging and dropping, selecting, and visually inspecting elements of these trees.
- Semantically, PoliSeer users do not need to understand programming-language typing rules beyond the ability to construct standard primitive values like integers and strings. Policy-specification languages typically introduce custom typing rules—for example, for typing policies, actions, results of actions, and policy reactions to actions, as well as rules for managing side-effects and/or rollbacks of imperative states. As described in Section 2, PoliSeer does several things to spare users from having to understand these frequently complex rules of semantics (e.g., PoliSeer automatically manages policy-module systems and determines and checks policy-argument types).

PoliSeer abstracts the lexical, syntactic, and semantic details of policy-specification languages into policy trees that can be created and manipulated with much less expertise than policies in general-purpose programming languages. The only requirements of PoliSeer users are that they can understand (1) basic GUI operations (like dragging and dropping components and clicking on menu items), (2) the semantics of library policies (e.g., a policy *conjunction* means that *both* of the argument policies will be enforced simultaneously), (3) tree representations of policies (i.e., that the children of a policy-tree node P represent the arguments to P), and (4) how to specify standard primitive values (integers, strings, etc). Note that users with any standard (GUI-based) operating systems would have to understand all these things, and many more, to author policies with existing tools.

Roadmap

We proceed as follows. Section 2 describes the design of the PoliSeer GUI; Section 3 explains and evaluates our implementation of PoliSeer as a Java application that inputs and outputs Polymer policies; Section 4 reports our experiences implementing a case-study policy in PoliSeer; and Section 5 concludes.

2. The PoliSeer Interface

PoliSeer aims to provide a straightforward graphical interface for conveniently and flexibly managing complex policies.

2.1 The Main Window

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

- The policy-selector panel allows the user to navigate the machine's file system to find existing composable policies. Our implementation begins by populating the policy-selector panel with all subdirectories and `.poly` files (i.e.,

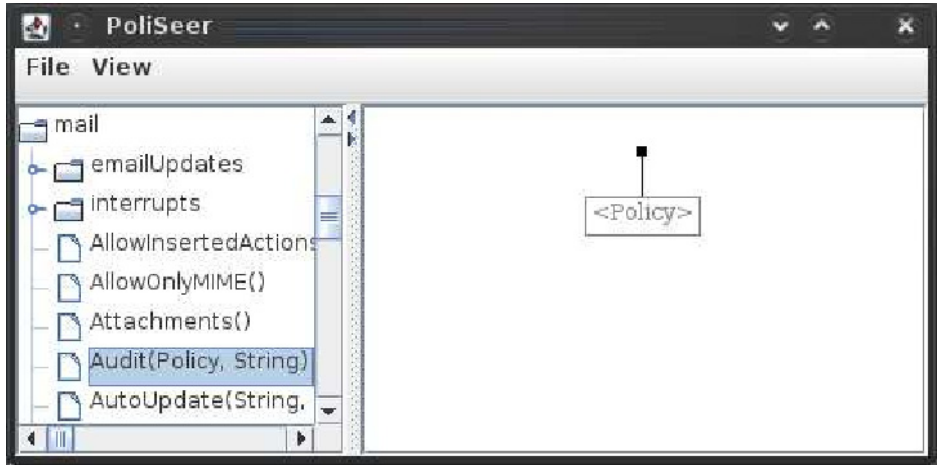


Fig. 2 Main PoliSeer window divided into policy-selector and policy-tree panels. The policy-tree panel is displaying the default, empty policy.

Polymer policy files) in the user's home directory. The policy-selector panel makes use of 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 that 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 Fig. 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.

- The policy-tree panel contains a graphical representation of the policy currently being created, visualized, or modified. When PoliSeer begins executing, it displays the empty policy as shown in Fig. 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, **Fig. 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 one `Policy` and one `String` argument for `Audit`.

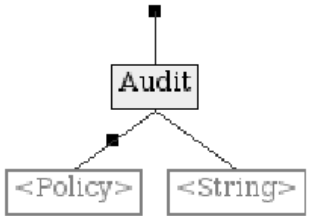


Fig. 3 Policy-tree panel showing a root `Audit` policy parameterized by another `Policy` and a `String`, though no children have yet been specified.

The only windows incorporated into PoliSeer besides the split main window are modal popup windows (for routine operations like selecting a location to load a policy from or save a policy to) and a window for viewing policy 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 clicked on a policy P in the policy-selector panel, 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, Fig. 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 Fig. 3 results from this addition; `Audit` has been added as the root node of the policy, but two new

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

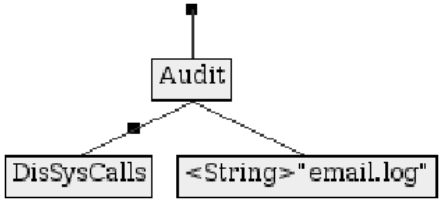


Fig. 5 Policy-tree panel showing an `Audit` policy with a subpolicy and string argument. This complete policy disallows system calls (i.e., `java.lang.Runtime.exec` methods) at runtime while logging all policy decisions to a file named `email.log`.

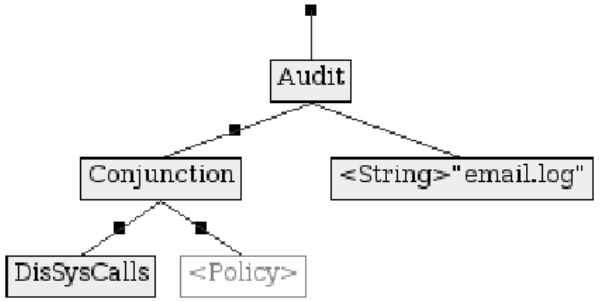


Fig. 6 The same policy-tree panel shown in Fig. 5, except that the user has now inserted a `Conjunction` policy between the `DisSysCalls` and `Audit` policies.

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 **Fig. 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 parsed as a value of the correct type. Currently, PoliSeer can only import and manipulate policies with a primitive-type `String` and `Policy` parameter, but all the policies provided in the standard Polymer distribution [4] satisfy this constraint.

Continuing with this example, **Fig. 5** shows a complete policy tree that results from inserting a (childless) policy and a string into the grayed-out nodes of Fig. 3. Two BIPs exist in Fig. 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 Fig. 5 ; the result is shown in **Fig. 6**.

By building policy trees, PoliSeer users may specify complex policies. All the superpolicies in policy trees can be thought of as *metapolicies*; superpolicies like `Conjunction` specify how to combine any possible combination of constraints imposed by subpolicies, even when those constraints conflict. For example, a PoliSeer user may specify a policy as being the conjunction of two subpolicies P_1 and P_2, where P_2 is defined to always respond to security-relevant actions in the opposite way as P_1 (e.g., when P_1 OKs an action, P_2 halts the target, and when P_1 does anything besides OK an action, P_2 OKs the action; this is the `Not` superpolicy described in Section 4.1). Such a composition of conflicting policies

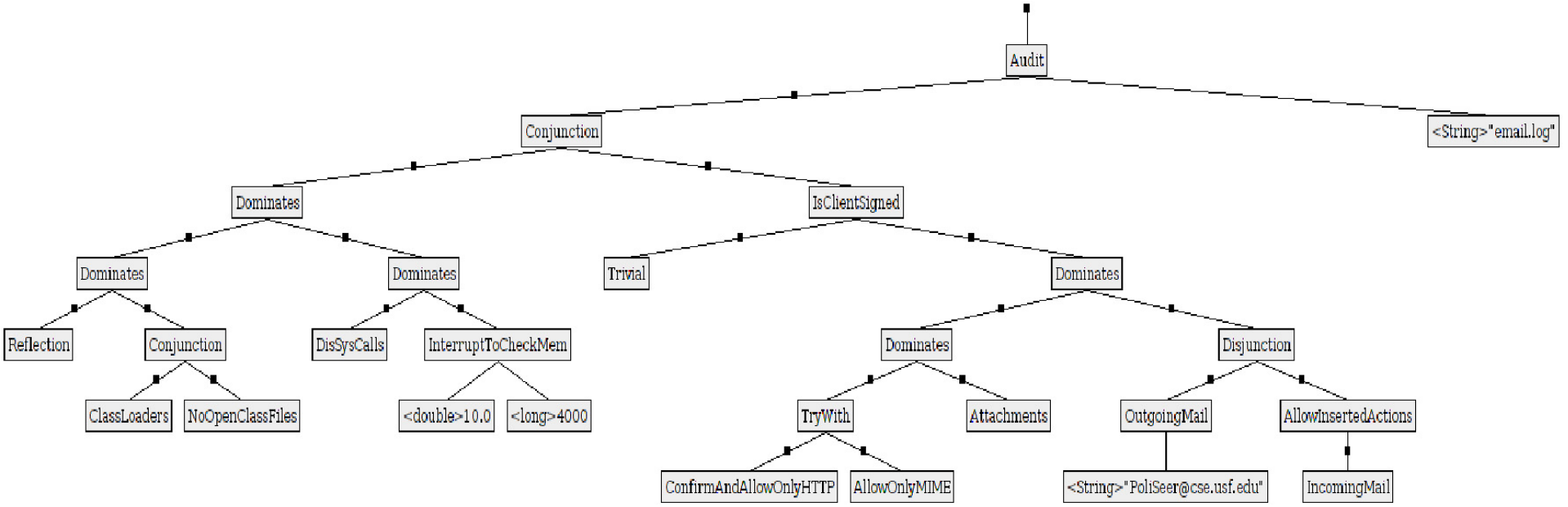


Fig. 7 Full tree for an example email policy (taken from Refs. 2), 3)).

is perfectly legal, and it is up to the semantics of the superpolicies to determine what happens when policies specify conflicting constraints. In this case, the `Conjunction` superpolicy obeys the strictest of its subpolicy constraints on each security-relevant action, so the overall (`Conjunction`) policy will always obey P_1's constraints as long as P_1 does not OK an action, but when P_1 OKs an action, the overall `Conjunction` will have to obey P_2 and halt the application. The ability to use metapolicies (i.e., superpolicies) to resolve conflicts between composed subpolicies is one of the key benefits of universally composable policies in systems like Polymer.

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 ready-to-enforce Polymer code for the policy in a `.poly` file with the `File -> Generate Policy Code` option. Conversely, users may resume creating, visualizing, 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 policy is missing one child of a `Conjunction` superpolicy, then the exported policy's constructor will accept a `Policy` argument to fill in for that missing child).

2.3 Visualizing Policies

As **Figs. 7**, **8** and **9** demonstrate, PoliSeer's policy-tree panel can provide a useful high-level visualization of complex security policies as compositions of simpler subpolicy modules. If PoliSeer's visualization of a policy 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

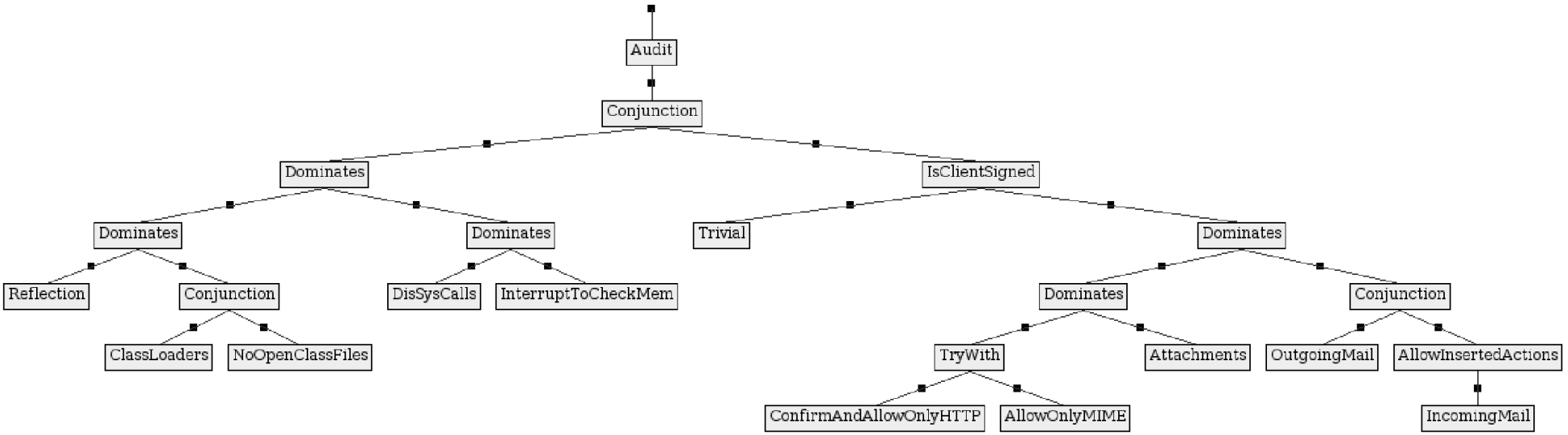


Fig. 8 The same policy tree shown in Fig. 7 but simplified by hiding non-policy nodes.

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, understand what the policy does and which arguments it expects, include `Audit` in the policy tree, and supply appropriate child-node arguments based on an understanding of `Audit`'s semantics.

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 Figs. 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

Although we have found PoliSeer's interface for creating and visualizing policies convenient and straightforward, we have found policy-modification operations more nuanced and challenging to enable and implement.

PoliSeer users may modify policy trees in three ways:

(1) Users may swap two existing sibling nodes (and their subtrees) by dragging and dropping one sibling node on 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),3)]). Nodes must have the same type to be swapped (e.g., a `String` cannot be swapped with a `Policy`), and PoliSeer currently does not support non-sibling node swapping due to policy-tree circularities that arise when swapping a node with one of its ancestors.

(2) Users may replace a policy node P in the policy tree with a policy P' selected from the policy-selector panel 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*. Technically, this means that PoliSeer must be able to assign each of P's nonempty children to be children of P' without introducing any type

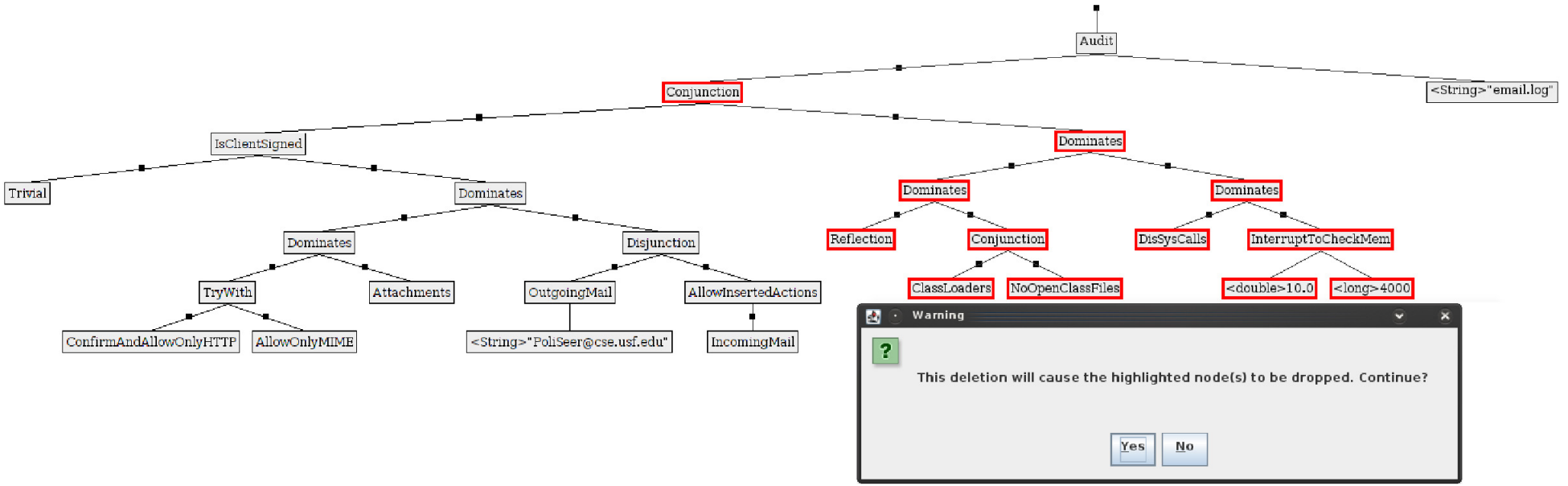


Fig. 9 Warning displayed before deleting a node with multiple policy children. PoliSeer has highlighted the descendants that will be discarded along with the node being deleted.

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 may 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. Figure 9 illustrates this interface.

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, available online at http://www.cse.usf.edu/~ligatti/projects/poliseer/. 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 (1828 lines of code). This module 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

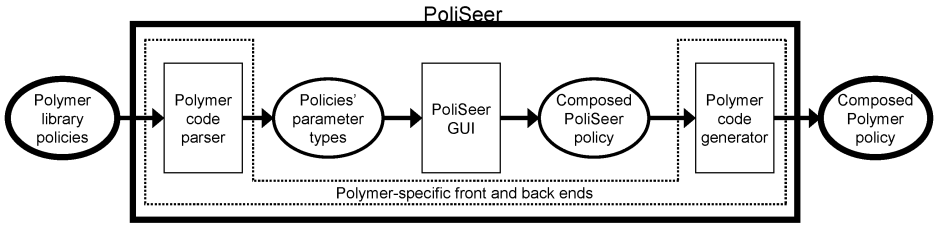


Fig. 10 Architectural overview of PoliSeer.

determine how they are parameterized, that is, which types of arguments the policies' constructors expect to receive. PoliSeer uses this type information to ensure that all policies receive arguments of the proper types and 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 top-down parser generator [14].

(2) The PoliSeer GUI (1451 lines of code). This module contains all the code to implement PoliSeer's graphical user interface, as described in Section 2.

(3) The back end (72 lines of code). This module generates Polymer code for the policy tree being visualized. Users can input the code that this module generates directly into the Polymer system, which will then 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 during basic operations such as parsing Polymer files and inserting nodes into policy trees. The tests were performed on a Sony Vaio laptop with Intel Core2 Duo 1.73 GHz CPUs and 1 GB 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 the GUI, and exit at the end of PoliSeer's `main` method. This time included the virtual-machine start-up time and was just 830 ms on average.

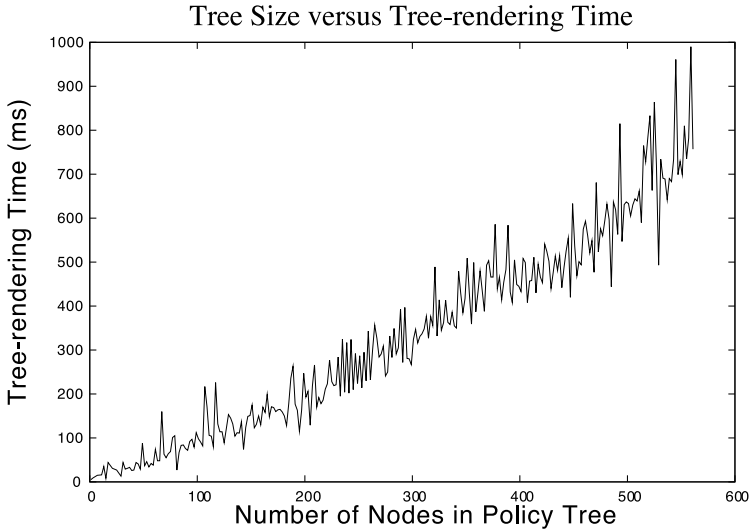


Fig. 11 PoliSeer performance rendering policy trees during node insertion.

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 3.1 ms 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 a performance similar to that shown in Fig. 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 the modified

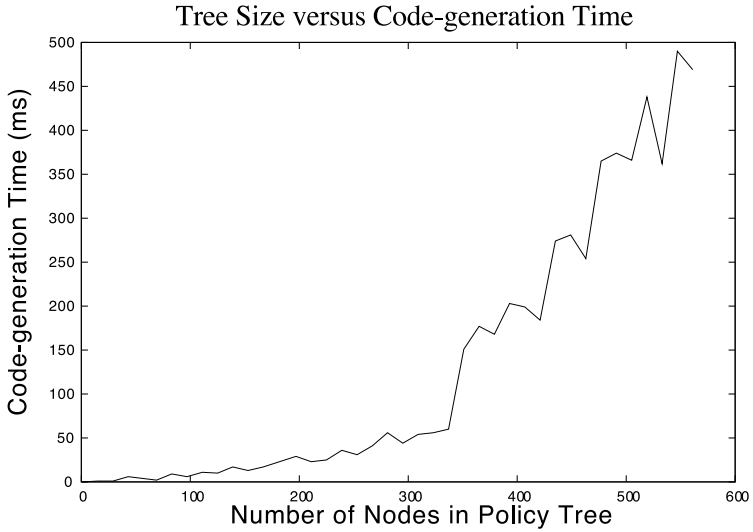


Fig. 12 PoliSeer performance generating Polymer code.

portions of trees during policy-tree manipulations.

Finally, we measured the time taken for PoliSeer to generate Polymer code files from policy trees. We implement code generation by (recursively) preorder-traversing the policy tree, while concatenating strings to construct every policy-tree node. As **Fig. 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 a complex case-study PoliSeer policy that restricts the runtime behavior of PoliSeer itself; that is, we have created a PoliSeer-controlling policy in PoliSeer. Moreover, we have successfully executed PoliSeer in the Polymer system while enforcing this PoliSeer-created policy.

4.1 Policy Overview

Figure 13 displays the policy tree for our case-study policy. 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 subpolicies' responses to the same action. In other words, the case-study policy always attempts to respect the restrictions of two high-level subpolicies.

We based the first of these two high-level subpolicies on policies described in earlier work [2)–4)]. This branch of the case-study policy enforces constraints that should be included in all Polymer policies to prevent the untrusted application program from using reflection, constructing class loaders, writing `.class` files, or invoking system-level functions (with `java.lang.Runtime.exec` methods). 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 (though their precise semantics is more subtle [2),3)]).

The second of the two high-level subpolicies in our case-study policy 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 (which again acts as a short-circuit `Conjunction` here).

The first branch of the PoliSeer-specific policy is the `NoNetworkOpens` policy, which disallows the untrusted application from opening any network sockets.

The second 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

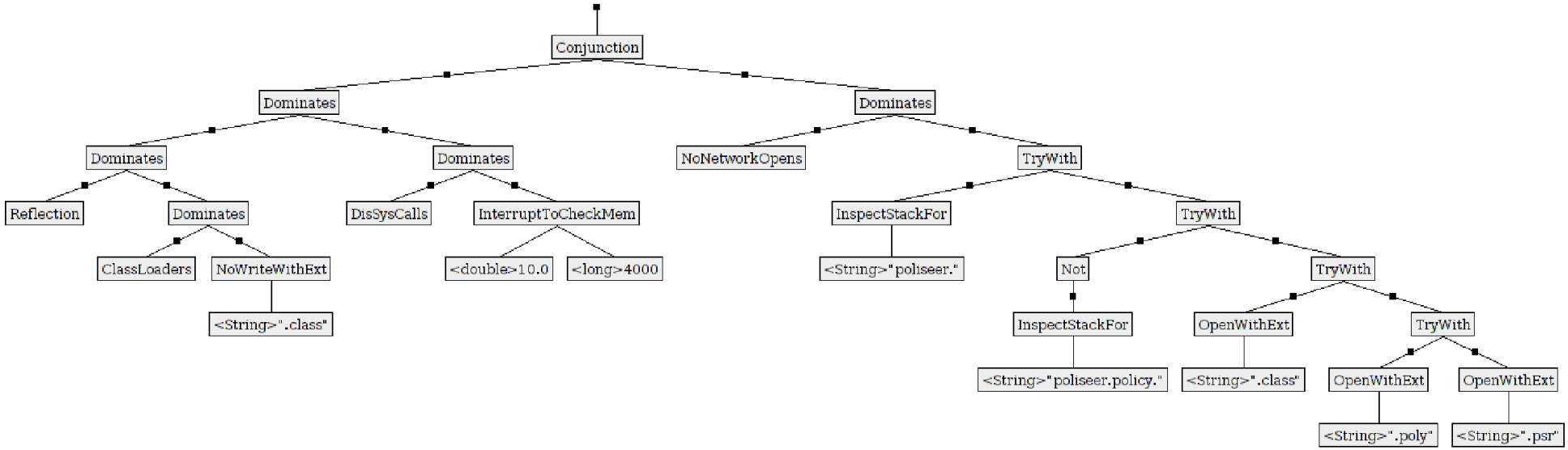


Fig. 13 Case-study policy, specified in PoliSeer, that constrains the PoliSeer application itself.

(`.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).

This second 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 in Polymer

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 reasons about (i.e., considers security relevant) application 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 the untrusted application (PoliSeer) invokes one of these security-relevant methods. 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. On the other hand, application-level operations that do execute security-relevant methods experience overhead when the case-study policy is being enforced, and that overhead is proportional to the number of security-relevant methods executed.

The average time for PoliSeer to start up in the Polymer system was 3.7 s, much higher than the 0.83 s 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 0.8 ms, and the average time to generate Polymer code increased by 1.3 ms (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 can infer that enforcing the case-study policy added approximately 1 ms 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 generate Polymer code for 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, except for code in the `poliseer.policy` package, could only open `.psr`, `.poly`, and `.class` files. It was natural for us to decompose this high-level goal into several subgoals, each testing for one of the conditions in which a file-open action would be allowed to execute; the overall policy only rejects actions that fail all the tests. The tests check, in turn: whether a non-PoliSeer method invoked the file-open action, whether a `poliseer.policy` method invoked the file-open action, whether the action is opening a `.class` file, whether the action is opening a `.poly` file, or whether the action is opening a `.psr` file. Composing all of these subpolicies with a (reusable) `TryWith` superpolicy yielded the desired high-level policy. This was a convenient (and actually enjoyable) way to specify a complex policy.

We also found it important when designing policies in PoliSeer to have a robust policy library available, as PoliSeer users who are unfamiliar with the underlying policy language (Polymer) will have difficulty constructing and importing their own Polymer policies. Fortunately, many existing policies comprise the 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 increase the library with four policies (`Not`, `InspectStackFor`, `NoNetworkOpens`, and `OpenWithExt`) before specifying the case-study policy in PoliSeer because Polymer's policy library is nascent. We believe each of the policies we added to the standard library are generic and will be useful to incorporate into a wide array of PoliSeer policies.

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 semantics during composition (by viewing the policy source, as described in Section 2.3).

We often found ourselves examining the 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 explanation of a policy's semantics from a quick examination of its source-code documentation.

Summary of Experiences Using the PoliSeer GUI

Overall we found the PoliSeer GUI a great aid for policy specification and visualization. Nonetheless, a couple GUI 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 conveniently 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 visualization 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 visualize 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 (as described in Section 2.3).

5. Conclusions and Future Work

We have presented PoliSeer, a tool for managing complex security policies. PoliSeer users rely on policy-composition experts to distribute libraries of universally composable policies (written in a language like Polymer). PoliSeer users build complex policies by composing those expert-written policies in meaningful ways. For example, we have constructed complex email-client and PoliSeer policies (Figs. 7 and 13). In our experience, PoliSeer has been a great aid for quickly specifying and generating code to enforce complex policies built as compositions of simpler subpolicies. We believe PoliSeer is useful, even for expert policy engineers, for clearly and conveniently visualizing complex policy trees. Moreover, PoliSeer's interface contains several considerations for conveniently modifying policies, such as policy replacement, branch-insertion points (BIPs), and branch deletions (cf., Fig. 9). Thus, we view PoliSeer as an integrated development environment (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 the PoliSeer project. One branch of work would involve conducting experiments and surveys to determine how usable PoliSeer's target users (system administrators and advanced end users) find the tool. Another branch of possible future work would involve improving the PoliSeer application to address some limitations that arose during the case study, which Section 4.3 introduced. In particular, 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-engineering tools will be as helpful and usable as standard software-engineering tools.

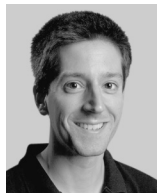
References

1) Bartal, Y., Mayer, A., Nissim, K. and Wool, A.: Firmato: A novel firewall management toolkit, *ACM Trans. Comput. Syst.*, Vol.22, No.4, pp.381–420 (2004).
2) Bauer, L., Ligatti, J. and Walker, D.: Composing Expressive Run-time Security Policies, *ACM Trans. Softw. Eng. Meth.*, To appear.
3) Bauer, L., Ligatti, J. and Walker, D.: Composing Security Policies with Polymer, *Proc. ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation* (2005).

4) Bauer, L., Ligatti, J. and Walker, D.: Polymer: A Language for Composing Runtime Security Policies (2008). http://www.cs.princeton.edu/sip/projects/polymer/
5) Bhatti, R., Damiani, M., Bettis, D. and Bertino, E.: Policy Mapper: Administering Location-Based Access-Control Policies, *Internet Computing, IEEE*, Vol.12, No.2, pp.38–45 (2008).
6) Brodie, C.A., Karat, C.-M. and Karat, J.: An empirical study of natural language parsing of privacy policy rules using the SPARCLE policy workbench, *Proc. 2nd Symposium on Usable Privacy and Security*, pp.8–19 (2006).
7) Damianou, N., Dulay, N., Lupu, E. and Sloman, M.: The Ponder Policy Specification Language, *Lecture Notes in Computer Science*, Vol.1995, pp.18–39 (2001).
8) Diehl, S.: *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*, Springer-Verlag, Berlin (2007).
9) Edjlali, G., Acharya, A. and Chaudhary, V.: History-Based Access Control for Mobile Code, *ACM Conference on Computer and Communications Security* (1998).
10) Erlingsson, Ú. and Schneider, F.B.: IRM Enforcement of Java Stack Inspection, *IEEE Symposium on Security and Privacy*, Oakland, CA (2000).
11) Evans, D. and Twyman, A.: Flexible Policy-Directed Code Safety, *IEEE Security and Privacy* (1999).
12) Havelund, K. and Roşu, G.: Efficient monitoring of safety properties, *International Journal on Software Tools for Technology Transfer* (*STTT*), Vol.6, No.2, pp.158–173 (2004).
13) Inglesant, P., Sasse, M.A., Chadwick, D. and Shi, L.L.: Expressions of expertness: the virtuous circle of natural language for access control policy specification, *Proc. 4th Symposium on Usable Privacy and Security*, pp.77–88 (2008).
14) JavaCC (2008). https://javacc.dev.java.net/
15) Jeffery, C., Zhou, W., Templer, K. and Brazell, M.: A lightweight architecture for program execution monitoring, *Program Analysis for Software Tools and Engineering* (*PASTE*) (1998).
16) Kim, M., Viswanathan, M., Ben-Abdallah, H., Kannan, S., Lee, I. and Sokolsky, O.: Formally Specified Monitoring of Temporal Properties, *European Conference on Real-time Systems* (1999).
17) Liao, Y. and Cohen, D.: A Specificational Approach to High Level Program Monitoring and Measuring, *IEEE Trans. Softw. Eng.*, Vol.18, No.11, pp.969–978 (1992).
18) Ligatti, J., Bauer, L. and Walker, D.: Run-Time Enforcement of Nonsafety Policies, *ACM Trans. Inf. Syst. Secur.*, Vol.12, No.3, pp.1–41 (2009).
19) Mayer, A., Wool, A. and Ziskind, E.: Fang: A firewall analysis engine, *Proc. IEEE Symposium on Security and Privacy*, pp.177–187 (2000).
20) Reeder, R.W., Bauer, L., Cranor, L., Reiter, M.K., Bacon, K., How, K. and Strong, H.: Expandable grids for visualizing and authoring computer security policies, *CHI 2008: Conference on Human Factors in Computing Systems*, pp.1473–1482 (2008).
21) Robinson, W.: Monitoring Software Requirements Using Instrumented Code, *HICSS '02: Proc. 35th Annual Hawaii International Conference on System Sciences* (*HICSS'02*)-*Volume 9* (2002).
22) Saigal, N. and Ligatti, J.: Defining and Visualizing Many-to-many Relationships between Concerns and Code, Technical Report CSE-090608-SE, University of South Florida (2008).
23) Schäfer, T., Eichberg, M., Haupt, M. and Mezini, M.: The SEXTANT Software Exploration Tool, *IEEE Trans. Softw. Eng.*, Vol.32, No.9, pp.753–768 (2006).
24) Sen, K., Vardhan, A., Agha, G. and Rosu, G.: Efficient decentralized monitoring of safety in distributed systems, *26th International Conference on Software Engineering* (*ICSE'04*), pp.418–427 (2004).
25) Shonle, M., Neddenriep, J. and Griswold, W.: AspectBrowser for Eclipse: A case study in plug-in retargeting, *Proc. 2004 OOPSLA Workshop on Eclipse Technology eXchange* (2004).
26) Xu, W., Shehab, M. and Ahn, G.-J.: Visualization based policy analysis: Case study in SELinux, *SACMAT '08: Proc. 13th ACM Symposium on Access Control Models and Technologies*, pp.165–174, ACM, New York, NY, USA (2008).

(Received October 31, 2010)
(Accepted April 8, 2011)
(Original version of this article can be found in the Journal of Information Processing Vol.19, pp.292–306.)

Daniel Lomsak was born in 1985. He received his B.S. from the University of South Florida 2008 and has been a Ph.D. student there since then. His main research interests are policy specification (e.g., practical and formal means of expressing security policies) and software security (e.g., security automata, policies as formal languages, and practical policy enforcement) as well as related topics such as software engineering (e.g., tools and constructs to combat policy complexity) and programming language design (e.g., modeling policy-specification languages and monitored programs).



Jay Ligatti was born in 1978. He received his M.A. and Ph.D. from Princeton University in 2003 and 2006, respectively. He has been an Assistant Professor at the University of South Florida since 2006. His research focuses on software security and its intersections with formal methods (e.g., models of enforcement), programming languages (e.g., policy-specification languages and code-injection attacks), networks (e.g., packet-classification algorithms), and software engineering (e.g., tools for building and maintaining complex policies). He was awarded the USF Outstanding Research Achievement Award in 2009 and the NSF Faculty Early Career Development award in 2008.