PolicyGlobe: A Framework for Integrating Network and Operating System Security Policies

Hamed Okhravi
University of Illinois at Urbana-Champaign
1308 W Main St.
Urbana, Illinois 61801
okhravi2@illinois.edu

Ryan H. Kagin
University of Illinois at Urbana-Champaign
1308 W Main St.
Urbana, Illinois 61801
kagin@illinois.edu

David M. Nicol
University of Illinois at Urbana-Champaign
1308 W Main St.
Urbana, Illinois 61801
dmnicol@illinois.edu

ABSTRACT

In modern networked systems with many machines and traffic control devices (such as firewalls), it is difficult to determine the overall effect of the security policies and configurations implemented inside the operating system and network devices. This paper describes PolicyGlobe, a framework to integrate operating system and network security policies. Using the idea of accessibility sets, PolicyGlobe integrates the Security Enhanced Linux (SE-Linux) access control policies with firewall configurations and traffic control policies. Using this framework, it is possible to construct a global accessibility set for each process in the system. PolicyGlobe makes it possible to determine the global effect of the local security policies and firewall configurations and answer the basic questions "can a subject in one machine access an object in another machine?" We have developed the integration algorithms, optimized the algorithms, implemented the entire framework, and conducted empirical studies on it. The studies show that in a network of 10 densely connected machines each with a large SE-Linux policy (∼275,000 lines of rules), PolicyGlobe can build the global accessibility sets in about 10 minutes. In a system with a more limited connectivity, the analysis takes a much shorter amount of time.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access controls, Information flow controls, Verification*; C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*

General Terms

Security

Keywords

Security Policy, Firewall, SE-Linux, Access Control, Policy Integration

1. INTRODUCTION

Cyber-security in modern large enterprise networks can be achieved by enforcing a security policy by the means of a number of policy enforcement points (PEPs). Secure systems often have a high-level description of the desired security objectives and goals. These high-level objectives are then implemented using many local low-level policies or configurations. For instance, access restrictions are often implemented using operating systems' access control mechanisms, personal firewalls, and dedicated network firewalls and traffic control devices.

The problem with this approach is that it is very difficult to determine the overall effect of the local policies and device configurations and answer the most basic questions such as "can a subject in machine 1 access an object in machine 2?" In modern secure operating systems with flexible and expressive security policy languages (such as SE-Linux), the accessibility question is difficult to answer even within a single machine. Hence, there are formal frameworks developed to analyze the operating system policies and answer the intra-machine accessibility question [17, 3]. However, these frameworks do not have a networked view of the system and cannot express the inter-machine access control or the network traffic policies.

In this work, we design and implement PolicyGlobe, a framework which integrates traffic control policies and operating system access control policies. We use the idea of access control spaces [7] to describe the Security Enhanced Linux (SE-Linux) policies, extend the spaces to describe the inter-machine access control, and integrate the firewall policies in the analysis. The access control spaces are built using the SE-Linux policies, the topology, and the rules inside the traffic policy enforcement devices (network-based firewalls, host-based firewalls, etc.) Consequently, the new framework is capable of integrating operating system policies with firewall rules.

Integrating operating system and network access control policies is useful in securing a networked system in different ways. First, the system administrator can understand the overall effect of the local configurations and policies and deduce who can access what resource in the system. Second, by integrating these policies, one can check the consistency of the low level configurations and policies with a high level policy or a set of security goals. This can ensure that the higher level goals are correctly achieved using these access control policies. Finally, the integration framework enables the administrator to correct possible flaws or misconfigurations in the system. For instance, if the high level security goal of an organization states that a low ranking employee cannot access some important documents while the integrated policy shows the contrary, the administrator can modify the low level policies in order to achieve the desired security restrictions.

We have implemented and optimized PolicyGlobe and conducted empirical studies on it. The results show that PolicyGlobe can analyze a network of 10 densely connected machine, each with a complex operating system policy of about 275,000 rules in about 10 minutes. We also analyzed a network with more restricted connections which took a much shorter amount of time.

The rest of the paper is organized as follows: section 2 provides a quick overview of the SE-Linux policy language constructs. Section 3 explains the idea of access control spaces and the SELAC framework for analyzing local SE-Linux policies. Section 4 describes PolicyGlobe and the algorithms used. The implementation of the framework and the empirical studies of the algorithms are detailed in section 5. We review the related work in section 6 before concluding the paper in section 7.

2. SE-LINUX POLICY OVERVIEW

The SE-Linux policy language is a flexible and fine-grained mandatory access control (MAC) scheme based on the Flask architecture[1] which supports two major policy types: type enforcement (TE) and Role-based access control (RBAC). This section reviews the main features of the SE-Linux policy language including its different policy types. For each type, different language constructs are introduced and a number of sample policies are provided. Detailed description of the language is well beyond the scope of this paper and can be found in the NSA manuals for SE-Linux [1].

2.1 Type Enforcement (TE) Policy

The type enforcement (TE) policy is one kind of policy in the SE-Linux policy language. In this policy, a type is associated with each subject (e.g. a process), often called the "domain" of the subject, and each object (e.g. a file, a socket, or a directory). The access control is then defined based on which subject type can access which object type in what mode. The following are the main language constructs used for type enforcement in SE-Linux:

`Attribute:` Defines different attributes associated with types. An attributes works as a macro and allows a rule governing many different types to be expressed in a compact form. For example, `attribute domain;` defines an attribute to be assigned to processes.

`Type:` Defines a type (of subject or object) and a set of attributes for that type. For example, the following rule defines the type "sshd_exec_t" and associates it with the attributes "file_type", "exec_type", and "sysadmfile".

```
type sshd_exec_t, file_type, exec_type, sysadmfile;
```

`Allow:` The main construct of the TE policy, defines what object types a subject can access and what permissions it has over them. The syntax of this construct is as follows:

```
allow subject_domain object_type : object_class permissions;
E.g., allow user_t bin_t : file { read execute };
```

`Constrain:` Puts extra limitation on accesses. It states that an access can be granted only if a Boolean condition is true.

`Type transition:` Has two different usages. It defines the type of newly generated objects and also the legitimate transitions of type for a process. For example, the following rule allows a process of the type "initrc_t" to change its type to "sshd_t" when runs an executable with type "sshd_exec_t".

```
type_transition initrc_t sshd_exec_t:process sshd_t;
```

In the SE-Linux, there are different permissions associated with each class of object. For instance, permissions for the "file" class includes `read`, `write`, `create`, `execute`, whereas the "socket" class has `listen`, `accept`, `send_msg`, and `recv_msg` permissions in addition to those of the file.

2.2 Role-based Access Control (RBAC) Policy

Role-based access control is another type of policy implemented in the SE-Linux. In an RBAC system, there should be two different mappings: the first is the mapping between users and roles, and the second is the mapping between roles and permissions. In the SE-Linux RBAC, the first mapping is defined using the `user` construct, but the second mapping uses the type enforcement (TE) as its basis. As a result, there is a mapping between roles and types and allow rules are defined only based on types (not roles). Note that roles are only assigned to subjects. All objects have the generic role of "object_r".

2.3 Network Configuration

When sending packets, the operating system needs to determine not only the protocol, but also the IP address that it sends information to, the interface that will be used, and the port through which traffic will be sent. In SE-Linux, these three are labeled `netifcon`, `nodecon`, and `portcon`. Each of these constructs applies a type to the ports, interfaces, and various IP ranges, to allow for certain restrictions of network traffic through any of these means. This way, the security policy can make decisions based on any of these information. Each of these constructs applies a label to the various network entities, and `netifcon` also applies labels to the packets that also travel through it. If the netifcon construct is not used, then the generic `netif_t` type is assigned. For `nodecon`, a node is labeled with a security context by subnet and network masks. Developments in SE-Linux allow for both IPv4 and IPv6 addresses to be used. Finally, `portcon` labels a port based on protocol and port number with a specified security context.

For example, the following rules assign `http_port_t`, `netif_eth0_t`, and `node_any_t` types to TCP port 80, eth0 interface, and 10.1.2.0/32 subnet respectively.

```
Portcon tcp 80 http_port_t
Netifcon eth0 netif_eth0_t
Nodecon 10.1.2.0 255.255.255.0 node_any_t
```

3. SE-LINUX POLICY MODELS

There have been two primary models of the SE-Linux system: a state space model and an access control space model (a.k.a accessibility sets model). In the state space model, each state is described by all the security contexts and a description of the entire system. In the access control space model, sets are created to establish relationships between various subjects within the system. The motivation behind these models is that it is nontrivial to answer simple questions like "Can a given subject access a given object?" Each of these models attempts to answer this and similar ques-

Table 1: **Access Control Spaces Model Sets**

Set or Function Name	Description
S	The set of valid security contexts for subjects (UxR(u)xT(r))
P(c)	The set of permissions provided by a particular class
M(s)	The set of permissions allowed for a particular security context
$\gamma(s,o,p)$	The filter function for particular constraints based on Booleans
$R^{\rightarrow}(s)$	The set of roles a particular security context can transition to
$T^{\rightarrow}(s)$	The set of allowed type transitions for a particular security context
$T^{\Rightarrow}(s)$	The set of types to which a context can make an automatic transition
$\#_R^O(s)$	The function that returns the role for a particular security context
$\#_T^O(s)$	The function that returns the type for a particular security context
$\Delta(s,c)$	The set of authorized permissions w.r.t. a context and permission

tions through different techniques. Here, we describe the accessibility sets model since we use it to determine the local (intra-machine) accessibility of objects.

3.1 SE-Linux Accessibility Sets Model

This model is a customized version of the access control spaces first introduced by Jaeger et al. [7]. In this model, based on the rules inside the policy, one builds the set of accessible objects for a specific subject to answer the question: "Can subject S access object O of class C in mode P?"

Each subject has a set of access control spaces, classified as permissible spaces, prohibited spaces, and unknown spaces. As the names indicate, the permissible and prohibited spaces are the precise assignments that are permitted and prohibited by the policy, respectively. It often happens that, while a security policy tries to define every case of a security decision, there are unspecified (and consequently unknown) results of a decision. It is ideal that these three spaces partition the entire assignment space; in reality, there is overlap between the spaces due to the specification. The model helps address this conflict through deriving properties about various assignments, constructing access control spaces, and finally analyzing conflicting spaces.

Zanin and Mancini [17] applied this model to SE-Linux, naming it the SE-Linux Access Control (SELAC) model. Based on each of the constructs in the SE-Linux policy language, the SELAC model creates a set that maps the relationship between subjects and objects. Table 1 lists several of the SELAC model sets and their description. A complete description of the sets and the algorithms used to create them can be found in the original paper [17].

The culmination of the SELAC model is the accessibility algorithm named `HAS_ACCESS`. The algorithm's inputs are a subject, an object, a class, and a mode, and it returns *true* if the subject can access the object of that class in the specified mode. The algorithm checks three different ways in which a subject can access an object:

1. The subject can directly access the object in the specified mode.
2. The subject can make an automatic transition to a different type which can, in turn, access the object in the specified mode.
3. The subject can transition roles to a different role which can give it access to new types, one of which can access the object in the specified mode.

The key element in this algorithm is the set Δ. A pair of an object and permission is a member of this set if the subject can directly access the object; this is precisely what the first condition is intending to check. The algorithm recurses through type transition and role transition sets to check the second and third conditions. The complete algorithm can be found in section A of the appendix.

4. POLICY INTEGRATION FRAMEWORK

This section explains PolicyGlobe for network and operating system policy integration. Policy integration through the SELAC model and the HAS_ACCESS algorithm already answers questions based on intrasubject interactions. For example, questions like "Can user_t access file_t in mode 'read'?" are directly answered by calling the algorithm with specific parameters. The algorithm works by checking for any of three different ways that a subject can access an object in a local machine (enumerated in the previous section).

In order to check for inter-network interactions, there is a fourth possibility that the algorithm needs to check:

4. The subject can create a socket and access the information from another computer through the socket.

To evaluate this, a new set of socket transitions needs to be created so that the PolicyGlobe algorithm can correctly check for any of these four possibilities.

4.1 Network Access Control

Network access control is implemented through the configuration of local devices and mechanisms, including router-based and host-based firewalls. A firewall is configured through a number of rules (referred to as a rule-set) which control the traffic and implement network access control.

The goal of this work is to integrate network and operating system access control policies. However, in a network of several firewalls with thousands of rules, it is not trivial to determine whether a machine can communicate with another machine using specific protocol and ports. The firewall rule-sets must be analyzed formally using an engine in order to answer this question. To this end, we use the Access Policy Tool (APT) [12]. APT is a tool that analyzes a network for connectivity among computers. The sole use of this tool in this work is to create a connectivity map for a network, a list of one-hop connections between computers within a network that abide by firewall rules and traffic security policies. The result is stored in a comma separated value list. The map simply shows the IP addresses and ports with which a machine can communicate for all the machines in the network.

4.2 Policy Integration Overview

In order to fully understand the ways in which subjects can interact across a network, one needs to integrate the operating system security policies, the firewall policies, and any other network information that could potentially allow or constrain traffic. This involves getting the security policy from the SE-Linux operating system as well as getting a list of connections with ports and protocols across the network. Given this information, the global accessibility sets are generated and the accessibility spaces within each of the operating systems are determined.

In order to evaluate the fourth possibility, we have to extend the accessibility set model by creating a new set `local-soc(subject)`. This new set will return pairs of port types and socket types for which the subject can read and write. Next, we check to see if there exist connections between machines such that two sockets on either side can be connected. What is meant by "connected" is that the two processes can communicate over the socket using the allowed port and protocol. If this is the case, then we have established what we will call a one-hop connection between subjects. We can create a square matrix of dimensions of the number of machines in a network that stores such information, called a connectivity matrix Π.

Given Π, we loop through the cells to determine chains of these socket transitions. For example, if type 1 on machine 1 can reach type 2 on machine 2, and type 2 on machine 2 can reach type 3 on machine 3, then we have determined a chain of socket transitions, allowing type 1 on machine 1 to communicate with type 3 on machine 3. Through this convolution of Π, we can determine all potential machines a subject can reach, and all possible subject types one could transition to upon reaching the machine. This convolved connectivity matrix is called Π^k.

Finally, we convert the convolved connectivity matrix Π^k to a set `Soc(subject,machine)`. This set returns pairs of subjects and respective machines which can communicate throughout the network. There are some assumptions to be made about the communication through sockets, which are elaborated in the next section. Once this set is created, queries can easily be made to check if certain elements exist in the set. The PolicyGlobe algorithm gets reduced to two main cases. In the first case, if two subjects are within a machine, then the original HAS_ACCESS algorithm can be used to determine if a subject is accessible by another subject. In the second case, if the two subjects are on different machines, it only needs to be determined if a subject can reach the target machine in question through socket transitions. If it cannot, then the access is not allowed; if it can, then the original HAS_ACCESS algorithm can be run on the target machine with the new subjects that were transitioned to. In the remaining sections, we will walk through the process to develop the modifications needed for network policy integration, describe some of the assumptions made on the integration scheme, describe the three-step process to create the global accessibility set, `Soc(subject,machine)`, and explain the modifications made to the HAS_ACCESS algorithm to evaluate the fourth criterion laid out earlier.

4.3 Integration Scheme Assumptions

Before delving into the details of the policy integration scheme, there are several assumptions that need to be made clear. These assumptions are made either to clear up ambiguity of the arguments or to simplify the problem.

It is assumed throughout the analysis that the entire network topology is available. This is necessary to determine what connections are valid within the network, and which sockets could potentially pass information between machines. It is also assumed that the entire SE-Linux policy from each of the machines within the network is available. Additionally, the traffic filtering device rules such as firewall rules need to be made available to determine not only if connections are valid, but if packets can be exchanged between the machines.

Originally, the question the HAS_ACCESS algorithm answered was "Can subject S access object O of class C in mode P?" The SELAC model is broken down such that we can answer questions that pertain to particular classes. In the interest of simplicity, we modify the set $\Delta(s, c)$ by taking the union of all classes for each subject:

$$\Gamma(s) = \bigcup_{c \in C} \Delta(s, c)$$

Thus, we create an accessibility set based on a particular subject for all classes, rather than for particular classes.

Sockets require active processes on two sides. To begin with, there is an initiating process that creates the socket on the initiating side. Then, there is a receiving process that creates the socket on the receiving end. Finally, for communication to happen, the initiating process writes to the socket and the receiving process reads from the socket. Because of the inherent nature of multiple independent processes for sockets, we make a couple of assumptions to simplify the model. First, every socket created by both the initiating and receiving side must be readable and writable, so that there can be bidirectional communication. Second, we will say that the receiving process is a slave to the initiating process. If the initiating process want to read a file on the receiving processes' machine, it will be assumed that the initiating process will write the request to the socket, the receiving process will read the file, and write the contents back to the socket, which the initiating process can read. Consequently, if the request was to create a new socket, a new socket would be created, and communication can occur through multiple machines. Third, because sockets are created on both sides, it is possible to have different types associated with the socket. We will assume that the type of socket on the receiving side is the same as the type of socket on the initiating side. While this is definitely a constrained version of the real world, the assumption is made for simplicity. Moreover, we will assume that the process reading from the socket is any type that could read from the socket with the exception of root types. If root types were included in the analysis, then every socket would leak the root access to files, which makes the answer to the question trivial. All of the server daemons in SE-Linux run with specific types rather than root. For example, `httpd_t, ftpd_t, smaba_t,` and `sshd_t` correspond to HTTP, FTP, Samba, and SSH daemon types. Finally, since the constructs `netifcon` and `nodecon` are used only in very specific instances, we will only determine connectivity between machines based on the `portcon` constructs and the various allow rules for necessary types.

Note that a process may run with variable access rights depending on the user running it. In SE-Linux when a process is initiated by a user, it is assigned an appropriate type based on the identity of the user (and the type transition rules.) Because the integration framework considers all of the SE-Linux rules, every possible type for a process is included in the analysis and its access rights are determined. For example, the analysis may indicate that a process run by a user can access a resource whereas the same process run by another user does not have such access rights.

The assumptions about the sockets make the analysis conservative; i.e., while the framework indicates that two processes *can* communicate, there is no way to know from the access control rules whether they *do* communicate in reality or not. A filter can optionally be defined by the administrator to eliminate communications that cannot happen in reality, but are allowed under the access control policies.

4.4 Step 1: Connectivity Matrix Generation

The first step has three main parts to it. We first determine what connections are allowed with respect to the firewall and other traffic devices in the network. We then determine all the security contexts that can create sockets on the initiating side of the network connections. Finally, we determine all the security contexts that can read from sockets on the receiving side of the network connections. If these three conditions are met, then we say that there is a connection between the two types between the two computers within the network, and based on our assumptions, we will add this information to the connectivity matrix Π.

Using the APT tool, we can determine what connections are allowed within the firewall and traffic policies within a network. It is beyond the scope of this work to discuss the details of the tool; it is only necessary to know that it returns a valid connectivity map. It is already an improvement over the previous methods to use this tool because we limit the number of iterations to the number of connections rather than the square of the number of machines; in a sparse network, this is a significant optimization, while in the worst case it does not simplify the problem.

The next step is to analyze the sockets on the initiating side of the connection by considering `portcon` and `allow` rules of SE-Linux. Within SE-Linux, when a subject creates a socket, the type of the socket takes on the same type as the subject. For example, if a process of type "httpd_t" created a socket to connect to the Internet, the socket would also be of type "httpd_t." When checking permissions on a socket, one needs to check the permission on the socket labeled with the process type, and one needs to check the permission on the socket with the port type. In other words, we need to check that the process can access the port for which data is being received and the socket object itself. Take, for example, the "httpd_t" type accessing port 80, which is labeled through `portcon` as type "http_port_t". This means we have to check the following rules:

```
1. allow httpd_t http_port_t:tcp_socket
    {name_connect, tcp_recv, tcp_send};
2. allow httpd_t self:tcp_socket
    {connect, read, listen, accept, write, create};
3. allow httpd_t self:tcp_socket
    {getattr, setattr, recv_msg, send_msg};
4. allow httpd_t netif_t:netif {tcp_recv, tcp_send};
```

The first rule says that the process needs to access the socket connected to the port by connecting, receiving, and sending data. The second and third rules say that the process needs to access the socket object itself in various modes. For the sake of completion, if there are net interface constraints, one would need to check to make sure that the interface can receive and send data through the fourth rule. The greatest amount of risk of information leak is when a process of one type uses a socket of another type because the subject can essentially transition to another type. Hence, in addition to the third rule above, we check the following rule for all other potential types. For example, say the type "user_t" created a socket, and left it open. Then, we would have to check for the following rule:

```
allow httpd_t user_t:tcp_socket
 {getattr, setattr, recv_msg, send_msg};
```

If that rule existed, we could now say that we can act as the socket type on the remote machine. In the first example, we can say that we are essentially any type that can read a socket of type "httpd_t"; in the second example, we are essentially any type that can read a socket of type "user_t."

Finally, we add our newly gained information to the connectivity matrix Π. Our matrix is laid out such that the source machine is the row and the destination machine is the column. The connectivity matrix Π symbolizes only the mandatory access control policy; if a system administrator wanted to forbid certain connections within the network on the discretionary level, a filter function would be used to zeroize certain elements of this matrix before convolution.

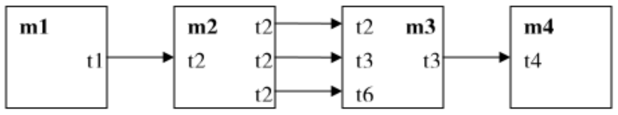

Figure 1: An example network configuration.

4.5 Step 2: Connectivity Matrix Convolution

The next step is to "convolve" this matrix to determine links of indefinite length. What we mean by "convolve" is to chain transitions between machines together, incrementing the length by one each time. This way, we can determine what machines can be reached, and within each of the remote machines, what types can the subject transition to. We will use the same terminology found in [13], and call the convolved connectivity matrix Π^k. We do this in a few sub-steps. The process will become readily apparent as we follow the example through this entire step.

The squares in Figure 1 symbolize the machines, the arrows are the sockets, and the types are the subjects within the machine of that type. For the sake of space, the type of the socket is neglected because we only care about the endpoints of communication. Specifically, in the socket from machine m1 to machine m2, the receiving subject is type t2. This means that t2 can read from the socket that t1 is using. By the assumption that the endpoints of the socket are of the same type, the socket could be of type t1 (where type t2 can read and write to sockets of type t1), type t2 (where type t1 can read and write to sockets of type t2), or another type t3 (where types t1 and t2 can read and write to sockets of type t3). To initialize the convolved connectivity matrix Π^k, we set Π^k equal to Π, and initialize a third element "hop_count" to keep track of the hop count. Thus, each element in Π^k contains a set of triplets of `(source_type, target_type, hop_count)`. This directly translates to the `source_type` on the machine denoted by the row being able

Table 2: An example connectivity matrix Π

Π	m1	m2	m3	m4
m1		(t1,t2,1)		
m2			(t2,t2,1),(t2,t3,1),(t2,t6,1)	
m3				(t3,t4,1)
m4				

Table 3: Convolved connectivity matrix Π^k

Π^k	m1	m2	m3	m4
m1		(t1,t2,1)	(t1,t2,2),(t1,t3,2),(t1,t6,2)	(t1,t4,3)
m2			(t2,t2,1),(t2,t3,1),(t2,t6,1)	(t2,t4,2)
m3				(t3,t4,1)
m4				

to reach the `target_type` on the machine denoted by the column in `hop_count` hops. We also initialize `iteration_count = 0`. Continuing our example, we have the following Π matrix shown in Table 2.

Our convolution process goes as follows. For `iteration_count = i`, we look for all entries where the hop count is equal to `i`. For each of those entries, we take `target_machine` and `target_type`, and check to see if there are any hops `target_type` can take to other machines within Π. If so, then we add a new element to the Π^k matrix. Calling the final new target machine and target type `new_target_machine` and `new_target_type`, we add `(source_type, new_target_type, i+1)` to Π^k`(source_machine, new_target_machine)`. All intermediate chains of connections of `hop_count < i` will have already been added to the matrix, so we need not add anything else. The final Π^k matrix is shown in Table 3.

The convolution algorithm can be characterized as a series of matrix multiplications. We define the operand "*" as follows:

$$(ta, tb, i) * (tc, td) = \begin{cases} (ta, td, i+1) & \text{if } tb = tc \\ 0 & \text{if } tb \neq tc \end{cases}$$

The above operand is used in matrix multiplication when Π^k is multiplied by Π. Using this definition, we can define Π^k recursively: $\Pi^k(iteration\_count = i{+}1) = \Pi^k(iteration\_count = i) \cup \Pi^k(iteration\_count = i) * \Pi$

We know that we do not need to continue if $\Pi^k(iteration\_count = i{+}1) = \Pi^k(iteration\_count = i)$, or when $\Pi^k(iteration\_count = i) * \Pi = 0$. We define `iterate_more` as a Boolean that determines whether or not we need to continue iterating through hops. This is easily determined by whether any of the entries in the aforementioned product is nonzero. For each loop, `iterate_more` is initialized to 0, and with each multiplication, if the result is nonzero, we set `iterate_more` to 1. The convolution algorithm is shown in section B of the appendix.

4.6 Step 3: Socket Connection Set Generation

The final step is to create a new set in the accessibility set model called `Soc(subject, machine)`, which returns a set of pairs of an object and machine that are accessible to the original subject and machine. Note that if a type cannot access a socket, then the question is reduced to the original local machine accessibility set. Thus, our convolved matrix Π^k only talks about socket connections and possible socket connection chains. To make the set `Soc(subject, machine)`, we loop through the convolved matrix, and for each row, we will add the `(subject,machine)` pair to the set that will be returned from the function called. To conclude the above example, calling `Soc(t2,m2)` will return `(t2,m3)`, `(t3,m3)`, `(t6,m3)`, and `(t4,m4)`.

4.7 PolicyGlobe Algorithm

Finally, to verify whether a subject on a machine can access an object on another machine, we develop the `PolicyGlobe` algorithm by modifying the `HAS_ACCESS` algorithm. Now that we have created `Soc(subject,machine)`, the `PolicyGlobe` algorithm can check for the fourth criterion, that of gaining access to an object over a chain of socket connections.

The complete `PolicyGlobe` algorithm is shown in section C of the appendix.

5. IMPLEMENTATION

It is nice to see how, in theory, the accessibility set model can answer questions by way of the `PolicyGlobe` algorithm. In reality, we often run into problems that are unforeseen with the theoretical algorithm, such as computer resource limitations or time constraints. This section goes through the steps that need to be taken to implement the `PolicyGlobe` algorithm in the previous section. First, some of the sets from the original accessibility set model need to be created. Second, our additional set `Soc(subject, machine)` needs to be created. Finally, we need to implement the modified algorithm to determine the solution to the global accessibility question.

5.1 Delta Set (△(subject))

This is by far the key element in the algorithm. This set is the space of authorized permissions. These are the explicit permissions granted to the subject and object pairs outlined in the security policy. Because our interest lies only within this set, we do not check other accessibility spaces such as the three kinds of prohibited permissions. We thus exploit the definition of the set Δ, and check only the allow and constrain rules, determined by the sets M and γ, respectively.

For the sake of space, we do not present the implementation of the M and γ sets. It suffices to mention that they are implemented using standard template logic (STL) multimap [15]. The reason for this is that it is an efficient way of making a one-to-many function, where one requests a subject, and the output is various pairs of objects and permissions. The complete implementational details can be found in our original publication [10].

In the implementation, SETools [16] provided a clean way to get all allowed rules, both universally and conditionally.

5.2 Type Transition Set

This set is the set of automatic transitions a subject can make. In order for an automatic transition to happen, we need to identify that there is the `type_transition` construct used for a subject, and that an allow rule exists for that `type_transition`. We check the $D_p(subject)$ and the $T^{\rightarrow}(subject)$ sets, respectively, and take the intersection to find out the set of automatic type transitions. The set `type_trans(subject)` is the intersection between the two sets, known as $T^{\Rightarrow}(subject)$.

The $D_p(subject)$ set is constructed by looping through the `type_transition` constructs and the $T^{\rightarrow}(subject)$ set by considering the allowed transitions through the `allow` construct. The $D_p(subject)$ set is stored in an STL multimap while the $T^{\rightarrow}(subject)$ set is held in an STL map.

Similar procedures are repeated to build the Role Transition Set ($R^{\rightarrow}(subject)$).

5.3 Local Socket Transition Set

There are three steps that must be taken to develop this set. First, we develop the connectivity matrix to determine all allowed single-hop connections. This creates the set `localsoc(subject)` on each machine. In order to do this, we have to check three conditions: first, determine, for a particular subject, what socket types it can access on the initiating side; second, determine all connections for which the sockets could potentially reach in the network (through only one hop); third, determine all the types of subjects that can read and write to the sockets on the receiving side.

5.4 Global Socket Transition Set

The process to create this set requires three steps: create the connectivity matrix Π, create the convolved connectivity matrix Π^k, and extract the data from Π^k to create `Soc(subject,machine)`.

5.4.1 *Part I: Building Connectivity Matrix* Π

Development of the connectivity matrix requires a couple of assumptions. First, we assume a certain naming convention for each of the files given to the connectivity matrix generation algorithm. For each operating system, two files are requested: a file containing the `localsoc()` set, and a file containing the all accessible objects for a subject. To generate the second file, one need only run the original `HAS_ACCESS` algorithm with a type that is known that cannot be reached, and save the resulting set `V`. These files should each be named appropriately to the code. We named the files `localsoc_i.txt` and `accessspace_i.txt`, where i represents the machine number (an arbitrary distinction). Once we make the `localSoc()` sets for each of the corresponding machines within the row of the connectivity map, we check to see which types can connect to which sockets, and which connections are established as a result.

The complexity is largely dominated by the reading of values from the file to the program; the default configuration of the `localsoc_i.txt` file has upwards of 275,000 values. Once loaded into an STL set, checking memberships and getting ranges are logarithmic in time. Thus, the complexity is linear with the amount of values in `localsoc_i.txt`

5.4.2 *Part II: Convolving the Connectivity Matrix* Π^k

Once the connectivity matrix Π is created, we convolve the matrix to generate Π^k. We define our operator * within the `Multiply` function, and any time cell multiplication is required, the `Multiply` function is called. This function is constant time, as it checks two parameters, and if they meet the requirements, then a return value is created and returned. Consequently, if they do not meet the requirements, then an error return value is returned.

The convolution is largely matrix multiplication, which for square matrices is $O(n^3)$, where `n` is the number of rows. Given the fact that each cell within the matrices is potentially a set of values, we define two averages. The average number of elements within a set within each cell of Π^k is defined as `avg(pi_k_elem)`. The average number of elements within a set within each cell of Π^k is defined as `avg(pi_elem)`. The complexity of the algorithm is:

$$O(n^3 \times avg(pi\_k\_elem) \times avg(pi\_elem))$$

5.4.3 *Part III: Generating Soc(subject,machine)*

The final step is straightforward. We perform simple translation between Π^k and the elements that are inserted into `Soc(subject,machine)`: for each element `(t1,t2,i)` in $\Pi^k[a][b]$, where `a` is the source machine and `b` is the target machine, we add `(t2,b)` to `Soc(t1,a)`. In order to generate this set, the algorithm cycles through each cell in Π^k, and for each cell, it cycles through each element in the set and adds it to `Soc(subject,machine)`. We define `avg(elem)` as the average number of elements within the set of any given cell within Π^k. Since the distribution of elements within Π^k is unknown, we can say that the complexity is lower-bounded by $O(n^2)$, where `n` is the number of rows, and is upper-bounded by $O(n^2(1 + avg(elem)))$. In fact, the complexity is exactly the sum of the cardinality of Soc(subject,machine) for each subject and machine pair.

5.5 Empirical Study

To illustrate some of the key features within the implementation, a study was conducted. A sample network of three machines was used, all with identical default SE-Linux installation security policies. In the trial, no changes were made to the policy. A sample connectivity map was used to demonstrate various types of connections through which computers could be connected. Connections were varied based on port (specifying "any" versus a number), protocol (specifying "any", "udp", or "tcp"), and IP (making note that connections are unidirectional). All of the analysis was conducted on an Optiplex 745 machine. The machine is equipped with a 2.4 GHz Intel E6600 with 2 GB of memory. Installed on the computer was Fedora Core 8 with SE-Linux, and the allocated partition for the OS was 40 GB.

Two small scenarios are used to determine the computational speed and complexity of the generation of this model. In the first scenario, the network comprises three computers, which are connected according to the connectivity map:

```
1. 192.168.1.1,any/tcp,192.168.1.2,any/tcp
2. 192.168.1.1,21/tcp,192.168.1.3,21/tcp
3. 192.168.1.1,598/tcp,192.168.1.3,598/tcp
4. 192.168.1.1,any/udp,192.168.1.3,any/udp
5. 192.168.1.2,any/any,192.168.1.1,any/any
6. 192.168.1.2,any/any,192.168.1.3,any/any
7. 192.168.1.3,any/any,192.168.1.2,any/any
8. 192.168.1.3,any/any,192.168.1.1,any/any
```

The variety of the connectivity map is to show that all conditions are tested when parsing the map. The entire connectivity map is only 8 lines. In the second scenario, the network comprises 10 computers, which are all connected to each other.

The first step in computing the local files `localsoc_i.txt` and `accessspace_i.txt` is the execution of the Unix script. This script uses the Setools suite to parse for certain features within the SE-Linux policy.

Execution of this script is largely based on the size of the security policy. Given that a security officer will not greatly influence the number of rules in the policy, the running time is expected to be about the same. The execution of this script took about 2:20. Since the default policy is run on all machines in both scenarios, this time is uniform across both of them.

Once the Unix file has executed the various Setools commands, it compiles and runs the local algorithms to build the accessibility sets. In running the algorithms, there are several factors that influence the running time. Since some of the algorithms cycle through all the rules, and this dominates cycling through any of the other files, the timing is largely due to the number of rules. More specifically, since certain rules have attributes, the timing is largely due to the number of decomposed rules, not just the number of rules, as a combination of attributes and grouped permissions could create thousands and even millions of rules with just one declaration. The overall timing to generate these two files on the aforementioned machine is approximately 3:31. Again, this time is uniform across both scenarios since the default policy is used for all machines.

To simulate a network, the above was done on the same machine several times, each with a different filename. Using the sample connectivity map, we can use the machine to simulate the analyzer for the complete accessibility set model. This three-step process creates the `Soc(subject,machine)` set. In the default policy, the `localsoc_i.txt` file contains upwards of 275,000 lines, and simply reading them into a set takes the bulk of the time, and in fact dominates the rest of the algorithm for generating the connectivity matrix Π. For each row of the connectivity map, the set takes about 2 s to generate. It is expected, then, that in a sparse network, the generation of Π will take a short amount of time; in a dense network, however, this could take much longer. In the first scenario, the generation of Π took about 22 s, while the more densely populated network of the second scenario took about 4:45 to generate.

Once the connectivity matrix Π was created, it was convolved to make Π^k. Both scenarios completed this operation on the order of 0.1 s. The conversion between the convolved connectivity matrix Π^k and the generation of `Soc(subject, machine)` is a one-to-one copy-and-paste algorithm. The complexity of this algorithm is solely based on the size of Π^k. Again, in both scenarios, the operation was completed on the order of 0.1 s.

In the default policy, `anaconda_t` can reach sockets of four different types, and as a result, the set `Soc(anaconda_t,1)` contains many different elements. To test the scenarios, the element `(hacker_t,anaconda_t,0)` was added to $\Pi[0][0]$, indicating that the type `hacker_t` could access a socket of type `anaconda_t` within machine 1. However, in both scenarios, there was no noticeable change in the time it took to compute the sets, despite many elements being added to Π^k.

Finally, the test for global accessibility can be checked through the `PolicyGlobe` algorithm. Since this algorithm simply checks a few membership tests, the algorithm is quick in producing a response. In the case where `hacker_t` has been injected into Π, a sample command as follows returns a true value:

```
PolicyGlobe(hacker_t, 1, dmesg_t, 2)
```

This is because `anaconda_t` on machine 2 can access `dmesg_t` on machine 2, `anaconda_t` on machine 1 can access `anaconda_t` on machine 2, and `hacker_t` on machine 1 can access `anaconda_t` on machine 1. There may be other paths (such as `anaconda_t` accessing `unconfined_t` first or `anaconda_t` accessing the same type on another machine first), but these are irrelevant; if access is possible, a true value is returned. As a result, injecting various security rules has little effect on the computation time.

The simulations bring a couple things to light. Primarily, because default security policies are so large, generation of local accessibility sets are consistent across machines. In generating the connectivity matrix Π, the time was directly proportional to the amount of rows in the connectivity map. The simulations indicated that convolving Π, independent of the number of entries, took very little time, as did the querying for global accessibility through the `PolicyGlobe` algorithm. Thus, the timing is largely proportional to the number of connections in the network.

6. RELATED WORK

There have been some work on composition and decomposition of policies. They mainly focus on one type of policy and try to formally compose/decompose different policies instances. Bonatti et al. [4] and Jajodia et al. [9] propose a modular approach for composing access control policies. Hull et al. [6] deal with the reverse problem, namely decomposing a global policy into multiple rule-sets for distributed policy enforcement points (PEPs). Al-Shaer and Hamed [2] and Nicol et al. [12] develop two frameworks for modeling and analyzing firewall policies.

Many frameworks have been developed for policy consistency checking. Among them Bonatti et al. [5] and Jajodia et al. [9] are two of the important attempts on integrating access control policies. The difference between these efforts and our framework is that most of the previous integration frameworks work with a simplified model of an access control policy while our framework works with actual firewall rules and SE-Linux policies.

The implementation of the real-world policies using SE-Linux as well as the analysis of the example policies can be found in [8] and [11]. Archer et al. [3] use PVS [14] [39] to prove properties about the state machine. The security properties in this framework are mapped to invariant properties of the state machine.

7. CONCLUSIONS AND FUTURE WORK

This work has used the concept of access controls spaces to compose policies across operating systems and traffic devices within a network. In particular, the new framework formalizes an analysis of a collection of SE-Linux operating systems within a network and provides a means for composing multiple security policies into a global view of the network, and a potential means to verify high-level policies. The framework brings to light various tools to help optimize the current research. Using this framework and the tools, one can create other security questions and proceed to check them by implementing algorithms similar to PolicyGlobe.

There are several directions future work can take. First, this framework can be extended to a consistency-checking algorithm similar to that of [13]. Second, this framework could be the grounds to create a high-level policy from a composition of low-level policies. This, coupled with an accurate translation mechanism, would be a powerful tool for security officials within corporations. Finally, this model can be extended to incorporate other traffic devices such as intrusion detection and prevention systems. We also plan to perform more experiments on larger and more realistic networks.

8. REFERENCES

[1] Se-linux. http://www.nsa.gov/research/selinux/index.shtml.

[2] E. Al-Shaer and H. Hamed. Modeling and management of firewall policies. *IEEE Trans. Network and Service Management*, 1(1), 2004.

[3] M. Archer, E. Leonard, and M. Pradella. Analyzing security-enhanced linux policy specifications. In *Proceedings of the 4th IEEE international Workshop on Policies For Distributed Systems and Networks*, June 2003.

[4] P. Bonatti, S. D. C. di Vimercati, and P. Samarati. A modular approach to composing access control policies. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, pages 164–173, November 2000.

[5] P. Bonatti, S. D. C. di Vimercati, and P. Samarati. An algebra for composing access control policies. *ACM Trans. on Inf. System Security*, 5:1–35, 2002.

[6] R. Hull, B. Kumar, and D. Lieuwen. Towards federated policy management. In *Proceedings of the 4th IEEE Iinternational Workshop on Policies For Distributed Systems and Networks*, page 183, June 2003.

[7] T. Jaeger, A. Edwards, and X. Zhang. Managing access control policies using access control spaces. In *Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies*, pages 3–12, June 2002.

[8] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the selinux example policy. In *Proceedings of the 11th USENIX Security Symposium*, pages 59–74, August 2003.

[9] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 474–485, May 1997.

[10] R. H. Kagin. *Optimizing Security policy integration and consistency validation.* M.S. thesis, University of Illinois at Urbana-Champaign, Urbana, 2008.

[11] P. A. Loscocco and S. D. Smalley. Meeting critical security objectives with security-enhanced linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, July 2001.

[12] D. M. Nicol, W. H. Sanders, S. Singh, and M. Seri. Usable global network access policy for process control systems. *IEEE Security & Privacy*, 6(6):30–36, 2008.

[13] H. Okhravi. *Security policy integration and consistency validation.* M.S. thesis, University of Illinois at Urbana-Champaign, Urbana, 2007.

[14] S. Owre, N. Shankar, and J. M. Rushby. User guide for the pvs specification and verification system. Computer Science Laboratory, SRI International, 1993.

[15] I. Silicon Graphics. Standard template library programmerï£¡s guide. http://www.sgi.com/tech/stl/, 2008.

[16] T. Technology. Setools policy analysis suite. http://oss.tresys.com/projects/setools, 2008.

[17] G. Zanin and L. V. Mancini. Towards a formal model for security policies specification and validation in the selinux system. In *Proceedings of the Ninth ACM Symposium on Access Control Models and Technologies*, pages 136–145, June 2004.

APPENDIX

A. HAS_ACCESS ALGORITHM

```
// V is the set of already tested security contexts
// AUTHORIZED is a Boolean variable containing the answer
// HAS_ACCESS is a Boolean function for computing the answer
1. V = $\Phi$;
2. AUTHORIZED = false;
3. HAS_ACCESS(x,c,y,w)

Boolean HAS_ACCESS(S subj,C class,O(c) obj, P(c) mode) {
// subj is the security context of the subject
// class is the class of the object
// obj is the security context of the object
// mode is the access mode to the object
   V = V $\cup$ $\{subj\}$
   if ( (obj,mode) $\in$ $\Delta(subj, class)$ ) {
        AUTHORIZED = true;
        exit;
   }
   else while(S$\setminus$ $V$ $\neq$ $\Phi$ && $exists\ t\ \in\ S\setminus\ V\ such\ that$
        (#$^O_T(t)$ $\in$ $T^{\rightarrow}(\#^O_T(subj))$ && $\#^O_R(t)$ $\in$ $R^{\rightarrow}(\#^O_R(subj))$)) {
        HAS_ACCESS(t,class,obj,mode);
   }
}
```

B. CONVOLUTION ALGORITHM

```
while(iterate_more)
{
  Pi_m[num_of_machines][num_of_machines] = 0;
  iterate_more = 0;
  for(int i = 0; i < num_of_machines; i++)
    for(int j = 0; j < num_of_machines; j++)
      for(int k = 0; k < num_of_machines; k++)
        for(each triplet pi_k_elem in Pi_k)
          for(each pair pi_elem in Pi)
            if(pi_k_elem.third == iterate_count + 1)
            {
               mult_result = Multiply(pi_k_elem,pi_elem)
               if(mult_result != ''error'')
               {
                  result_set = Pi_m[i][j];
                  result_set.insert(mult_result);
                  Pi_m[i][j] = result_set;
                  iterate_more = 1;
                }
             }
  Pi_k = Pi_k $\cup$ $Pi\_m$;
  iterate_count++;
}
```

C. POLICYGLOBE ALGORITHM

```
V = null;
AUTHORIZED = false
```

```
PolicyGlobe(x,m1,y,m);
S = set of subjects left to explore
M = set of machines
Boolean PolicyGlobe
  (subject subj, machine m1, object obj, machine m2) {
    if(m1 == m2) {
        V = V ∪ $\{subj\}$
        if( (obj,‘‘read’’) ∈ $\Delta(subj)$ ) { // $direct\ access$
             AUTHORIZED = true;
             exit;
        }
        else {
             while(exists t ∈ $S\setminus V$ $such\ that\ t\ in$
               type_trans(subj) and t ∈ $role\_trans(subj)$) {
                    PolicyGlobe(t,m1,obj,m2);
               }
       }
     }
     else {
        // check for each type in m2 that m1
        // could possible become through sockets
        for each t ∈ $S$ $such\ that$ $(t, m2)$ $within$ $Soc(subj, m1)$) {
            PolicyGlobe(t,m,obj,m2)
            if(AUTHORIZED == true) {
               exit;
            }
        }
     }
}
```