Supporting Access Control Policies Across Multiple Operating Systems

Lawrence Teo* Gail-Joon Ahn
Laboratory of Information Integration, Security, and Privacy (LIISP)
University of North Carolina at Charlotte
{lcteo,gahn}@uncc.edu

ABSTRACT

The evaluation of computer systems has been an important issue for many years, as evidenced by the introduction of industry evaluation guides such as the Rainbow Books and the more recent Common Criteria for IT Security Evaluation. As organizations depend on the Internet for their daily operations, the need for evaluation is even more apparent due to new security risks. It is not uncommon for large organizations to evaluate different systems, such as operating systems, to identify which would best fit their security policy. Each system would undoubtedly use different methods to represent access control policies. The security policy would therefore need to be translated into specific access control policies that each system understands, which is challenging when large and complex systems are involved. In this paper, we focus on the evaluation of operating systems. We describe Chameleos, a policy specification language that is designed to specify the access control policies of multiple operating systems. The strength of Chameleos is its flexibility to cater to many operating systems, while remaining sufficiently extensible to support the specific features of each system. We describe the design and architecture of Chameleos, and demonstrate that Chameleos can flexibly and effectively represent the access control policies of grsecurity and SELinux – two very different systems.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*constraint and logic languages*; D.4.6 [**Operating Systems**]: Security and Protection—*access controls*

General Terms

Security, management, design, languages.

*Lawrence Teo is also affiliated with Calyptix Security Corporation.

Keywords

Chameleos, policy specification, access control, operating systems, flexibility, extensibility

1. INTRODUCTION

As more organizations depend on computer systems and the Internet for their daily operations, it is becoming increasingly important to evaluate these systems to make sure that they are safe from security risks. Throughout the history of computer security, a number of industry evaluation guides like the Rainbow Books and the Common Criteria for IT Security Evaluation have been introduced to address this need. However, the evaluation of large and complex systems, like those prevalent in large enterprises, is very challenging. Each system may use different methods, languages, and formats to represent their configuration policies. It is difficult for an organization to translate its requirements into specific policies that each system understands. How can the evaluator be sure that the policy implemented on each new system matches the organizational requirements? Are there any incompatibilities among the policies of each candidate system?

To address this challenges, we proposed a solution to the problem with the introduction of Chameleos [16], a new policy specification language that is both *flexible* and *extensible*. Flexibility means that the language should be able to cater to multiple systems. Extensibility means that our language should be able to support the specific features in the policies of each target system or application. While Chameleos will eventually work with a variety of systems, its current incarnation focuses on access control policies on operating systems. Throughout the paper, we call the target systems in which Chameleos works with as *security-aware systems*, since we are primarily interested in representing security-related policies for these systems. We are currently developing Chameleos for three security-aware systems; however, we believe that it would be possible to support more systems by following the methodology described in this paper.

Our earlier work [16] introduced Chameleos where we provided conceptual descriptions on how Chameleos can be used to represent operating system access control policies. We concentrated on Systrace [12] as the primary security-aware system. This paper extends our work where we move from concept to implementation, and now describe how Chameleos can be used to represent the access control policies of two more security-aware systems – grsecurity [14] and SELinux [10].

The rest of the paper is organized as follows. Section 2 describes the background and related work. Section 3 discusses the design and architecture of Chameleos. In Section 4, we demonstrate how the access control policies of grsecurity and SELinux can be represented with Chameleos. This is followed by a discussion of ongoing and future work in Section 5. We then conclude the paper in Section 6.

2. RELATED WORK

The most relevant work that is related to our project is the Authorization Specification Language (ASL) by Jajodia et al [6, 7]. A widely accepted language in the access control community, ASL is a very flexible and expressive language that can be used for multiple access control policies. It has been adopted into different areas like modular authorization [17], logical access control frameworks [1], and privacy policies [8]. We used ASL in a number of our experiments [16] to help us design and develop Chameleos.

THINK [13] is a kernel-based framework that is designed to protect flexible operating system architectures. Unlike THINK, Chameleos focuses on supporting access control policies for operating systems *above* the kernel layer. This makes Chameleos accessible to practitioners who do not wish to use kernel-level access control facilities.

The motivation of our work should not be confused with that of XACML (eXtensible Access Control Markup Language) [11]. XACML is an XML-based language. Its goal is to allow access control policies to be specified by any application that requires authorization for users. In many ways, we view it as the other extreme of ASL. In our work, we strive to achieve middle ground by designing a flexible language within a particular domain – in this case, the operating systems domain. We did not choose XML as the basis of our language due to the potential overhead it might impose when used with low-level operating system access control policies, which may in turn diminish the practicality and usability of the language [16].

In Section 4, we will demonstrate how Chameleos is used to represent the access control policies of grsecurity and Security-Enhanced Linux (SELinux). grsecurity [14] is an "ACL system" that consists of a Linux kernel patch and other administration utilities to allow the specification of fine-grained access control policies for programs in a Linux system. SELinux [9, 10] is a research prototype that comprises a modified Linux kernel and specially patched programs to allow comprehensive access control policy features like mandatory access control, type enforcement [2], role-based access control, and multi-level security.

3. DESIGN AND ARCHITECTURE

3.1 Objectives and Design

Chameleos is designed with two objectives in mind: (1) it has to support multiple security-aware systems, and (2) it must be able to be implemented (we are focusing on a practical language, and not a theoretical one).

The advantages of implementing a single language for many security-aware systems are manifold. Having a single language would provide a common syntax for administrators to implement various policies. There is no need to re-learn the syntax for different systems, thus presenting a convenient way for the administrator to specify multiple system policies. This is especially true in the evaluation of two different systems. Also, if there are similar systems, we do not need to convert the policies from one system to the other. We strive to achieve a *"write once, deploy everywhere"* philosophy in the design of Chameleos.

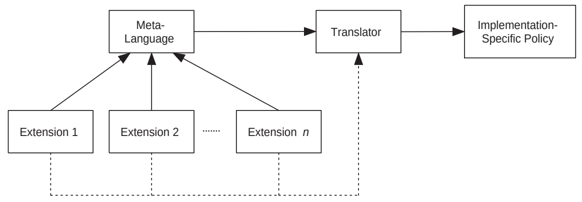

Figure 1: The Chameleos architecture.

We have defined a preliminary version of the Chameleos grammar using Extended Backus-Naur Form (EBNF) [15]. Space restrictions do not permit us to include the full EBNF grammar in this paper, but we shall briefly describe the key ideas here. The Chameleos grammar allows us to specify various access control policy notions such as generic subjects and objects (and their relevant types). Since access control policies frequently involve groups of entities, the Chameleos grammar also allows the specification of arbitrary sets and groups. More complex utilities such as compound expressions, arbitrary comparison operators, and arbitrary permissions are also supported. Other areas that we are considering include the specification of hierarchies and constraints. We strive to be as comprehensive as possible, while keeping flexibility and extensibility in mind. We will revisit these ideas again in Section 4.4.

3.2 Architecture

The Chameleos architecture (Figure 1) consists of three main components: the meta-language, a translator, and extensions. The meta-language is the core Chameleos language itself, namely the generic syntax of Chameleos. This includes operators, statement terminators, reserved keywords, variable types, and other related entities. The meta-language also clearly specifies how functions and procedures should be defined.

The next component of the Chameleos architecture is the translator. As its name implies, a translator is used to convert a Chameleos policy into a system-specific policy. For example, a translator can translate a Chameleos Systrace policy that is written in Chameleos into an actual Systrace policy. The translator needs to know the meta-language natively, and be able to load extensions into the system when required.

Extensions are used to support specific systems (one extension supports one system). To write a Chameleos policy for a specific system, say SELinux, we would need to use the specific features of that system. For example, a Chameleos SELinux policy would need to be able to support the specification of type transitions and roles. A Chameleos policy for another system like Systrace would focus on system calls. To support such extensibility, a Chameleos extension is used. An extension comprises the variables, library of functions (that can be implemented as a well-defined API), and other properties of a specific system. To illustrate, a Chameleos SELinux extension would be composed of convenience functions to specify SELinux notions like users,

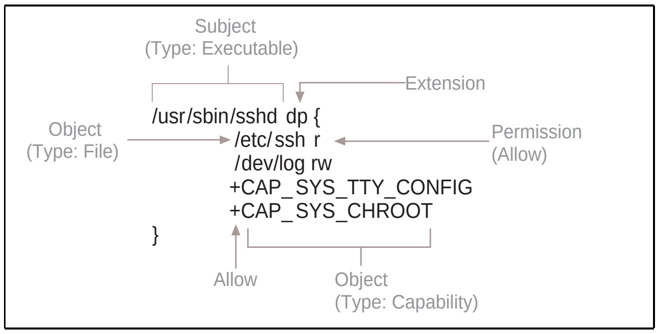

Figure 2: Excerpt of a grsecurity policy annotated with Chameleos directives.

user-role assignments, and type transitions. An extension can be likened to a module (analogous to a Java class package in Java). The Chameleos EBNF grammar provides a generalized framework to develop extensions that can support access control policies for multiple operating systems. As new versions of the supported operating systems are released, these extensions will be updated to reflect the latest features in those operating systems, while maintaining backward compatibility wherever possible. This would enable Chameleos to evolve as new systems become available.

The contents of an extension are specified in a file (such as `selinux.extension`). For instance, the variables declared inside a typical SELinux extension could be:

```
subject_type domain, user;
object_type  file, lnk_file, sock_file, fifo_file;
perm         p_create, p_read, p_write, p_getattr,
             p_setattr, p_link, p_unlink, p_rename;
opmode       allow, user;
```

This SELinux extension allows the declarations of SELinux-specific subject types, object types, and permissions. These variables can then be used throughout the specific policy.

4. REPRESENTING ACCESS CONTROL POLICIES WITH CHAMELEOS

In this section, we discuss the criteria that we used to select security-aware systems that were used to aid the development of Chameleos. We then describe two security-aware systems – grsecurity and SELinux – and demonstrate that Chameleos can flexibly and effectively represent their policies, even though they are very different systems.

4.1 Selection of Security-Aware Systems

We are developing Chameleos using an evolutionary bottom-up approach, which would enable us to build a solid language that can be practically implemented [16]. In order to do this, it is important to select a sample of security-aware systems that we can use to design and test the language incrementally. This initial pool of systems is very important, as it would fundamentally influence future development of the language. In this initial development phase, we focus on grsecurity and SELinux as our target systems. In our earlier work [16], we conducted some work on Systrace [12] as well. We believe that these systems are different enough to help us develop a flexible and extensible language. In this section, we discuss the roles of grsecurity and SELinux in the development of Chameleos.

```
import grsecurity.extension;

/* Set our global policy to default-deny. */
global_policy(default_deny);

subject("/usr/sbin/sshd", executable)
assoc(extension, dp)
{
    object("/etc/ssh", file)
    {
        perm(allow, perm_r);
    }

    object("/dev/log", file)
    {
        perm(allow, perm_r, perm_w);
    }

    object("CAP_SYS_TTY_CONFIG", capability)
    {
        perm(allow);
    }

    object("CAP_SYS_CHROOT", capability)
    {
        perm(allow);
    }
}
```

Figure 3: The grsecurity policy from Figure 2 represented using Chameleos.

4.2 grsecurity

We shall now discuss grsecurity's access control policy. grsecurity's policies are the simplest and therefore the easiest to develop for. We present an excerpt of a grsecurity policy in Figure 2. We have annotated it with directives on how the policy should be implemented using Chameleos features. Briefly, this grsecurity policy states that the executable program `/usr/sbin/sshd` is allowed to read from the directory `/etc/ssh`[1]. `sshd` can also read and write to the `/dev/log` file. Lastly, `sshd` is allowed to have two capabilities: `CAP_SYS_TTY_CONFIG` and `CAP_SYS_CHROOT`.

Consider the first line. `/usr/sbin/sshd` is designated as a subject. In the context of grsecurity, `/usr/sbin/sshd` is an executable, so it is specified as a subject of type "Executable." Likewise, we denote `/etc/ssh` as an object of type "File." Arbitrary permissions can be given, such as `r` for read. Permissions can be allowed or denied; in this case, the read (`r`) permission is allowed. Different objects, such as capabilities, can still be designated as objects, but they are labeled as different types. In this case, `CAP_SYS_TTY_CONFIG` and `CAP_SYS_CHROOT` are specified as objects of type "Capability." Note that the subject is associated with the subject modes `d` and `p` (for a more thorough discussion of grsecurity features, we refer the reader to the grsecurity documentation [14]).

We now show the grsecurity policy specified using Chameleos in Figure 3. The grsecurity extension is imported using the `import` statement. The contents of the grsecurity extension file (named `grsecurity.extension`) are:

[1]Although /etc/ssh is actually a directory, UNIX considers directories as files.

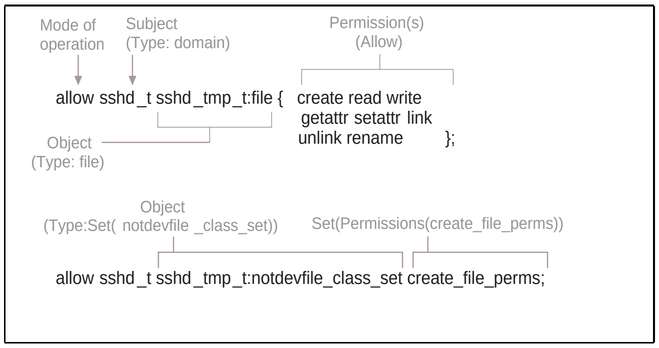
Mode of operation
Subject (Type: domain)
Permission(s) (Allow)
allow sshd_t sshd_tmp_t:file { create read write getattr setattr link unlink rename };
Object (Type: file)
Object (Type:Set(notdevfile _class_set))
Set(Permissions(create_file_perms))
allow sshd_t sshd_tmp_t:notdevfile_class_set create_file_perms;

Figure 4: Excerpt of an SELinux policy annotated with Chameleos directives.

```
subject_type executable;
perm         perm_r, perm_w, perm_x;
object_type  file, capability;
group        d, p;
assoc        extension;
```

We define the global policy directive as default-deny using the `global_policy` function. We then define the subject block for `/usr/sbin/sshd` and create the association with the `d` and `p` extensions using the `assoc` statement. Within the subject block, we define each object and the subject's permissions, using `object` blocks and the `perm` statement. We can define the object type for each object as an argument to the `object` statement.

It can be observed that Chameleos is flexible enough to capture all the elements of the grsecurity policy. Also, it is sufficiently extensible to capture grsecurity-specific characteristics, such as the extensions (via the `assoc` statement).

4.3 Security-Enhanced Linux (SELinux)

SELinux has the most comprehensive support for access control policies among our target systems. An excerpt of an SELinux policy is shown in Figure 4. There are two similar statements in this policy, with some differences in terms of granularity. We shall discuss each statement in turn.

The first statement permits a subject in the `sshd_t` domain to perform a set of operations (create, read, write, link, unlink, rename, and set and get the attributes) to an object of type `sshd_tmp_t` of object class `file`. An administrator who wishes to replicate these permissions to multiple files has two options: (1) use a similar statement for each single file that need to adopt these permissions; or (2) group the permissions into a set, and assign the set to the files instead. Option 1 would introduce a lot of repeating statements, which may reduce the readability of the policy. Therefore, Option 2 would be more favorable, as shown in the second statement. A subject in the domain `sshd_t` can now execute any of the permissions given in the `create_file_perms` set to any object of type `sshd_tmp_t`, provided that it is in the object class `notdevfile_class_set` set. This means that an `sshd_t` subject can now create, read, modify, delete, and rename any non-device file as long as it is of type `sshd_tmp_t`.

While the second statement provides a more convenient method to express permissions that are similar to the first statement, it loses some granularity in the process. For instance, if the `create_file_perms` set includes more (or less) permissions than those declared in the first statement, the administrator might unintentionally allow more (or less)

```
import selinux.extension;

/* First statement. */
opmode(allow)
{
    subject("sshd_t", domain)
    {
        object("sshd_tmp_t", file)
        {
            perm(allow, p_create, p_read, p_write,
                 p_getattr, p_setattr, p_link,
                 p_unlink, p_rename);
        }
    }
}

/* Second statement. */
define_set(notdevfile_class_set,
           file, lnk_file, sock_file, fifo_file);

define_set(create_file_perms,
           p_create, p_read, p_write, p_getattr,
           p_setattr, p_link, p_unlink, p_rename);

opmode(allow)
{
    subject("sshd_t", domain)
    {
        object("sshd_tmp_t", notdevfile_class_set)
        {
            perm(allow, create_file_perms);
        }
    }
}
```

Figure 5: The SELinux policy from Figure 4 represented using Chameleos.

permissions than what was initially desired. Sets and other convenience functions should therefore be used with care.

Chameleos can represent this SELinux policy excerpt using the extension presented earlier in Section 3.2. The statements in Figure 4 can be supported with Chameleos using the Chameleos policy in Figure 5, which is described as follows: The first statement declares `allow` as a mode of operation using the Chameleos `opmode` statement, since SELinux supports many types of operation in a single file. `sshd_t` is declared as a subject of type `domain`, and `sshd_tmp_t` is declared as an object of type `file` to represent the SELinux object class. The multiple permissions can also be specified as Chameleos permissions. For the second statement, we use the Chameleos set function `define_set` to define the `notdevfile_class_set` and `create_file_perms` sets. In both statements, Chameleos allows the subject to execute the permissions on the corresponding object.

4.4 Feature Comparison

We summarize this section by presenting Table 1 that compares the features of Chameleos with SELinux, Systrace, and grsecurity. We first provide a general description of the features listed in the first column of the table. *Subjects* and *objects* are basic access control concepts. A subject is an entity that is authorized (or not authorized) to access an object. A typical subject could be a program while a typical object could be a file. *Subject type* and *object type* refer to

	Chameleos	**SELinux**	**Systrace**	**grsecurity**
Target systems	All security-aware systems in the OS domain	Linux only	OpenBSD, NetBSD, MacOS X, and Linux	Linux only
Subjects	Generic	System-specific	System-specific	System-specific
Objects	Generic	System-specific	System-specific	System-specific
Subject types	Yes	Represented using classes and types	ABI only	Files only
Object types	Yes	Yes	System calls only	Files only
Variables	Yes	Yes	No	No
Macros	Yes	Yes (m4 macros)	No	No
Groups	Yes	No	No	No
Roles	Yes	Yes	No	Future version
Arbitrary sets	Yes	Yes	No	No
Compound expressions	Yes	Yes	Yes	No
Association	Yes	Yes	Yes	Yes
Comparison operators	Arbitrary	System-specific set	System-specific set	System-specific set
Aliases	Yes	No	No	No
Mode of operation	Yes	Yes	No	No
Permissions	Arbitrary	System-specific	System-specific	System-specific
Hierarchies	Yes	Yes	No	No
Constraints	Yes	Yes	No	No
Specification of defaults	Yes	Yes	No	No

Table 1: Feature comparison between Chameleos, SELinux, Systrace, and grsecurity.

the kind of subject (say, what kind of program) and the kind of object (for instance, a specific kind of file) respectively. To reduce repetition in policies, *macros* can be used. *Groups* enable entities to be represented as a collective. *Roles* facilitate the assignment of *permissions* to users, where common groups of permissions can be assigned to roles instead of individual users (the latter would be tedious). *Sets* refer to the ability to group related entities as a set. *Compound expressions* allow multiple expressions to be joined, such as the conjunction of multiple boolean expressions using *comparison operators* like AND and OR. *Associations* link one entity to another, such as assigning a user to a role. *Aliases* are similar to the `typedef` notion in C. *Mode of operation* refers to the ability of statements to carry out different operations. *Hierarchies* are used to support notions like role hierarchies. *Constraints* allow us to restrict the number of ways in which a policy statement can be specified. *Specification of defaults* assigns default values to variables.

Throughout Table 1, the common theme is that Chameleos strives to be as generic as possible, while the other systems have features specific to their particular area. This design allows Chameleos to be flexible such that it can support multiple security-aware systems. We have also attempted to make the list of features as exhaustive as possible, so that Chameleos can support the specific features of each system.

As an example, consider the types of permissions used by various systems. UNIX uses read, write, and execute file permissions. grsecurity introduces new permissions like the ability to view hidden files (`v`). The Andrew File System enables users to allow/deny listing of their files by other users. To adequately support these permissions and those of future systems, Chameleos supports arbitrary permissions.

Likewise, Chameleos supports arbitrary comparison operators. While most systems use regular (in)equality comparison operators, some support special comparison operators. For instance, Systrace uses `match` to compare regular expressions. Support for arbitrary permissions and comparison operators makes Chameleos extensible to more systems.

5. ONGOING AND FUTURE WORK

Chameleos began its life as a language to specify operating system access control policies. We have faced the numerous challenges and learned many lessons while designing and developing Chameleos. Based on this experience, our current plan for Chameleos is for it to support security-aware systems outside the operating systems domain. Our new vision for Chameleos is for it to be a family of languages for multiple security-aware systems [15]. As a start, we intend the Chameleos family to have three primary variants: Chameleos-`os` (for operating systems), Chameleos-`ids` (for intrusion detection systems), and Chameleos-`fw` (for firewalls). Covering these key security-aware systems would enable organizations to evaluate their systems in a more comprehensive and effective manner. For example, an organization will be able to use Chameleos to test the introduction of a new web server and how it would affect IDS signatures and firewall rules, with a single Chameleos policy. The new Chameleos architecture will support the ability to automatically deploy and enforce translated policies.

To support these capabilities, we are currently building a network testbed [15] to test multiple operating systems, servers, firewalls, and IDSs. Our current focus is on UNIX-based open source systems due to cost constraints, but we do plan to support Windows-based systems at a later stage.

Apart from the renewed vision and plan for a Chameleos family of operating systems, we are also working on new components for the Chameleos architecture. These components include a syntax checker, analyzer, and reverse translator. The *syntax checker* would serve as the foundation for all syntax checking requirements in the other components. The *analyzer* would be used to analyze a Chameleos policy for conflicts and ambiguities. The analyzer would have to take constraints [3] and conflict resolution techniques [4] into account, especially for complex systems like SELinux [5]. The *reverse translator*'s role is to translate a system-specific policy into a Chameleos policy.

Other areas that we are working on include improvements to the language itself, such as safety analysis, safety checks, and support for dependencies among extensions (one likely way to do this is to borrow concepts like inheritance from the object-oriented world). In the usability area, we intend to introduce stock Chameleos templates, which would help administrators define Chameleos policies. This would be most useful when using different variants in the Chameleos family.

6. CONCLUSION

We have presented the design of Chameleos, a flexible and extensible language that can be used to represent the policies of multiple security-aware systems. The goal of Chameleos is to help organizations evaluate large and complex systems, so that better decisions can be made on the design of a large system, especially from the security angle. We adopt a *"write once, deploy everywhere"* philosophy, where a single Chameleos policy can be translated into specific policies for multiple systems. This would be of tremendous benefit to system evaluators, since they would not have to re-learn the syntax of each and every new system they encounter.

Chameleos differs from other languages, where our aim is to make it practically realizable and highly usable. This is different from the goals of other languages that are more theoretical in nature.

In this paper, we have demonstrated how Chameleos can be used to represent the access control policies of grsecurity and SELinux successfully and effectively. grsecurity and SELinux are two very different security-aware systems, so this shows that Chameleos is indeed flexible and extensible.

While the current version of Chameleos works with operating systems only, future versions will be in the form of a family of languages for operating systems, firewalls, and intrusion detection systems. This would help organizations to make even better evaluations of their systems and networks.

Acknowledgments

This work was partially supported at the Laboratory of Information of Integration, Security and Privacy at the University of North Carolina at Charlotte by grants from the National Science Foundation (NSF-IIS-0242393) and Department of Energy Early Career Principal Investigator Award (DE-FG02-03ER25565).

7. REFERENCES

[1] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A logical framework for reasoning about access control models. In *Proceedings of the 6th ACM Symposium on Access Control Models and Technologies*, pages 41–52, Chantilly, VA, 2001.

[2] W. Boebert and R. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference*, 1985.

[3] T. Jaeger. On the increasing importance of constraints. In *Proceedings of the 4th ACM Workshop on Role-Based Access Control*, pages 33–42, Fairfax, VA, October 1999.

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

[5] T. Jaeger, R. Sailer, and X. Zhang. Resolving constraint conflicts (to appear). In *Proceedings of the 9th ACM Symposium on Access Control Models and Technologies (SACMAT 2004)*, IBM T.J. Watson Research Center, Yorktown Heights, NY, June 2004.

[6] S. Jajodia, P. Samarati, and V. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 31–42, Oakland, CA, May 1997.

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

[8] G. Karjoth and M. Schunter. A privacy policy model for enterprises. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, Nova Scotia, Canada, June 2002. IEEE Computer Society Press.

[9] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference (FREENIX '01)*, June 2001.

[10] NSA. Security-Enhanced Linux. http://www.nsa.gov/selinux/.

[11] OASIS. OASIS eXtensible Access Control Markup Language TC. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.

[12] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, August 2003.

[13] C. Rippert. Protection in flexible operating system architectures. *ACM SIGOPS Operating Systems Review*, 37(4):8–18, October 2003.

[14] B. Spengler. grsecurity. http://www.grsecurity.net/.

[15] L. Teo and G.-J. Ahn. Towards effective security policy management for large and heterogeneous environments. Technical Report UNCC-SIS-06, UNC Charlotte, Spring 2005.

[16] L. Teo and G.-J. Ahn. Towards the specification of access control policies on multiple operating systems. In *Proceedings of the 5th IEEE Workshop on Information Assurance*, pages 210–217, United States Military Academy, West Point, NY, June 2004.

[17] H. F. Wedde and M. Lischka. Modular authorization. In *Proceedings of the 6th ACM Symposium on Access Control Models and Technologies*, pages 97–105, Chantilly, VA, 2001.