Real-time Access Control Reconfiguration

Ashish Gehani and Gershon Kedem
Department of Computer Science, Duke University
Email: gehani@cs.duke.edu

***Abstract*— As the frequency of attacks faced by the average host attached to the Internet increases, reliance on manual intervention for response is decreasingly tenable. Operating system and application based mechanisms for automated response to perceived threats will be necessitated. One possibility is the modification of the reference monitor of the operating system such that the choice about whether a right should be granted (that was previously made with a lookup of a static value), can instead be made by dynamically delegating the decision to code that is customized to the specific right and is able to use the context of the request to make a more informed choice. We describe such an *Active Reference Monitor* (ARM) and the associated changes in semantics of the security subsystem.**

Keywords: Predicated, Access Control, Reconfiguration

I. Introduction

The dramatic increase in the scale of the Internet has had several implications. It has brought with it an increase in the number of potential attackers. It allows, and has resulted in, rapid changes in deployed software with a commensurate proliferation of exploitable bugs. Further, it provides a medium by which the attackers can rapidly communicate vulnerability information between each other.

By performing auxiliary checks prior to granting a permission, the chance of it being granted in the presence of a threat can be reduced. By tightening the access control configuration, the system's exposure can be reduced. By relaxing the configuration, the exposure can be allowed to increase. The use of auxiliary checks will introduce runtime overhead. In addition, when permissions are denied, applications may be prevented from functioning correctly. These two factors require that the use of the auxiliary checks must be minimized. Thus, the tension between between security and performance must be balanced.

One method to achieve this balance is by dynamically reconfiguring the access control configuration of the system based on the runtime context to trade performance only when attacks warrant the tighter access control. Automating the intervention would significantly reduce the *mean time to response* in the face of a threat, and with it the cumulative exposure of the system.

Supported by a USENIX Association Research Grant and a North Carolina Networking Initiative Graduate Fellowship.

II. Design

A. User Space Response

One option for automating intrusion response is the use of the NOSCAM [Gehani02b] proactive auditing utility. It synchronously monitors the output of an intrusion detector to determine the current threat level, and can be configured to invoke specific responses based on the threat level. This approach suffers from three drawbacks, the latter two of which are inherent to the approach of invoking a user space application to effect response.

1) Granularity: The first problem is that more information is needed to customize the response than what is available as output from a typical intrusion detector. Without this, the response has to be generic, which reduces its utility and increases the frequency of it occurring when not required. This could potentially be addressed through modification of the intrusion detector.

2) Overhead: The second issue is that this is a "push" based method, where changes in the defense posture of the system are made each time a potential threat is detected with the associated overhead of effecting the change. (Consider the example of making a large set of files inaccessible to a particular user if their activity matches that defined in a particular intrusion signature. A push based system changes the permissions of all the files each time the signature matching passes a predefined threshold, then returns the permissions to their previous values after the threat is deemed to have passed. This imposes very significant overhead.)

3) Asynchronicity: The third flaw is that there remains a temporal gap between when activity occurs and when it is detected, followed by a delay till the time a response can actually be launched. This creates a race condition between detection and response. It allows numerous attacks to effect damage where it would have been possible to mitigate their effect if the response could have been effected asynchronously.

B. Operating System Support

The experience described above argued that automated response needed operating system support to be effective. We had previously instrumented both a Linux 2.2.12 kernel and a Java 1.2 runtime to create the SADDLE auditing subsystem. Based on measurements of the overhead that was incurred there, we concluded that instrumenting all system calls with hooks for responses to be inserted was not a preferable option. We would need to discriminate and choose a small subset of system calls where the most impact could be achieved. We chose to add the hooks to the extant security subsystem, calling the result an active reference monitor for reasons described below.

C. Security Policy

The access control matrix uses the two axes of a grid to define the set of *principals* and *resources* of a system. Each cell corresponds to the access of a specific principal to a particular resource. The *rights* to be allowed are stored in the cell. Operations that change the set of of principals, set of resources or rights in a cell, of the system by definition trigger a change in the *protection state* of the system. A set of rules define which states are *safe*. This is the *access control policy* of the system.

The model was first described by Lampson [Lampson71]. Harrison, Ruzzo and Ullman [Harrison76] formalized the model. They defined a *leak* to be the transfer of a right from one cell of the matrix to another. The system is said to be safe if it starts from a safe state and all leaks are allowed by the access control policy. They showed this problem was undecidable for arbitrary systems. Jones, Lipton and Snyder [Jones76] articulated the take-grant model with properties similar to those of capability based systems. New models continue to be proposed, such as Sandhu's typed access matrix [Sandhu92].

Denning [Denning76] defined *information flow* between two objects as the dependence of the value of one object on the value of the other. A set of rules regarding which flows are permissible constitutes the system's *information flow policy*. Bell and La Padula's Multi Level Security policy [Bell73] for the military and Brewer and Nash's Chinese Wall policy [Brewer89] for corporations, are examples of such information flow policy.

Complete security policy covers both access control and information flow. It may be specified in any formal language. For example, Foley [Foley89] used predicate calculus, Cuppens [Cuppens93] used modal logic, and Peri [Peri96] used temporal logic.

D. Permission Semantics

Policy can be specified using the constants, variables and operators of a formal language, L, adhering to the language's grammar. The axioms, X, rules of inference, I, and proof technique, Q, must also be specified. A statement, σ, in L can be verified by starting with the set X, applying rules from I, according to the methodology of Q, and checking to see if σ can be derived. In principle, it is possible to completely specify and verify the security policy of a system, as shown by Peri [Peri96]. If L is expressive enough, the response to exceptional conditions (such as partial intrusion matches) can also be specified as part of the policy. Yet such approaches are not used in practice due to

the extreme complexity of fully specifying the policy (even without a response component) of a typical deployed system. Additionally, they rely on being able to precisely and completely define the elements of the formal system, an unrealistic expectation when only part of the system is within the policy specifier's administrative domain. Finally, they do not account for the gap between the abstract model and the implementation of it.

An alternative approach is to use a subset of the security policy that can be framed intuitively. While this method suffers from the fact that the resulting specification will not be complete, it has the benefit that it is likely to be deployed. The specific subset we consider is that which constitutes the authorization policy. These consist of statements of the form $(\sigma \Rightarrow p)$. Here σ can be any legal statement in L. If σ holds true, then the permission p can be granted. In the current paradigm, the reference monitor maintains an access control matrix, M, which represents the space of all combinations of the set of subjects, S, set of objects, O, and the set of authorization types, A.

$$M = S \times O \times A, \quad \text{where} \quad p(i,j,k) \in M \quad (1)$$

The space M is populated with elements of the form:

$$p(i,j,k) = 1 \quad (2)$$

if the subject $S[i]$ should be granted permission $A[k]$ to access object $O[j]$, and otherwise with:

$$p(i,j,k) = 0 \quad (3)$$

In our new paradigm, we would replace the elements of M with ones of the form:

$$p(i,j,k) = \sigma, \quad \text{where} \quad \sigma \in L \quad (4)$$

Thus, a permission check will no longer be a lookup of a binary value, but instead be the evaluation of a statement framed in a suitable language, L, which will be required to evaluate to either true or false, corresponding to 1 or 0.

E. Temporal Constraints

A user space application gains access to a resource through a system call. This call in turn will check whether permission exists by a call to the reference monitor. The latter call implicitly makes several assumptions, two of which are affected in this paradigm in a manner that must be addressed.

The first is the expectation that call will complete in $O(1)$ time. Given that we propose to grant a permission, p, predicated on the successful evaluation of a statement, σ, it would appear that the time complexity of the call to the reference monitor will increase. In addition, the choice of the language, L, would appear to be a factor. (Consider, for example, that σ may be decidable in some languages but not in others.)

To avoid the above issues, we use a constant bound, t, on the number of steps that the proof technique, Q, can take. If this bound is crossed, the evaluation is deemed to have failed. This guarantees that the reference monitor will return in $O(t) = O(1)$ time as required by the implicit assumption made by applications. Under this constraint, the choice of language is also no longer material. (Recall that a Turing-complete language with alphabet Σ, if limited to being recognized in t steps, becomes decidable since it can be recognized by a finite state automaton corresponding to the regular expression of the disjunction of the appropriate subset of the $|\Sigma|^t$ possible strings in L.)

The second assumption relates to the period of time for which the return value of a permission check remains valid. In the old paradigm, this was governed solely by a static value stored in the system that was unlikely to have changed during the course of execution of an application. In the new paradigm, the check can be dependent on a large number of factors some of which are evaluated in real-time and may therefore vary rapidly, such as system load or free memory. The result is that when permission is denied, it can now reasonably be expected that this may change in a short period for some permissions. To maintain compatibility with existing applications, this can not be communicated through the existing programming interface of the system calls. It is, however, important to expose the reason for the denial so that new applications (or modifications to old ones) can use the information to decide whether to wait and request permission again. (They could also utilize the information to take alternate defensive actions.)

Since the details must be communicated in parallel with the extant control flow of the

application, a new signal, e, is defined. It can be sent asynchronously when a permission is denied with the tuple (p, σ) stored in a temporary buffer, b, accessible to the signal handler. Note that σ is the statement that evaluated false causing permission p to be denied. Applications can gain access to the information in (p, σ) by defining a signal handler for e which extracts the data from b each time it is invoked and inserts it in a thread-specific location, B, accessible to the interrupted thread. Application code can access the cause of a permission denial by immediately accessing B each time it is interrupted by signal e and extracting the tuple (p, σ). Note that the code for the system call has not been modified, so e propagates immediately to the application code and can be immediately handled, before another permission check can occur in the course of executing the system call.

F. Activation

The goal of evaluating a predicate before granting a permission is to tighten the circumstances under which access to a resource is granted. The constraint, however, may not be required by default. Since predicate evaluation imposes a computational overhead, it would be preferable to activate it only when needed. There are two factors that impact the choice, which are the cost and the benefit of utilizing the auxiliary check.

The cost is the increase in running time of the application that will be imposed. It is proportional to the frequency, f, with which that permission gets evaluated in a typical workload. If the expected value of f is larger, the cost associated with its use is greater.

The predicate evaluation will typically be activated based on an increase in the threat level as determined by an intrusion detector. The threats being monitored for by the intrusion detector require a number of permissions to be granted in order to succeed. The frequency of occurrence of the permission in question in this set serves as a baseline estimate for the weight w. The greater the value of w, the more benefit there is to tightening the permission in question. In practice, however, the intrusion response system must also factor the likelihood of each threat and the cost of the consequence

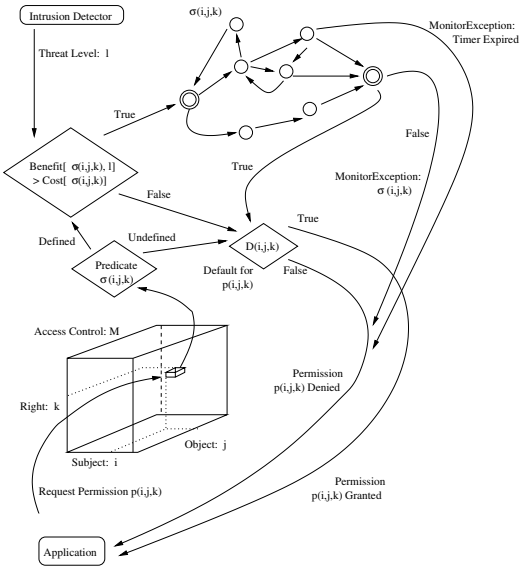

Fig. 1. **ActiveMonitor** predicated permissions have 3 distinguishing features: (i) Constrained running time, (ii) Dynamic activation if expected benefit exceeds cost, (iii) Interrogatable for cause of denial.

of its occurrence into the weight w. The details of how w can be calculated for a risk-based intrusion response system are described in [Gehani03].

When the host's exposure (or system risk if a risk-based intrusion prevention system is in use) crosses exceeds a threshold, then a permission, whose auxiliary checks are not currently in use, is selected. Its predicates are activated prior to be granted. This reduces the host exposure (and risk). The permission that will be selected will be the one which have the least impact on performance and the maximum impact on exposure reduction. Thus the permission which currently has the maximum $\frac{w}{f}$ (and whose predicates are not already in use) will be selected. Similarly, when the system is over-constrained and the exposure (and risk) can be allowed to rise, then the permission with the minimum $\frac{w}{f}$ will be selected. This is because this will provide the maximum improvement in performance. Experiments validating this approach are described in [Gehani03].

III. Implementation

To realize our model of evaluating predicates prior to granting permissions, we augment a

conventional access control subsystem by interceding on all permission checks and transferring control to our **ActiveMonitor**. If an appropriate binding exists, it delegates the decision to code customized to the specific right. Such bindings can be dynamically added and removed to the running **ActiveMonitor** through a programming interface. This allows the restrictiveness of the system's access control configuration to be continuously varied in response to changes in the threat level.

Our prototype was created by modifying the runtime environment of Sun's Java Development Kit (JDK 1.4), which runs on the included stack-based virtual machine. The runtime includes a reference monitor, called the **AccessController**, which we altered as described below.

A. Interposition

When an application is executed, each method that is invoked causes a new frame to be pushed onto the stack. Each frame has its own access control context that encapsulates the permissions granted to it. When access to a controlled resource is made, the call through which it is made invokes the **AccessController**'s *checkPermission()* method. This inspects the stack and checks if any of the frames' access control contexts contain permissions that would allow the access to be made. If it finds an appropriate permission it returns silently. Otherwise it throws an exception of type **AccessControlException**.

We altered the *checkPermission()* method so it first calls the active **ActiveMonitor**'s *checkPermission()* method. If it returns with a `null` value, the **AccessController**'s *checkPermission()* logic executes and completes as it would have without modification. Otherwise, the return value is used to throw a customized subclass of **AccessControlException** which includes information about the reason why the permission was denied. Thus, the addition of the **ActiveMonitor** functionality can restrict the permissions, but it can not cause new permissions to be granted. Note that it is necessary to invoke the **ActiveMonitor**'s *checkPermission()* first since the side-effect of invoking this method may be the initiation of an exposure-reducing response. If it was invoked after the **AccessController**'s *checkPermission()*, then in the cases that an **AccessControlException** was thrown, control would not flow to the Active Monitor's *checkPermission()* leaving any side-effect responses uninitiated.

Code that is invoked by the **ActiveMonitor** should not itself cause new **ActiveMonitor** calls, since this could result in a recursive loop. To avoid this, before the **ActiveMonitor**'s *checkPermission()* method is invoked, the stack is traversed to ensure that none of the frames is an **ActiveMonitor** frame, since that would imply that the current thread belonged to code invoked by the **ActiveMonitor**. If an **ActiveMonitor** frame is found, the **AccessController**'s *checkPermission()* returns silently, that is it grants the permission with no further checks.

B. Invocation

When the system initializes, the **ActiveMonitor** first creates a hash table which maps permissions to predicates. It populates this by loading the relevant classes, using Java *Reflection* to obtain appropriate constructors and storing them for subsequent invocation. At this point it is ready to accept delegations from the **AccessController**.

When the **ActiveMonitor**'s *checkPermsission()* method is invoked, it uses the permission passed as a parameter to perform a lookup and extract any code associated with the permission. If code is found, it is invoked in a new thread and a timer is started. Otherwise, the method returns `null`, indicating the **AccessController** use the static configuration decide if the permission should be granted. The code must be a subclass of the abstract class **PredicateThread**. A skeletal version is presented in Figure 2. This ensures that it will store the result in a shared location when the thread completes and notify the **ActiveMonitor** of its completion via a shared synchronization lock.

The shared location is inspected when the timer expires. If the code that was run evaluated to `true`, then a `null` is returned by the **ActiveMonitor**'s *checkPermission()* method. Otherwise a string describing the cause of the permission denial is returned. If the code had not finished executing when the timer expired, a string denoting this is returned. As described

```
public abstract class PredicateThread extends Thread{

    protected PredicateThread(Permission permission,
                                        Object lock);
    public void run(){

        if(condition) result=true;

        synchronized(lock){
            lock.notify();
        }
    }

    public boolean getResult();
}
```

Fig. 2. Skeletal version of **PredicateThread**

Access Type	Time
getSystemLoad()	0.34 ms
getTransmitted(eth0)	1.02 ms
getReceived(eth0)	1.01 ms
getFreeSwap()	0.39 ms
getFreeRAM()	0.39 ms

TABLE I
RUNNING TIME OF EXAMPLE PRIMITIVES.

above, when a string is returned, it is used by the modified **AccessController** to throw an **ActiveMonitorException**, our customized subclass of **AccessControlException**, which includes information about the predicate that failed. The thread forked to evaluate code can be destroyed once its timer expires. Care must be taken when designing predicates so that their destruction midway through an evaluation does not affect subsequent evaluations.

Finally, the **ActiveMonitor**'s own configuration can be dynamically altered. It exposes *enableSafeguard()* and *disableSafeguard()* methods for this. These can be used to activate and deactivate the utilization of the auxiliary checks for a specific permission. If a piece of code is being evaluated prior to granting a particular permission and there is no longer any need for this to occur, it can be deactivated with the *disableSafeguard()* method. Subsequently that permission will be granted using only the **AccessController**'s static configuration using a lookup of a binary value. Similarly, if it is deemed necessary to perform extra checks prior to granting a permission, this may be enabled by invoking the *enableSafeguard()* method.

IV. EVALUATION

In order to successfully perform an auxiliary check prior to granting a permission, enough time must be allocated for the code to run. However, this introduces a proportionate latency. This raises two issues. The first is whether reasonably complex predicates can be allocated within the time allocated. The second is the relationship between the maximum time allowed for a single permission check and the overall running time of an application. Various factors may mask the effect. In a multi-threaded application, blocking on a permission check may not affect overall performance if another thread is able to proceed. The latency introduced for a filesystem permission may be insignificant compared to the latency of the disk access.

To examine the first issue, we wrote a library to sample metrics such as processor load, memory usage and network utilization. It uses the Java *Native Interface* to access the underlying Linux operating system. By measuring the running time of a few primitives, and given a maximum time for evaluating a predicate, we can estimate how complex a predicate can be. From the measurements in Table I, we can see that if a predicate was allowed a slot of time on the order of $100 - 200$ milliseconds before its timer expired, then reasonably complex safeguards could be instituted.

We then turn to the second issue - the effect of time allocated for a single permission check on the overall running time of applications. To estimate this we carried out the following experiment. We gathered a suite of applications with which to test the running time of the system as the value of t as described in Section II-E was varied.

The applications used are from the *SPECjvm98* benchmark [SPECjvm98]. *check* exercises the virtual machine's core functionality such as subclassing, array creation, branching, bit operations, arithmetic operations. *compress* is a Lempel-Ziv compressor. *jess* is an expert system that solves puzzles using rules and a list of facts. *db* performs a series of add, delete, find and sort operations on a memory resident database. *mpegaudio* is an MP3 decompressor. *jack* is a lexical parser. *mtrt* is a ray tracer. The only application in SPECjvm98 that was left out

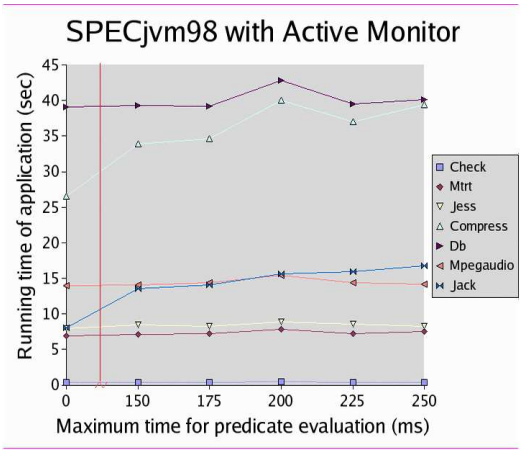

Fig. 3. Worst case impact on SPECjvm98 applications when the maximum time for predicate evaluation is varied.

was *javac* due to compatibility issues.

We instrumented the **Permission** class, then ran the benchmark suite once and determined that there were 5939 permission objects instantiated during the run. Of these, 4363 were **PropertyPermission("file.encoding")** objects, used to check the current default character set. Next, we found that 1443 were **FilePermisison** objects, used to access 487 different files. We chose to conduct our analysis by defining **ActiveMonitor** bindings for **FilePermission**'s. The worst cases occurs when the safeguards for all permissions have been activated and all the predicates in use are unable to complete in the allocated time. To examine this case, we defined the code so it would return only when the timer expired after t time has passed.

The graphs in Figure 3 show the running time of the SPECjvm98 applications as the bound on how long a permission's predicate could be evaluated for was increased. Based on this we can conclude that reasonably complex predicates can be evaluated with an acceptable impact on the running time of typical applications. The uninstrumented case is represented by the data point where the predicate is evaluated in 0 ms. Only two applications degrade noticeably, *compress* and *jack*, due to numerous **java.io.FilePermission** requests.

V. Related Work

Flexible access control has been studied in detail by the database community. Research has been done on making the checks dependent on the time of access, history of previous accesses, content of the record and structure of the query [Baraani96]. Databases, however, form a very structured domain and the range of threats (and therefore responses needed) is narrower than that of an operating system. [Campbell98] explored the use of active capabilities for mobile agents where signed scripts were used to gain access to a resource.

In the context of operating systems, research has been conducted into interposition of system calls to monitor for intrusive activity. [Uppuluri01] describes the use of finite state automata to describe specification based intrusion signatures that can be monitored for in real time using system call interposition and raises the possibility of invoking a response. [Fraser99] describes a wrapper language for wrapping generic system calls. We focus only on modifying access control related behaviour, restricting the domain in which semantics change, thereby reducing the scope for unintended side-effects.

Projects such as Flask [Spencer99] and RSBAC [Ott01] provide a general framework which address a similar goal. Our work differs in the following specifics. First, we seek to maintain the constant running time guarantee implicitly made by the operating system to an application for a permission check. Second, we seek to expose to the application through a programming interface the (potentially) richer semantics of a permission denial, so that the application has the option to adapt intelligently. Third, we aim to activate predicate evaluation only when the benefit warrants the cost. Fourth, the Active Reference Monitor is specifically designed to allow the activation and deactivation of predicate evaluation at the granularity of individual permissions without re-initializing the security subsystem.

VI. Conclusion

We have argued for the use of predicated permissions with temporal constraints on their running time, demonstrating empirically the performance impact of the approach. In addition, we describe how the changes in permission check semantics can be exposed to applications via a programming interface. Predicates can be dynamically altered to change the host's

level of exposure, a property that is needed when managing the risk of a system while it is running.

REFERENCES

[Baraani96] A. Baraani-Dastjerdi, J. Pieprzyk, and R. Safavi-Naini, Security in Databases : A Survey, Technical Report TR-96-02, Department of Computer Science, The University of Wollongong, Australia, 1996.

[Bell73] D. E. Bell and L. J. La Padula, Security computer systems : mathematical foundations, Technical Report FSD-TR-73-278, Hanscom Air Force Base, Bedford, MA, 1973.

[Brewer89] David F. Brewer and Michael J. Nash, The Chinese Wall security policy, Proceedings of the IEEE Symposium on Security and Privacy, pages 206-214, Oakland, CA, May 1989.

[Campbell98] Roy Campbell and Tin Qian, Dynamic Agent-based Security Architecture for Mobile Computers, Second International Conference on Parallel and Distributed Computing and Networks, 1998.

[Cuppens93] F. Cuppens, A logical analysis of authorized and prohibited information flows, Proceedings of the IEEE Computer Society Symposium on Security and Privacy, pages 100-109, May 1993.

[Denning76] Dorothy E. Denning, A lattice model of secure information flow, Communications of the ACM, 19(5):236-243, May 1976.

[Foley89] S. N. Foley, A model for secure information flow, Proceedings of the IEEE Computer Society Symposium on Security and Privacy, pages 248-258, May 1989.

[Fraser99] Timothy Fraser, Lee Badger, and Mark Feldman, Hardening COTS Software with Generic Software Wrappers, IEEE Symposium on Security and Privacy, May 1999.

[Gehani02b] Ashish Gehani and Gershon Kedem, NOSCAM : Sequential System Snapshot Service,CSDS 1st Computer Forensics Workshop, 2002.

[Gehani03] Ashish Gehani, Support for Automated Passive Host-based Intrusion Response, PhD thesis, Duke University, 2003.

[Harrison76] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman, Protection in operating systems, Communications of the ACM, 19(8):461 471, August 1976.

[Jones76] A. K. Jones, R. J. Lipton, and L. Snyder, A linear time algorithm for deciding security, Proc. 17th Annual Symp. on Foundations of Computer Science, 1976.

[Lampson71] B. W. Lampson, Protection, 5th Princeton Symposium on Information Sciences and Systems, Princeton University, March 1971.

[Ott01] Amon Ott, Rule Set Based Access Control Linux Kernel Security Extension, 8th International Linux Congress, November 2001.

[Peri96] R. Peri, Specification and Verification of Security Policies, PhD Thesis, University of Virginia, 1996.

[Sandhu92] Ravi S. Sandhu, The typed access matrix model, Proceedings of the IEEE Symposium on Research in Security and Privacy, pages 122 136, Oakland, CA, May 1992.

[SPECjvm98] www.specbench.org/osg/jvm98/

[Spencer99] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Anderson, and J. Lepreau, The Flask Security Architecture: System Support for Diverse Security Policies, Proc. of the 8th USENIX Security Symposium, Aug. 1999.

[Uppuluri01] Prem Uppuluri and R. Sekar, Experiences with Specification-Based Intrusion Detection, Lecture Notes in Computer Science, 2001.