Copyright © 2003 by Ashish Gehani
All rights reserved

SUPPORT FOR AUTOMATED PASSIVE HOST-BASED INTRUSION RESPONSE

by

Ashish Gehani

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Gershon Kedem, Supervisor

_____
Carla Schlatter Ellis

_____
Alexander Hartemink

_____
Campbell Harvey

_____

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2003

ABSTRACT

(Security)

SUPPORT FOR AUTOMATED PASSIVE HOST-BASED INTRUSION RESPONSE

by

Ashish Gehani

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Gershon Kedem, Supervisor

_____
Carla Schlatter Ellis

_____
Alexander Hartemink

_____
Campbell Harvey

_____

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University

2003

Abstract

Vulnerabilities continue to be discovered with high frequency. Threats that exploit them can be recognized by intrusion detectors. Manual response, however, is becoming decreasingly tenable. We introduce a model for automatic real-time mitigation of the risk posed to a host. The model is derived from an extant risk analysis framework used by the information assurance community, applying it to the operating system paradigm. We describe runtime support for implementing the scheme.

SADDLE provides an auditing architecture that allows high fidelity auditing for intrusion detection with limited computational load and storage requirements. ARM modifies the reference monitor to dynamically constrain permissions to control the probability of exposing threatened resources. RICE allows guarantees to be made about the confidentiality, integrity and availability of data after a penetration occurs. NOSCAM provides a service for pro-active gathering of forensic evidence for postmortem analysis of an attack. These systems are combined through a prototype response engine, RheoStat, whose utility is demonstrated using a set of synthetic attacks.

Contents

List of Tables

List of Figures

Acknowledgements

It has been my good fortune
to have family, friends and faculty
who have afforded me the opportunity
to prolong the wonder of youth
and its innocent search for truth.

As the frontiers I explore
are set to change once more
to ones I haven't seen before,
I'd like to take this chance
to thank all those around me
who have helped me to advance.

Chapter 1:
Introduction

1.1 Attacks

ARPAnet, the precursor to the Internet, was designed to allow communication while under attack. Survivability of the network in the face of node failure was a high priority. The security of a single node and the accountability that would assist in maintaining this, however, was a low priority [Clark88]. With the Internet's rapid growth has come a proportional increase in exposure to attackers.

The range of attack methodologies has been wide. They exploit errors in design, implementation, or configuration. A large number of defensive technologies have also been developed in response. These include virus scanners [Cohen85], firewalls [Chapman92], compiler extensions for hardening code [Cowan98], intrusion detectors [Denning87], cryptographic file systems [Blaze93], encrypted network connections [RFC2246], and formal security policy verifiers [Peri96].

However, 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.

1.2 Response

To maintain the security of a networked node, it must be continually monitored. When there is suspicion that an attack may be underway, it is prudent to effect a response. The first course of action would be to interrogate the runtime environment to obtain finer grain data to cross-check the audit information that raised the alarm. If the suspicion remains, the next step would be to reconfigure the system (potentially reducing functionality) to limit the exposure of portions that may be vulnerable to the attack in progress. Data that may be affected by the attack should be safeguarded. Measures to ensure its confidentiality, integrity and availability after a successful attack should be taken. Finally, an effort should be made gather and preserve forensic information from the environment that may not be available at a later point in time.

1.3 Automation

Maintaining security levels in the face of increased attacks requires a commensurate effort invested in response. As the frequency rises, so does the benefit of automation. Additionally, *time based security* [Schwartau99] argues that the time to detect and then react to an attack must be less than the amount of time afforded by the system's protection mechanisms. Automating the response reduces the onus placed on the protection subsystem.

Current intrusion detection technology does not yield a low enough false positive rate to allow unchecked *active responses* [Axelsson00]. Since there is a significant possibility of misidentifying activity as aggressive

when it is not, taking offensive action against the hypothetical attacker can yield new unwarranted attacks rather than stopping current ones.

Instead, we identified as a target of opportunity the case when an intrusion detector can identify an attack with an intermediate level of certainty. If all such activity were flagged as intrusive, the detector would cumulatively trigger false alarms at an unacceptably high rate [Axelsson99]. Hence, the detector will opt to allow this activity to pass unchecked. We argue that the intrusion detectors should instead have the option of effecting a *passive intrusion response*, defined as operations on resources within the owner's domain of administration, in such cases.

1.4 Risk

We equate protecting a system with minimizing the risk it faces. If we term the risk R, it is dependent on three factors. The first is the threat it faces. The probability that it faces a threat is termed T. If there is no threat to the system, then it is not at risk. The second is the vulnerabilities that exist in the system. The probability of these being exposed is termed V. If there are none, then even in the presence of a threat, no risk is posed. The third factor is the consequence of an attack succeeding. We term the cost of the consequence, C. If there is no consequence, the system is not at risk.

$$R = T \times V \times C \tag{1.1}$$

Threat is primarily defined by the attacker. T can be reduced through methods such as the use of egress filtering or disarming hostile hosts [Bruschi01] if T originates in a domain where the administrator is coop-

erative. However, recognition of T is the domain of the defender. Vulnerabilities and consequences fall within the control of and can therefore be reduced by the defender.

1.5 Mitigation

The adaptive auditing architecture, SADDLE [Gehani02a], was designed to allow intrusion detectors to obtain fine grained audit data. This allows better discrimination when searching for partial matches to intrusion signatures. Its role is to support finer granularity estimates of T and implicitly R. This is the subject of Chapter 2. Once such a match has occurred, there are at least three types of responses that can be effected.

The first type of response is the dynamic adjustment of the access control configuration of the system, so as to minimize the exposure of vulnerabilities, V, perceived to be relevant to current threats. The ARM subsystem, described in Chapter 3, describes programmable permissions, which use the context of the runtime environment to decide whether to grant a request.

The second type of response aims to guarantee the security of the data, since this limits the consequences, C, should the attack succeed. The RICE subsystem, described in Chapter 4, maintains the confidentiality, integrity and availability of data using cryptography and replication by taking action before an intrusion completes.

The third type of response is the preservation of evidence for subsequent examination. Since the risk model does not capture a fraction of the potential threats, there remains the possibility of a successful penetration

which effects damage. In the event this occurs, the forensic analyst will need the evidence to characterize the attack source. Chapter 5 describes the tool NOSCAM [Gehani02b], which can be used to effect this. It continuously monitors output from an intrusion detector, pro-actively gathers forensic information and then records it to an immutable medium.

We describe the formal framework for using the above set of response primitives in Chapter 6. The model is derived from an extant risk analysis equation used by the information assurance community, applying it to a host operating system paradigm. We then use this to guide the development of a prototype real-time risk management based intrusion response engine, RheoStat, that consists of a modified version of a SADDLE-aware intrusion detector, JStat, to dynamically modify ARM's permission set, RICE's file set, and NOSCAM's audit set.

Future avenues of research and lessons learned are described in Chapter 7 along with concluding remarks.

Chapter 2:
Adaptive Auditing Architecture

2.1 Overview

Intrusion detection uses anomalous activity in a system to recognize an attack. Toward this end, events are audited and resources are monitored, with the logs thus generated serving as the input to an analysis engine. Increasing the fidelity of recognition implicitly requires refining the granularity of logging. Current systems can achieve this at the cost of increased computational load and storage space. SADDLE provides an architecture that alleviates the apparent tension between the aforementioned goal and its associated cost.

2.2 Background

A prototypical *intrusion detection system* consists of several components. *Sensors* record events in the system or traffic on a network link, committing the data to *log files*. This is fed into an *analysis engine*, which uses a *signature database* to decide when an intrusion has occurred. The database can consist of a set of bounds on statistical measures. If current activity strays beyond the bounds, the analyzer concludes that *anomalous behaviour* is occurring. Alternatively, the database can contain *misuse rules* that each describe a set of events that leads to an access control breach. If current activity matches a rule, the engine flags an intrusion. A third option uses *application specifications* of expected behaviour. If a process's events deviate from its specification, an alarm is raised. An *on-*

line analyzer processes log entries at the rate they are generated, while an *offline analyzer* does the pattern matching later.

[Denning87] framed the model used by current intrusion detectors. Numerous schemes have been tried for intrusion recognition [Axelsson00]. However, intrusion detectors are still not ubiquitously deployed. *False positive rates*, or how often an alarm is raised in the absence of a breach in security, are measured relative to the maximum number of possible false positives (which is proportional to the number of audit records). Intuitively, the utility of a detector diminishes if there is an increase in the proportion of the alarms that are raised but turn out to be false. [Axelsson99] uses the base-rate fallacy of Bayesian statistical estimation theory to argue that the criterion of interest should be the false positive rate relative to the number of expected intrusions instead. Current intrusion detection systems fare poorly using this metric. Thus there still exists a need to decrease false positive rates.

2.3 Refining Detection Signatures

There exists a general method to reduce the frequency with which normal system activity is incorrectly flagged as intrusive. This can be achieved by increasing the specificity of the signatures used for detection. For example, for anomaly detection using statistical signatures, sampling can occur at a greater frequency, derivatives of the frequency can be utilized, or cross-products of different metrics can be used. For rule-based and specification-based signatures, in addition to the class of event, parameters can be matched, temporal constraints can be imposed, or filters based

on sequence history can be used. A combinatorial analysis verifies that the expected value of the false positive rate can be drastically reduced by such an approach.

The disadvantage of using more detailed signatures is two-fold. Systems based on detecting anomalous behaviour may experience an increase in false negatives. Since a signature that is less specific may match more subsequences of the audit record, the chances of it matching a real intrusion are higher. The increase in specificity makes the signature inherently more brittle in the face of variation in the attacker's statistical profile. On balance, the reduced false positive rate is likely to outweigh the increased false negative rate. Rule-based and specification-based detectors, however, are not adversely affected. A more specific rule or specification is constructed using *a priori* knowledge of a particular attack. Therefore, a real attack will continue to be detected, while a previous false positive may no longer match. The rate of false negatives is monotonically non-increasing with a rule's specificity.

The second concern is the increase in storage and processing overhead that is imposed on the auditing subsystem. To refine a signature used to detect anomalous behaviour, more detailed audit data is needed. For example, improving temporal fidelity involves increasing the frequency of sampling the variables of interest, such as the CPU, filesystem or network load. This results in a commensurate increase in the storage space used as well as the processing done initially for logging, and subsequently for parsing and filtering, the audit data. In the case of rule-based signatures, refinement can be effected through the use of more contextual events. As more events are logged, the storage overhead increases. Refinement can

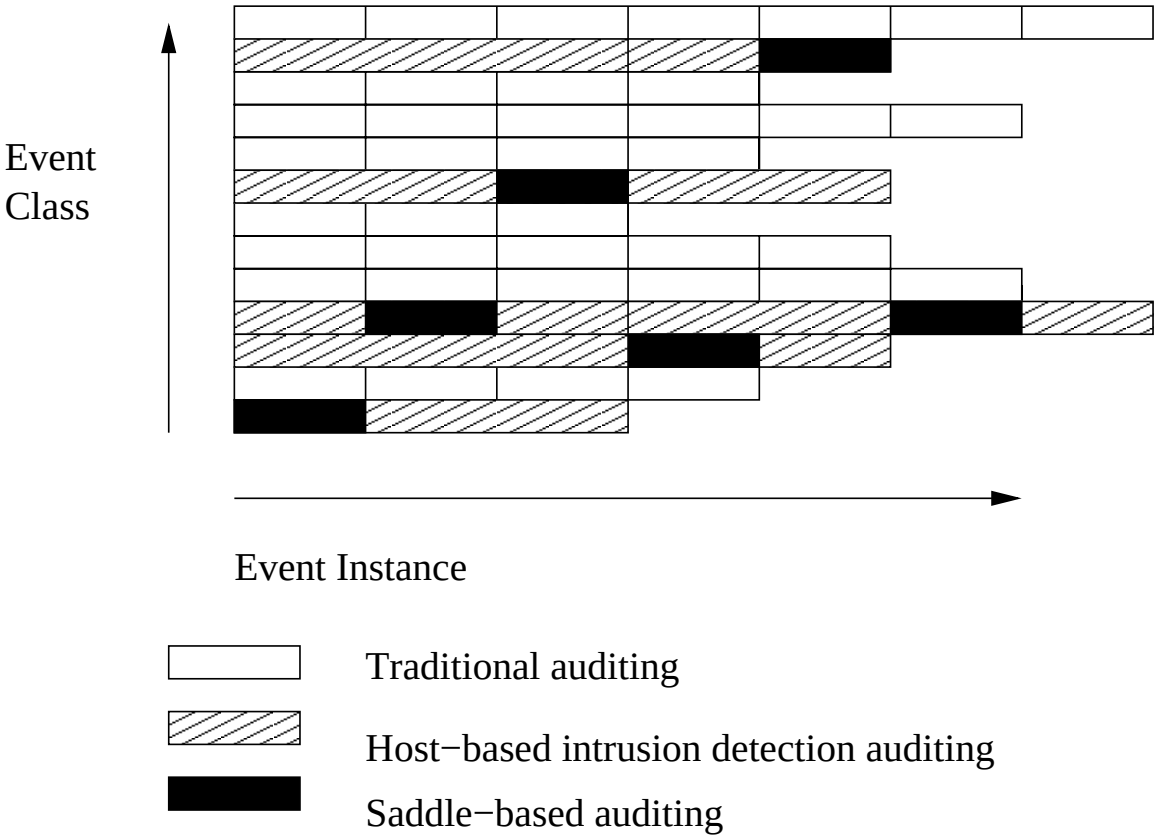

Figure 2.1: SADDLE allows the audited event set to be honed dynamically. Each horizontal bar represents the set of instances generated that are of a single event class. Events that are marked in white are generated by a traditional auditing subsystem which does not discriminate on the basis of an event's relevance to intrusion detection. Events that are marked in grey are generated by a typical host-based intrusion detector's auditing subsystem (in addition to being generated by a traditional auditing subsystem). Events that are marked in black are generated by a SADDLE-compliant system (in addition to the two other types of auditing subsystems). SADDLE-compliant systems audit the event if it is *currently* relevant to the intrusion detector's signatures. Each class is progressively more discriminating in the events it chooses to audit, reducing performance and storage overhead.

also be achieved by increasing the details collected for each event, such as inclusion of parameters along with system calls. This increases the runtime cost of processing an event significantly. For example, the use of a tool that traces each system call as it is invoked imposes enough overhead that it renders any large application unusable in practice.

2.4 Design

Using more detailed intrusion signatures does not inherently require a significant performance penalty. Rather, the impact described in the previous section is due to the architecture of current auditing subsystems. The first issue is the coarse granularity at which they can be manipulated. Logging involves all instances of a class of event. If the intrusion detection system requires a detailed version of one instance, detailed logging is turned on for all instances implicitly. The second issue is the static nature of determining whether a particular audit record should be generated. An intrusion detection system may only need to know about a small subset of all events at any given instant. Yet all events that may ever be of interest are logged. The final issue involves extensibility. If a system was designed to sample a variable, such as the processor load, at a fixed interval and later a change is required, no methods exist to implement this. The change may be quantitative such as the change in the sampling interval, or qualitative, such as the need to obtain the current rate of change of the variable instead of its current value.

The above limitations can be seen to derive from the unidirectional nature of the communication between the logging facility and the intru-

sion detector. It is possible to remedy the problems through the use of a reverse channel through which the detector can specify to the auditing subsystem exactly what information it currently needs. It should be possible to dynamically modify the description of what information should be audited. A programmable interface should be present to allow the filtration and processing of audit data before it is committed to the log record. This also facilitates automated changes instead of manual reconfiguration. [Bishop96] describes a formal procedure for deriving exactly which pieces of data must be logged to assure the security of a particular protocol. Event pattern recognition can occur in either the auditing subsystem being described or by the intrusion detector, depending on the event filters defined. Depending on the specifics of a rule, the right choice can be either one. The architecture must be flexible enough to handle this. A simple programming interface is needed which can be invoked by an intrusion detection application, and is amenable to the above constraints.

2.4.1 Audit Registry

There may be multiple audit data producers and consumers, with potential duplication of requests from consumers. A means for these streams to interact is needed. We term a process that provides such a service the Audit Registry. It will expose a programmable interface that intrusion detection applications can invoke. It will also convert the collection of requests into a canonical form and then initiate and terminate fine-grained auditing as needed. An architecture that conforms to this description effects System Auditing via Demand Driven Logging of Events (*SADDLE*).

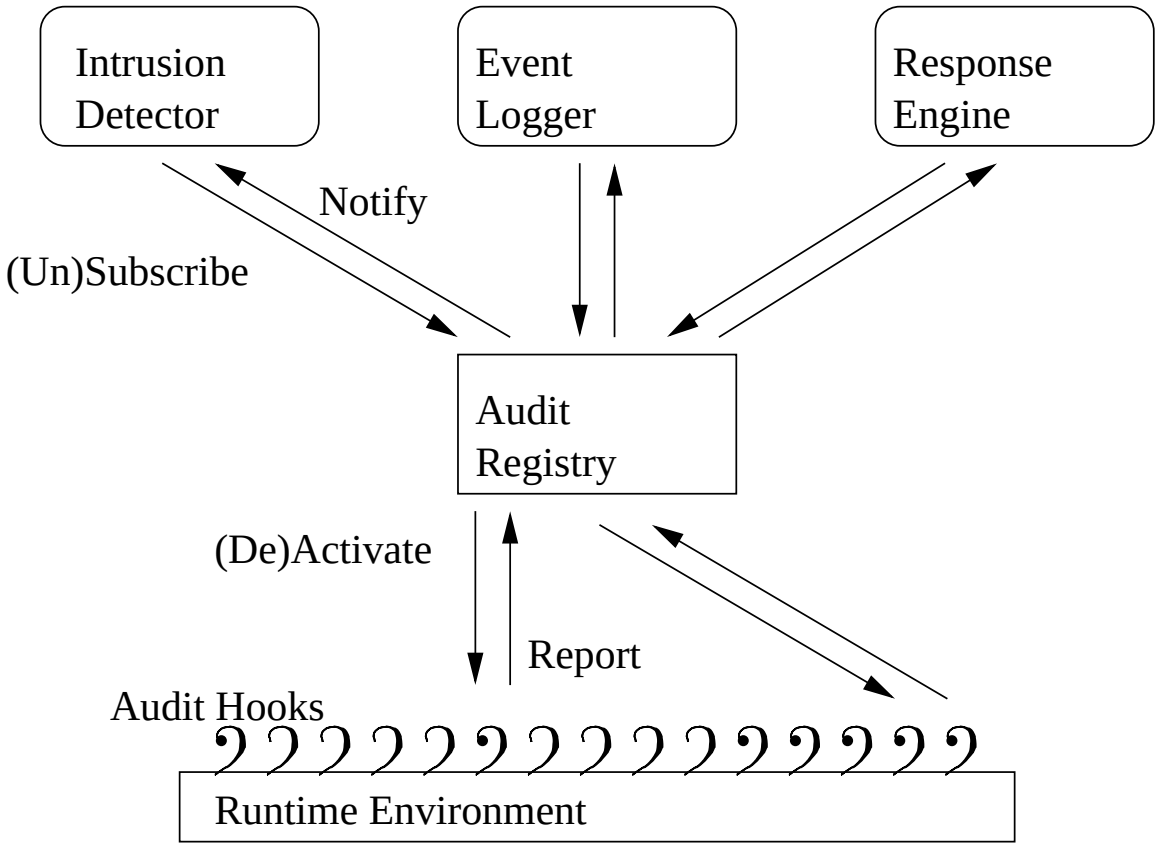

Figure 2.2: SADDLE architecture - Audit Registry correlates requests from applications and activates runtime hooks only when the need exists.

2.4.2 Audit Trail Producers

A design choice must be made regarding the audit trail producers. The data can come from the network or the host. Typically data derived from the network is sampled from a raw stream. Stateful packet inspection can potentially allow more complex determinations to be made as well. The design of the architecture should not preclude the use of this data. However, in the absence of clear security semantics, data from such a source may be unreliable.

Audit data gathered on a host can come from trusted sources such as the kernel, system libraries in user space, or applications whose trustworthiness can be established by a mechanism external to the scope of the architecture. Checking the user or group identity, or verifying an authentication credential are typical ways to decide whether to trust a program. Audit data from the operating system has several advantages. Its format can be controlled regardless of the application that is executing. The semantics of the audit record are clear. It is trusted.

A potential advantage of log data from applications is the richer semantics available. However, this comes with the tradeoff that this data is usually generated for the purposes of debugging rather than security, may not conform to an audit record specification, and may not be trustworthy since it is not typically feasible to verify the runtime security properties of an application without the use of a scheme such as proof-carrying-code [Necula97].

2.4.3 Audit Trail Consumers

The Audit Registry continuously accepts the event streams generated by the audit trail producers. It must mediate access to this data. We propose that it should expose an interface of the following form that can be invoked by applications such as intrusion detectors:

registerCallback(Event event, Callback callback, Boolean logInFile);

unregisterCallback(Event event, Callback callback);

When an *event* occurs, the handler *callback* will be invoked with *event* as a parameter. This requires *callback* to have been defined to conform to to a predefined specification. If *logInFile* is true, then the event will be appended to the *persistent audit trail*, stored on disk. This interface allows intrusion detectors without support for a SADDLE architecture to use an audit trail as usual through the use of wildcard events and null callback handlers. A SADDLE-compliant detector instead uses the callbacks to get exactly the data it needs directly, even turning off requests for logging to a file to reduce overhead when fine-grained signatures are used.

2.4.4 Callback Chains

Each time an event of interest occurs, a callback is made to the applications that had subscribed to the event. An application can then utilize the information and take further action. In the case of an intrusion detector, the most frequent choice will be to unregister interest in the event and then register interest in another event. However, latency between the unregister and register operations could allow an event of interest to occur between. This would cause an intrusion detector to remain uninformed

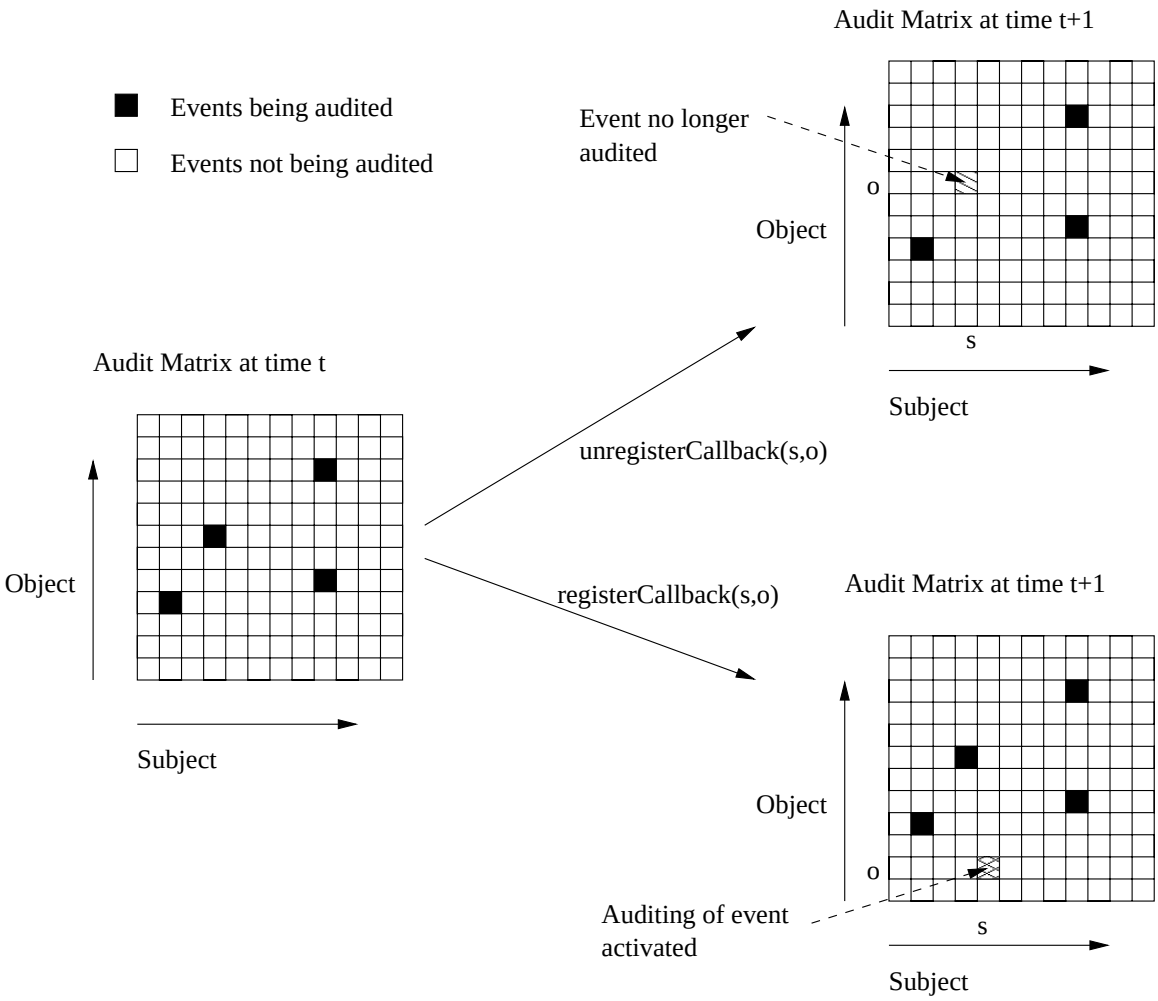

Figure 2.3: The Audit Control Matrix has two axes representing subjects and objects in a system. When a subject's acess to an object is to be audited, the corresponding cell in the matrix is defined to be true. Otherwise is defined to be false. The configuration of the matrix is statically defined in a traditional auditing subsystem. Using the **AuditRegistry**'s *registerCallback()* interface, cell's can be activated dynamically. Similary, the **AuditRegistry**'s *unregisterCallback()* can be used to remove an entry from a cell in the Audit Control Matrix.

about an event, creating a false negative.

One method of resolving the issue is to pass a lock back to the application along with the callback. After it has unregistered the event and registered any others of interest, it can surrender the lock. However, this method suffers from the drawback that the latency of a call will depend on the length of time that an application opts to hold the lock for. Since multiple threads may obtain locks that they are mutually dependent on, this scheme allows deadlock to occur. In addition, if an application refuses to surrender a lock, it can starve other processes. Thus, this scheme is untenable.

Instead, we introduce the notion of a *callback chain*, where each callback is replaced with a subsequent one by the Audit Registry itself. Since all events are reported to the Audit Registry, by synchronizing the relevant data structure internally, it can ensure that unregistering a callback and registering its successor is an atomic operation. This prevents the reporting of an event between the two operations. To effect this, each callback can designate another callback as its successor. When an event triggers a callback, if it has a successor the Audit Registry invokes the callback, then subsequently unregisters it and replaces it with its successor. Thus, a chain of callbacks can be created, which recognize a consecutive series of events, without the possibility of an event of interest occurring in the system but not being reported to the audit trail consumer.

2.5 Implementation

We now describe the implementation of SADDLE on two different platforms. In both cases, the implementation is restricted to the use of mandatory logging in the operating environment and does not handle authenticated application generated audit trails.

The first implementation modifies the Java runtime environment (version 1.4). Auditing functionality is added to a subset of the methods in the classloader, access controller, filesystem and networking code. When a class is loaded, a privileged action is performed, a file is opened, a network connection is made or accepted, or when a network service is started, the Audit Registry is informed.

The second is done on Linux (with kernel 2.2.12). The kernel auditing facility is used to monitor a subset of the system calls that are analogous to those modified in the Java runtime. This is effected by inserting a kernel module that registers its own wrapper functions in place of the relevant system calls. The wrapper function audits the call, and then invokes the original call. The use of this module allows us to monitor when a privilege elevation occurs, when a program is executed, when a file is opened, when a network connection is made to or from a remote host, and when a program starts a network service.

Both the modified platforms now have auditing code that communicates with a common implementation of the Audit Registry described in the previous section. In addition, a SADDLE-compliant intrusion detector, JStat, modeled after the state-transition approach of [Ilgun95] is constructed with the input events all being members of the set described

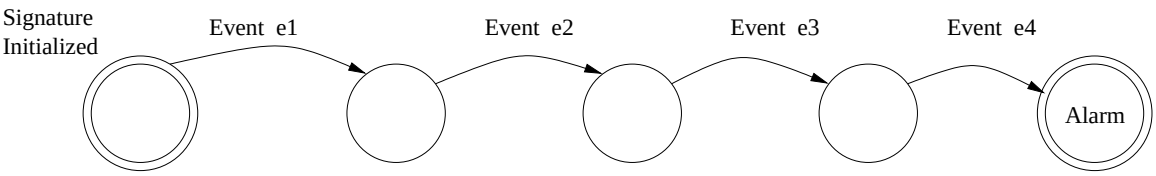

Figure 2.4: Modeled after STAT, JStat's intrusion detection signature consists of a series of states. Each state encapsulates a condition. When it becomes true, the signature's current position is advanced to the next state. When it reaches the last state an intrusion is deemed to have been detected and an alarm is raised.

above.

2.6 Evaluation

We now describe three experiments undertaken to evaluate the utility of SADDLE. First, we compare the storage requirements of generic auditing, auditing entire classes of events relevant to signatures, and auditing just the instances needed for correct operation of the intrusion detector. This enables us to characterize the benefits of using SADDLE-based selective auditing. Next, micro-benchmarks are performed on some of the calls that were modified to include auditing functionality. Finally, a macro-benchmark is run with a SADDLE-compliant state transitional analysis intrusion detector while varying the number of signatures. This allows the characterization of the scalability of a SADDLE-based system. The first and third are performed by driving a web server with a set of requests, since this is an application that exercises the subsystems that we have modified.

The first experiment measures the size of the audit trail generated in three cases: (i) when all the system calls in our audit set are logged, (ii) when only system calls that are present in intrusion detection signatures

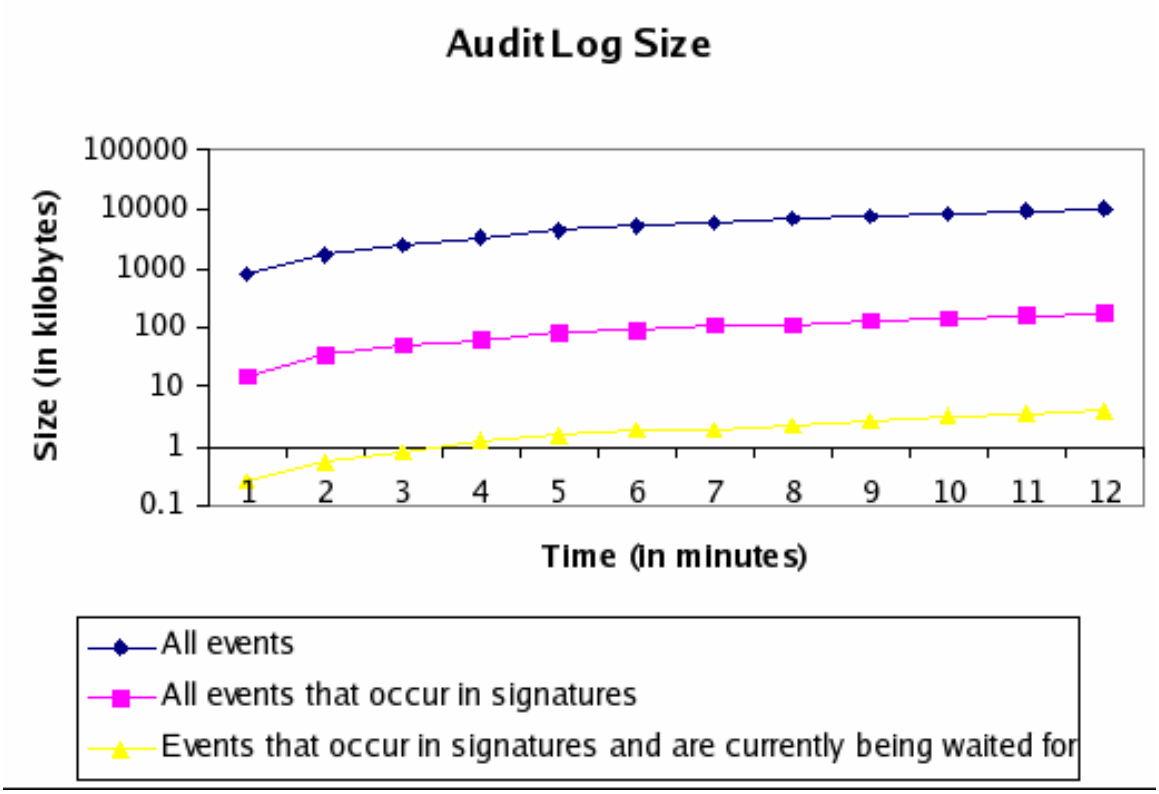

Figure 2.5: Audit trail size reduction is effected via automated filtering of which system calls to log.

are logged, and (iii) when relevant system calls are logged only when they are currently being watched for by the intrusion detector. A set of 10 signatures, averaging 5 events in length, was used. The workload was generated by using [SPECweb99] with 10 clients and 1 Apache web server (version 1.3.9). The graph in Figure 2.5 shows the drastic reduction in audit trail size by filtering out classes of events that will never match any event in the intrusion detection signature database, and the further filtration by temporal relevance. The third type is effected using a SADDLE-based JStat. We see the utility of SADDLE when fine grain auditing (such as the system calls instrumented) is required and the intrusion detector is to operate in real-time on data from a heavily loaded node such as a web server.

	Java	Java with Saddle	Linux	Linux with Saddle
File open	0.5 ms	4 ms	60 μs	120 μs
Network socket	150 ms	160 ms	10 ms	12 ms
Load class/binary	15 ms	20 ms	7.2 ms	7.8 ms

Table 2.1: Impact of SADDLE on system call performance.

The modifications made in the runtime environment of Java and Linux introduce overhead. We wished to examine the extent of this. We picked the 3 most frequently invoked calls - opening a file, creating a network socket to a remote node, and starting the execution of a binary, and measured their time with and without the SADDLE-based system in place. Network connections were made to local virtual hosts to avoid variation in network conditions, and Java classes (that were loaded) and Linux binaries (that were executed) were generated with no other operations in them. The results are summarized below.

Initial implementation was done on the Java platform and synchronized calls by holding a lock in the Audit Registry to guarantee consistency in the Audit Registry's view of the sequence of events. This was improved in the Linux implementation through the use of FIFO buffer to which calls were reported from where the Audit Registry read them allowing the caller to proceed without blocking on an Audit Registry lock.

Another aspect of interest was how the SADDLE-enabled JStat affected the performance of the workload as the number of intrusion detection signatures increased (and hence caused more instances of the modified system calls to be audited). To evaluate this we ran SPECweb99, varying the number of signatures in each run and noting how many requests were served per minute. From the graph in Figure 2.6 we see that the number

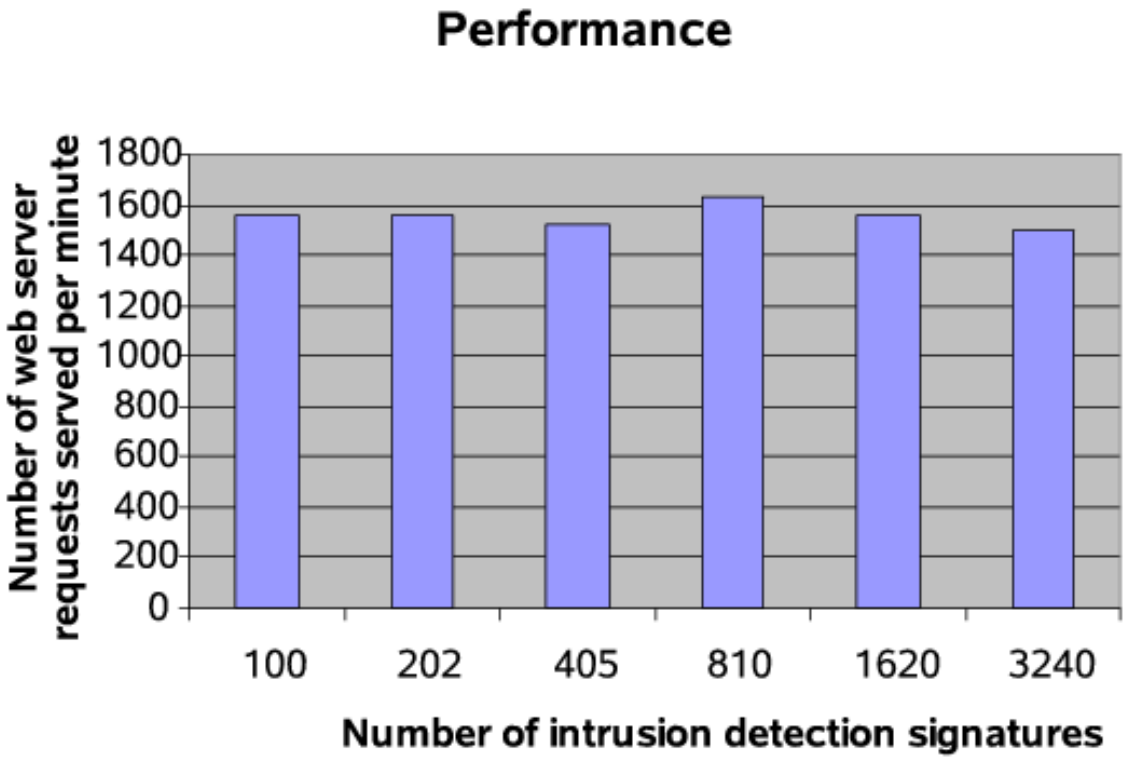

Figure 2.6: Workload performance is not affected as the number of detection signatures increases.

of signatures did not noticeably decrease the performance. Although the penalty paid on individual system calls can be high as seen above, the overall contribution of the set of audited calls is still not significant - allowing performance to scale with the use of a large number of intrusion detection signatures.

2.7 Related Work

[Anderson80] first used audit records to monitor for threats against Air Force computers. Most host-based intrusion detection systems still use logs[Axelsson00]. [Schneier99] tackles the problem of maintaining the integrity of log files on a host. [Sibert88] considered the problem of secure

auditing in a distributed system. [Chapman00] focused on reconstructing formal semantics from logs. [Roger01] implemented an online algorithm that checks the model defined by declarative constraints stated in a temporal logic against log records. Previous efforts have focused on log processing, while our work hones log generation for intrusion detection.

2.8 Summary

The benefits of the use of SADDLE are threefold. First, by obviating the need to store all events of a class, it allows very fine grain auditing while maintaining reasonable storage requirements for the log files. Next, by migrating the decision whether to log closer to the point of event generation, there is a reduction in the propagation of unused data through the (typically slow) I/O subsystem, thereby potentially reducing the running time of applications. Finally, there is a reduction in the administrative burden of manual selection of event granularity and determination of which subset of events to audit.

Chapter 3:
Dynamic Defense Decisions

3.1 Overview

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.

3.2 Background

A range of defensive technologies exist that address specific threats, such as cryptographic file systems [Blaze93], encrypted network connections [RFC2246], compiler extensions for hardening code [Cowan98], and formal security policy verifiers [Peri96]. However, they are not designed to address a constantly changing set of threats.

Other security tools do address this variation, such as virus scanners [Cohen85], firewalls [Chapman92], and intrusion detectors [Denning87] by focusing on recognizing a pre-configured set of threats. However, these

tools limit their action to raising an alarm. We argue that it is possible to further manage the risk by dynamically reconfiguring the permissions of a system.

3.3 Limiting Exposure

Security research has focused on finding and removing sources that contribute to V, such as buffer overflows, and tools that reduce C in the presence of T and V, such as firewalls. (Recall that T and V refer to the probabilities of threats and vulnerabilities being exposed, respectively, and that C refers to cost of consequences for the system as described in Section 1.4.) These methods are dependent on manual intervention for reducing the values of V and C. Automating the intervention would significantly reduce the *mean time to response* (*MTTR*). This would decrease the cumulative exposure of the system. Therefore, it is our aim to provide system support to facilitate automated reduction in V.

3.4 Design

3.4.1 User Space Response

One option for automating intrusion response is the use of the NOSCAM [Gehani02b] pro-active 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.

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.

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.)

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.

3.4.2 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.

3.4.3 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.

3.4.4 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 specify-

ing 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.

$$M = S \times O \times A, \quad \text{where} \quad p(i,j,k) \in M \tag{3.1}$$

The space M is populated with elements of the form:

$$p(i,j,k) = 1 \tag{3.2}$$

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

$$p(i,j,k) = 0 \tag{3.3}$$

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

$$p(i,j,k) = \sigma, \quad \text{where} \quad \sigma \in L \tag{3.4}$$

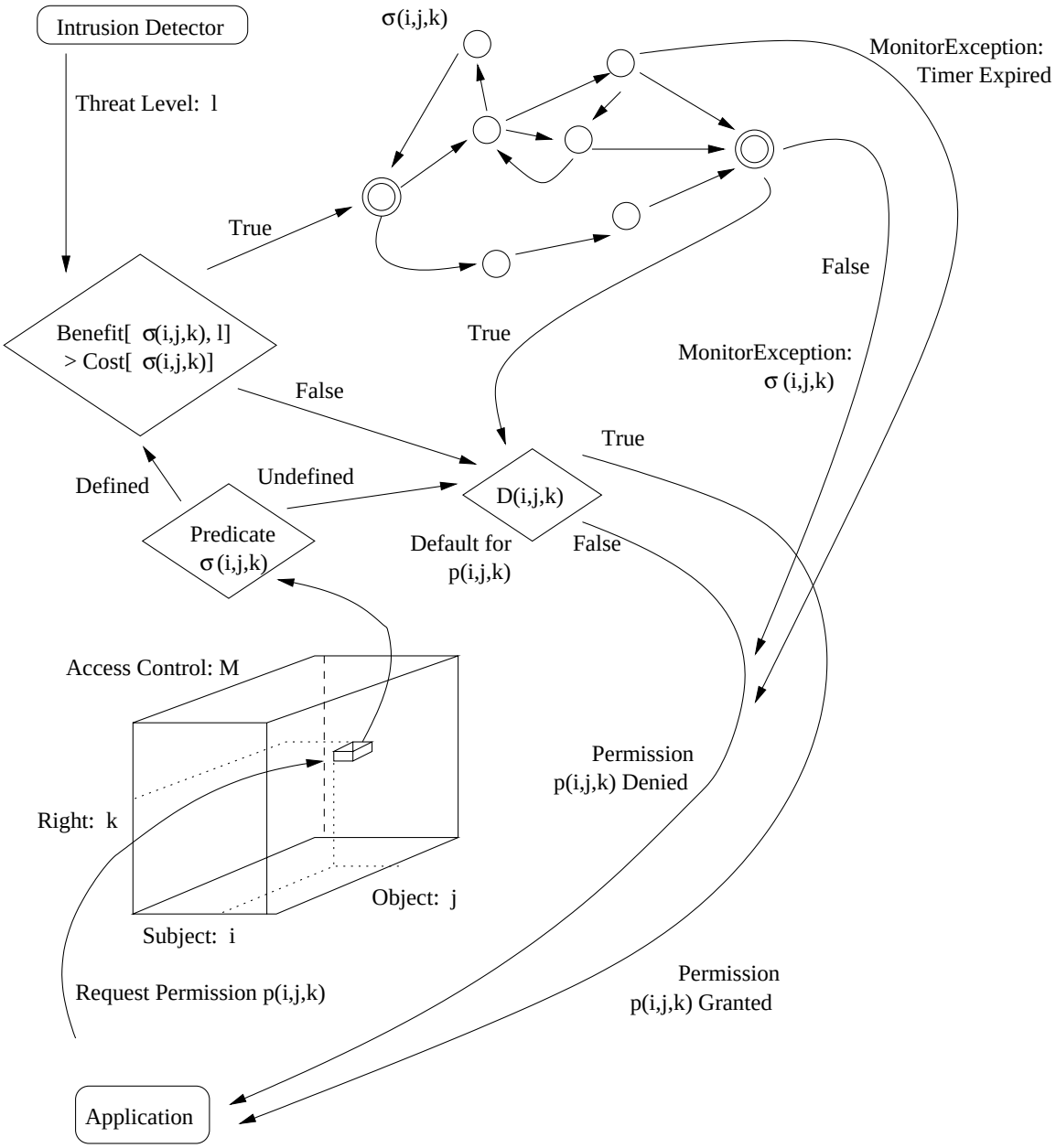

Figure 3.1: ARM (Active Reference Monitor) predicated permissions have 3 distinguishing features: (i) Constant running time, (ii) Dynamic activation if expected benefit exceeds cost, (iii) Interrogatable for cause of denial.

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.

3.4.5 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.

3.4.6 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 three factors that impact the choice.

The first is the running time of the predicate, t. The next factor is the frequency, w, with which that permission gets evaluated in a typical workload. If the expected value of t is larger, the cost associated with its use is greater. Similarly, if w is higher, the overhead imposed is greater.

The predicate evaluation will typically be activated based on 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 is termed, f. The greater the value of f, the more benefit there is to tightening the permission in question.

Thus, the cost of activation, $a(p)$, for a permission p, is given by:

$$a(p) = \frac{f}{w \times t} \tag{3.5}$$

If the current level of threat to the system is l, then a threshold $b(p, l)$ specific to each permission can be calculated. When

$$b(p, l) > a(p) \tag{3.6}$$

holds true, the predicate is evaluated before granting the permission. In this manner it is utilized only when the benefit from doing so is warranted by the circumstances.

3.5 Implementation

We chose to augment a conventional reference monitor by interceding on all permission checks and transferring control to our **ActiveMonitor**. This subsystem delegates the decision to code customized to the specific right if an appropriate binding exists. Such bindings can be dynamically added and removed to the running **ActiveMonitor**. The goal is to continuously vary the restrictiveness of the system's access control configuration, with minimal overhead, in response to the changing potential of an intrusion occurring. Salient points of the implementation are described below.

3.5.1 Reference Monitor Modification

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 here.

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 value of `false`, a customized subclass of **AccessControlException** is thrown. If it returns with a value of `true`, the **AccessController**'s *checkPermission()* logic executes and completes as it would have without modification. Thus, the addition of the **ActiveMonitor** functionality can restrict the permission set, 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 a 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.

3.5.2 Mapping Permissions to Predicates

When the **ActiveMonitor**'s *checkPermission()* is invoked, it must locate and invoke the statement upon whose successful evaluation the dynamic check is to be predicated on. It does this by using the permission as a key to do a lookup in a hash table, termed the Active Table. If it finds a mapping, it extracts the code found, invokes it and returns the answer it obtains to the caller. If no predicate exists, it returns true. The hash table is currently defined for the entire system, causing the **ActiveMonitor** checks to be effected globally.

If deemed necessary, the system could be extended to allow users to customize access by others for resources they own. This could be implemented by adding a second hash table consisting of mappings to code defined by users for guarding access to resources that they own. If the global check passed, the resource's owners custom code would also be invoked.

3.5.3 Privileged Predicates

We opted to define a core set of functionality that could be used by the code run by the **ActiveMonitor** in deciding whether to grant a permission. The first reason is that this is functionality that needs system support in the form of privileged execution, and the second is that it is likely that these functions would otherwise be redundantly defined independently in different predicates, increasing the load on the system with no benefit. The first element is a library of methods that can be used to interrogate the system about runtime conditions such as the average processor load, free mem-

ory, free swap space, and the amount of data received and transmitted by each network interface.

The second element is support for a database of intrusion signatures predefined by the administrator. A call can be made checking whether a particular signature has been matched by system activity thus far, allowing the permission to be denied if it has. Alternatively the signatures can be used to represent specifications of correct behaviour in which case the signature would be required to evaluate as true to allow the permission to be granted.

The signatures are based on the state transition scheme defined in [Ilgun95]. We implemented this in a separate component called JStat and modified a set of methods in the Java runtime to generate events that are fed to JStat. Currently the modifications of the runtime cover file reading and writing, network client and server socket creation, class loading, execution in privileged mode, exceptions and errors.

3.5.4 Active Monitor

When the system initializes, the **ActiveMonitor** first creates and populates the Active Table, the hash table mapping permissions to predicates, by loading the relevant classes, using reflection to obtain appropriate constructors and storing them for subsequent invocation. It also reads in the database of intrusion signatures for JStat. At this point it is ready to accept delegations from the reference monitor. When the **ActiveMonitor**'s *checkPermsission()* method is invoked, it uses the permission passed as a parameter to perform a lookup in the Active Table, and extract any code associated with the permission.

If code is found, it is invoked in a new thread and a timer is started. When the code completes, the output value is stored in a shared hash table, using the specific permission instance as the key. When the timer expires the value stored is checked. If it succeeded, the method returns evaluating as true. If it fails, an **ActiveMonitorException**, our customized subclass of **AccessControlException**, is thrown, which reports the name of the predicate that failed. If the code has not completed in the allotted time, no value would have been stored and the **ActiveMonitor** will throw the same exception indicating that the permission is not to be granted. In general, the thread forked to evaluate the code can be destroyed at this point. (Although the Java Thread API currently allows this, stopping a thread is deprecated in the API, so care has to be taken when defining predicates so that they are guaranteed to terminate.)

3.6 Evaluation

The **ActiveMonitor** design allows the use of any intrusion detection methodology. To facilitate easy development of predicates, we provide support for state transition rule-based signatures in the form of JStat. These signatures may be used as part of a predicate that is evaluated immediately prior to granting a permission. The result of the use of JStat is that when a predicate checks if a sequence of events has previously occurred in the system it is effectively executing a single call to JStat to consult it on the current state a predefined automaton, so the response has low overhead that does not alter appreciably the running time of the permission check. This is due to JStat operating independently in the background, process-

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 3.1: Cost of evaluating runtime context predicate primitives.

ing events as they are generated.

A library of privileged predicates was also defined for use when the virtual machine is running on Linux. These use a native interface to sample metrics such as CPU load, memory usage and network utilization. The running time of some examples are listed below:

However, we still needed an estimate of how complex the predicates that are evaluated by the **ActiveMonitor** can be. 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 3.4.5 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 was *javac* since it requires an older JDK.

We instrumented the **java.lang.Permission** class, then ran the bench-

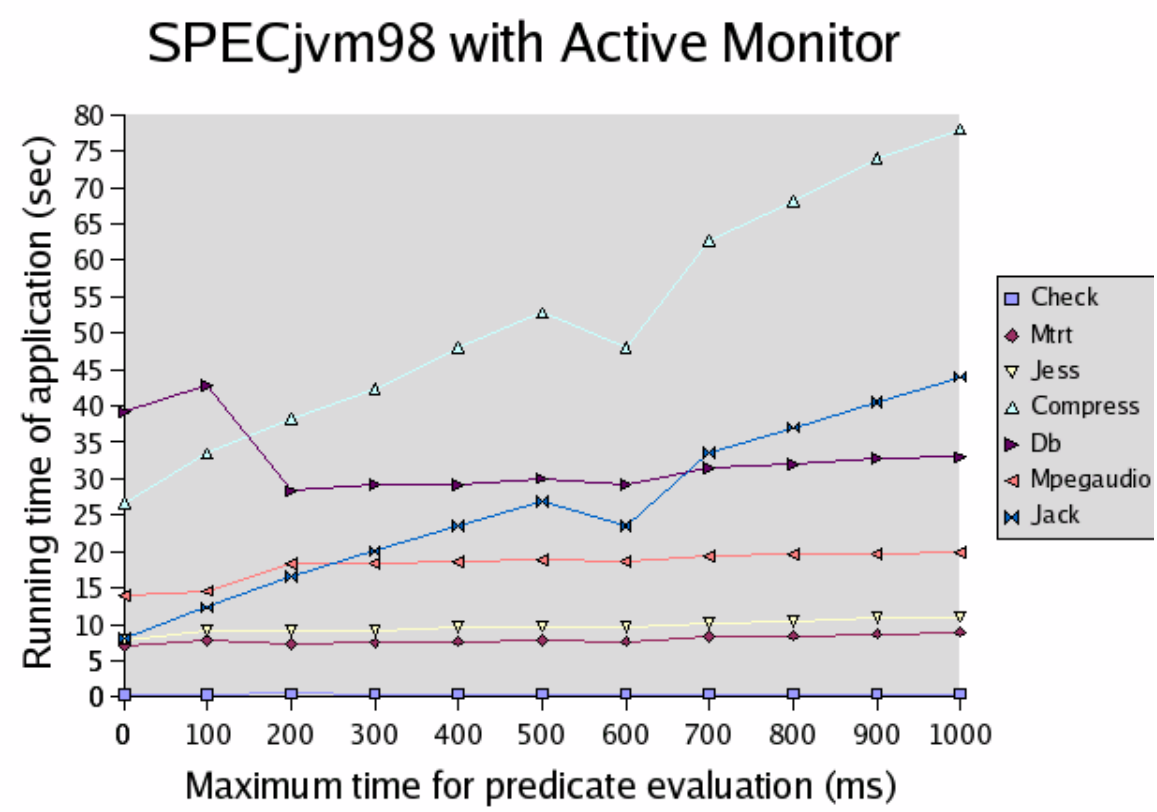

Figure 3.2: Impact on SPECjvm98 applications when the predicate evaluation time is varied.

mark suite once and determined that there were 5939 permission objects instantiated during the runs. Of these, 4363 were **java.util.PropertyPermission("file.encoding")** objects, used to check the current default character set. Next, we found that 1443 were **java.io.FilePermisison** objects, used to access 487 different files. Therefore, we chose to conduct our analysis by defining Active Monitor bindings for **java.io.FilePermission**'s. Since we were interested in the worst case, we defined the code so it would return when the timer would have expired after t time had passed.

The graphs in Figure 3.2 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 reason-

ably 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.

3.7 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 gen-

eral 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.

3.8 Summary

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.

Chapter 4:
Curtailing Consequences Cryptographically

4.1 Overview

Hosts connected to the Internet continue to suffer attacks with high frequency. The use of an intrusion detector allows potential threats to be flagged. When an alarm is raised, preventive action can be taken. One major goal of such action would be to assure the security of the data stored in the system. If this operation is to be effected manually, the delay between the alarm and the response may be enough for an intruder to effect significant damage. The alternative is to automate this response. We describe a subsystem that effects Response Initiated Cryptographic Encapsulation (RICE) of data that is deemed to be threatened by an attack. If it is activated when an attack appears likely to succeed, it guarantees the confidentiality, integrity and availability of threatened data even after a successful penetration occurs.

4.2 Background

Vulnerabilities in deployed software continue to be discovered and exploited by attackers. If possible, the error in design, implementation or configuration that results in a weakness ought to be addressed directly. However, in many environments, users need to install, manage and continue to use software that may introduce the vulnerabilities. Here alternative preventive measures must be utilized, such as firewalls and intrusion

detection systems. These tools allow an extra level of defense to be introduced to prevent exposure of the weaknesses that may exist in the system.

Efforts have been made to utilize information about the attack to take precautionary measures automatically, such as terminating processes or network connections. The approaches typically aim to cut off an attacker's access to the execution environment. The last line of defence is protecting the information stored itself, which is what we focus on.

4.3 Dynamic Data Protection

Computing systems are designed to manipulate data. It is this data whose security is paramount and the final target of protection. Our goal therefore is to assure that the data's confidentiality, integrity and availability is maintained, even after the system is penetrated. We aim to achieve this by providing a response primitive that can be invoked by an intrusion detection system when it detects an attack that is likely to succeed. Invoking the primitive must ensure that the data is encrypted to guarantee confidentiality, a hash of it is signed to allow its integrity to be subsequently verified, and it is replicated at another node so that it is available even if the local copy is lost.

Effecting the above mentioned operations on a large data set is a computationally intensive task. The latency of the operation would be too high if it were to be completely executed at invocation time. Our strategy to address this issue is to amortize the cost over the course of the life of the data. This is done by maintaining data in a protected state until an application needs access to it. At this point, transparent to the application, it is

exposed. When it is no longer used by any application, it is transparently protected once again. This process imposes an overhead whenever data is used, but it has the benefit of allowing data to be rapidly protected, by simply deleting exposed copies of the data and the cryptographic key material needed for the transparent manipulations. The scheme uses mechanisms similar to a cryptographic filesystem. The differences are highlighted in Section 4.7.

4.4 Design

4.4.1 Protection Groups

Protection groups allow predefined subsets of the data to be cryptographically safeguarded atomically. We use them for several reasons. Each group has a public key pair associated with it. Their use is elucidated in Section 4.4.2.

The safeguards are instituted in response to threats. Each threat has associated with it a set of files that may be affected by it. The grouping is orthogonal to any operating system attributes. If an intrusion detector determines the need to take precautions against a threat, one course of action is to safeguard the relevant files. A protection group serves as the data structure used to track a subset of files that are always affected together, regardless of what the current threat may be. Hence, a single threat may affect a number of protection groups.

Since protection groups are defined independent of any other file attributes, they can be defined as arbitrary sets. This ensures that files that are unlikely to be affected by a threat are not safeguarded. This property

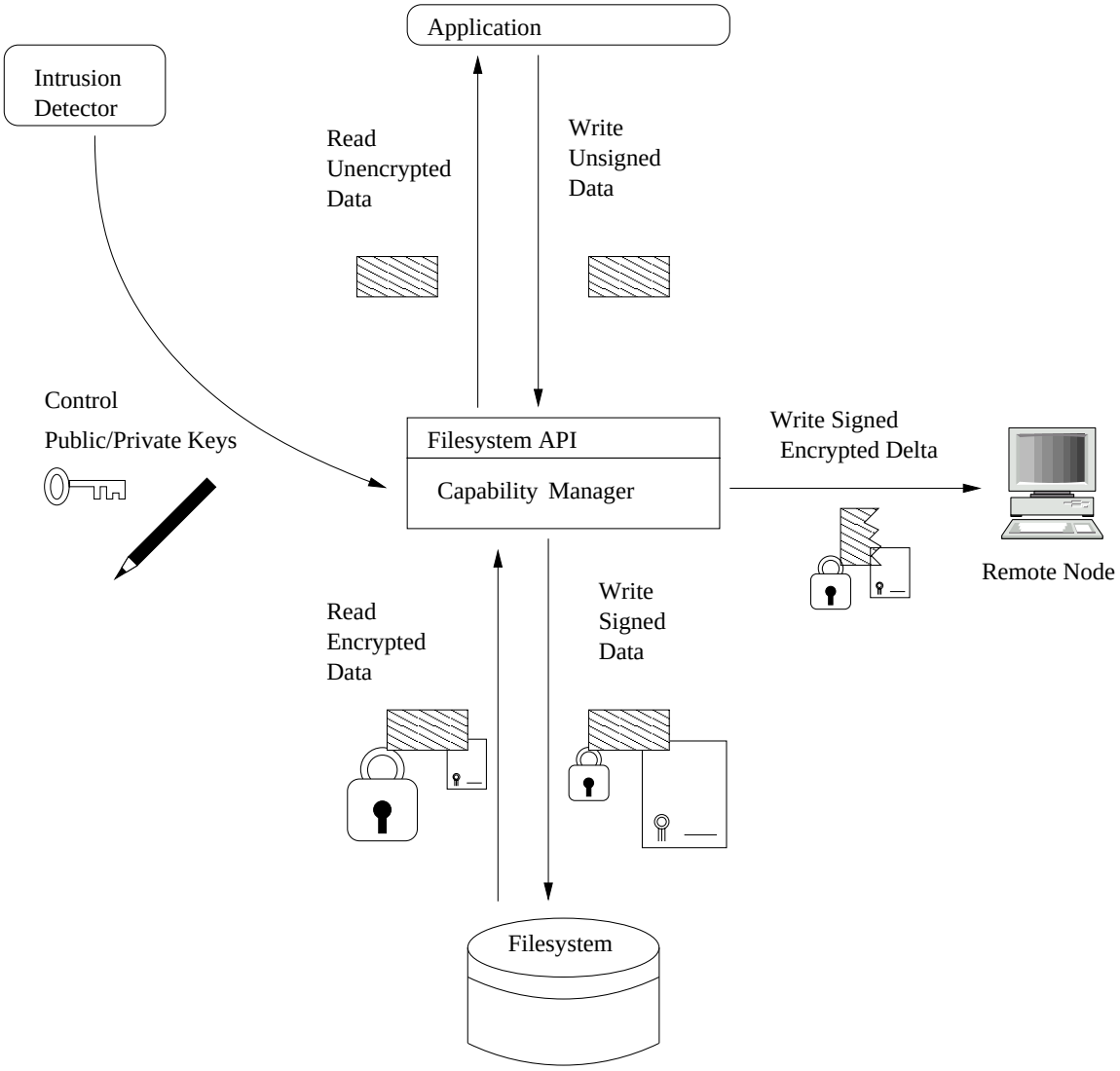

Figure 4.1: RICE allows an intrusion detector to rapidly provide data protection when faced with a likely attack. Cryptographic encapsulation is reduced to simple key deletion so it can be effected rapidly.

ensures that subsystems and applications that do not utilize data that is threatened can continue operating normally.

When a threat appears and a set of files must be protected, it is imperative that the safeguards be instituted in as short a time period as possible. The longer it takes, the more damage can be effected in the interim. The use of protection groups allows the system to perform a small number of key deletion operations on the groups' meta-data, rather than a large number of operations on all the constituent files.

4.4.2 Assurance

When the system is under threat of penetration, data must be safeguarded. The aim is to guarantee three properties for the data - confidentiality, integrity and availability - that will hold even after a successful attack.

Confidentiality

Each file that is part of a protection group is kept encrypted in a symmetric cipher with its own unique key, which serves as a *cryptographic capability*. This capability in turn is kept encrypted with the group's public key. If an attacker is likely to compromise the system in a manner that threatens the protection group of the file, the private key of the group will be deleted. This will prevent the file's cryptographic capability from being decrypted. Without the file's capability accessible, the file's confidentiality is guaranteed.

When an application seeks to use the file and its protection group has not been threatened, then the runtime environment is able to transparently enable access to the file by retrieving its cryptographic capability

(which can be unsealed with the extant protection group private key), decrypting the file and then opening the temporary decrypted version. When the application is done with the file, it is re-encrypted and the temporary version deleted from the system.

If a file was still in use at the time of the penetration, along with the deletion of the group's private key, the temporary decrypted version of the file will be deleted. Any changes made since it was last opened would be lost, but its confidentiality would be maintained.

Integrity

In order to be able to verify the integrity of a file, a cryptographic hash of the contents of the file is maintained with the file's meta-data. To prevent the hash from being manipulated without authorization, it is always sealed with the file's protection group's public key before it is stored. If the file is changed after being opened, then the hash of the new version must be computed, sealed with the protection group's public key, and stored in the file's meta-data. When a file is opened, either for reading or writing, the file's hash is computed and compared to the one stored in the meta-data (after unsealing the stored hash using the protection group's private key). If the hashes match, the file's integrity is deemed to have been verified.

If an intrusion detector determines that a protection group is threatened, it deletes the group's public key. Once this has been done, any changes that an attacker makes to a file will be detectable. Since the protection group's public key is no longer present, it is not possible to seal the hash of the changed version of the file. When the file is accessed subsequently, the fact that the computed hash does not match the stored hash

(after it has been unsealed with the protection group's private key), signals that the file's integrity has been compromised.

If a file was in use when an intrusion occurs, the integrity of any changes that were made since the file was opened will not be recorded. This is due to the fact that the decrypted version of the file will be deleted without re-encrypting it (since that would introduce an unacceptable delay which an attacker may be able to exploit), and hence since the changes will be lost there is no question of verifying their integrity. If the attacker does not alter the file, it will remain in the state that it was before the last time it was used and its integrity can be verified. If the attacker alters it, the integrity check will fail.

Availability

The goal of guaranteeing the availability of data in the face of an attack is usually managed by instituting a regular backup regimen. When a system penetration is detected, data from a backup prior to the intrusion is extracted and used to replace the tainted version. This is an inherently synchronous process, bringing with it a necessary tradeoff. Increasing the frequency of the backup decreases temporal extent of data loss. However, it also imposes an increased overhead. These two factors must be balanced. In addition, either all the files are backed up or the entire filesystem must be inspected to search for files that have changed. This fact places a lower bound on the time to effect a single backup. The bound grows with the size of the filesystem, a quantity that continues to increase with time.

We address the issue through the use of an asynchronous approach.

We incorporate functionality in the runtime environment which copies changes made in files to a remote node. If an attack is subsequently deemed to have occurred, the state of any file that has been changed can be computed using the sequence of changes that have been copied over.

When a file is accessed, a copy of the original version is maintained. After the file is closed, the runtime determines if the file has been written to. If it has, a delta is computed between the original version and the new version. A hash of the delta is computed and sealed with the file's protection group's public key. The delta itself is encrypted with the file's cryptographic capability. Thus, the modifications are provided the same confidentiality and integrity guarantees as the original file. The sealed hash and the delta are placed in a temporary location on the disk. A separate process synchronizes the deltas with a remote node.

4.4.3 Virtual Layer

In order to implement the changes needed to provide the data security guarantees in a manner that is transparent to extant applications, it was necessary to introduce them in the operating system itself. There are two possible approaches. The first option is to modify the filesystem itself, altering its data structures to include the new meta-data needed, along with the cryptographic transformations that use the auxiliary protection information. The alternative approach is to introduce the functionality as a *virtual layer* over an existing filesystem. When runtime environment calls are made to operate on files, they can be intercepted and the new transformations effected if required, making calls to the native filesystem as needed.

With the latter approach, there is a further choice of where to store the security related meta-data. One option is store both the meta-data and the actual content in the native filesystem version of the file. The other option is to maintain the meta-data separately. We opted to use the virtual layer approach with the meta-data stored separately. Described below are some of the factors that were involved in making the choice.

Using either a different native filesystem format or a virtual layer with the meta-data stored with the data within a single file in the native filesystem has several limitations. It will not be *backward-compatible* with any extant data stored in a currently deployed filesystem due to differing formats. All that data will have to be copied over. The functionality provided by any *attributes* stored in the meta-data of the old filesystem will either be lost or have to be re-implemented. The new filesystem will not be interoperable with any other runtime system that does not have support for the new file format. Additionally, the resulting system will not be *extensible* - that is if new attributes are to be added to the meta-data of each file, they can not be inserted for each file without rewriting the entire filesystem.

Maintaining the meta-data separately brings with it the advantage of being able to add new fields for existent files with little cost. For example, to add functionality to retrieve a file's cryptographic capability dynamically from a remote capability server if it is not present, new fields would be needed to store the capability server's location. The cost to introduce the field into the meta-data stored separately would be proportional to writing out all the meta-data, and would not incur the cost of having to write out all the data stored in the filesystem as well.

Using a virtual layer approach with the meta-data stored separately

from the files has the disadvantage that the native filesystem's synchronization of the meta-data can not be leveraged. However, this is addressed by limiting the use of shared data structures that must be locked - they are used only when a file is opened and closed, not when it is read or written. Therefore the overhead introduced is minimal.

The approach of storing the meta-data with the data has the advantage of allowing files to be transported from one filesystem to another, even across different hosts, and yet retain their protection profile so that they may potentially be accessed independently of the resource in which they reside. Since we are focused on a single host operating environment, this did not provide a significant advantage.

4.4.4 Protection Granularity

Another choice that must be made is the granularity at which cryptographic operations are to be performed. Cryptographic file system projects, such as those described in Section 4.7, either encrypt or decrypt an entire file or just a block at a time. Operating at file granularity results in a performance impact when opening and closing a file, while operating at block granularity introduces overhead for read and write operations. Since reading and writing are far more common operations in a typical workload, we opt to use file granularity. Below we further describe the tradeoff involved in the choice.

Performing cryptographic operations at file granularity results in the fact that opening and closing a file, which is an $O(1)$ operation in a traditional filesystem, becomes an $O(n)$ operation, where n is the length of the file. This is because the entire file must be decrypted and its integrity

verified when opening the file. Similarly, the file must be encrypted and its hash computed when closing the file. If blocks of size b are used and cryptographic operations are performed at block granularity, then open and close operations have $O(b) = O(1)$ cost. One method to address this issue is to fix an upper limit on the size of file that may be protected, say k. The complexity of opening and closing a file is then $O(k) = O(1)$ if the k is a constant.

The advantage of performing operations at file granularity manifests when files are being read and written. Operating at block granularity introduces the latency of decryption and encryption during reads and writes. If operations are performed at file granularity, an unencrypted version is used during read and write operations so there is no cryptographic overhead. When the workload used involves concurrent accesses of files by multiple processes or there exists significant locality of reference, then the fact that reads and writes have no extra cost in this approach results in a performance advantage over the block granularity approach. If a significant portion of the file is used, then operating at file granularity approximates the use of an optimal pre-caching policy that has perfect lookahead, coupled with an infinite size cache.

If the following conditions are *all* true for the files in the workload, then using block granularity would have been preferable - a very small fraction of each file is used (since operating on the entire file would add significant overhead), the file is not reused (since block granularity reuse is much more expensive as cryptographic operations must be effected on each use), the file is not used by concurrent processes (since there is no extra cryptographic cost added for all processes after the first that use the

file).

4.5 Implementation

Below we describe the organization of the meta-data used, the tool *Group Manager* used to manipulate it manually, and the runtime subsystem *Capability Manager* that transparently manages it for applications.

4.5.1 Meta-data

Each file that is protected by RICE has several attributes that are stored in an instance of the **ObjectMetaData** data structure. These include:

objectLocation The location of the file in the filesystem at the time of protection.

objectGroup The protection group to which the file belongs.

instances The number of concurrently open instances of that currently exist.

decrypted The location of a temporarily unencrypted version (if one exists) of the file.

pristine The location of a temporary copy of the file in the state that it was when the file was opened, before any writes occurred. It only exists if a file is currently open and serves as a baseline against which deltas of the file can be computed.

sealedCapability The cryptographic capability (symmetric key) used to encrypt the file, wrapped in the public key of the protection group of which it is a member.

capability The value of the unsealed cryptographic capability, which is only present while the file is open.

currentCheckpoint A counter used to indicate the position in the sequence of deltas that are computed each time a file is closed after changes have been made.

computeDelta The value serves as the equivalent of a dirty bit on a page. It indicates whether any instance of the file was opened for writing, in which case a delta must be computed when it is closed.

sealedHash The cryptographic hash of the file as it was when it was last closed, kept sealed in the protection group's public key.

idempotency Each instance of an open file has a unique hash associated with it that is stored in this set.

Each protection group is stored in an instance of the **ObjectGroup** data structure. Each instance contains the group's name, its public key (used to seal the cryptographic capabilities and hashes of files in the group) and its private key (used for unsealing those capabilities and hashes). In addition it contains a hashtable of pointers to the meta-data of the group's members, which is indexed by the full path of the member's location in the filesystem.

Finally, the **Resources** data structure contains two hashtables. The first is indexed by the names of protection groups, associating the group name with a pointer to the group's meta-data, from which a list of all member files may be extracted. This is used when a group is to be protected,

since each member's decrypted and pristine copies must be erased if confidentiality is to be guaranteed. The second hashtable is indexed by the full path of a file. Upon being queried about a file, it returns the meta-data of the group to which the file belongs.

4.5.2 Group Manager

The **GroupManager** is a tool for the administrator to manually manage the protection status of files. It performs all operations on a *groups database* which stores all the meta-data associated with all the files of all protection groups. To make changes to this database, a password is required. By maintaining the meta-data of the virtual layer in this manner, it is possible to have multiple groups databases and switch between them to institute a different protection policy. We describe below the operations that may be performed using the **GroupManager**.

All operations require a password since they all read or write the groups database. The password is used to create a symmetric key which is used for decryption of the groups database when it is being read and encryption when it is being written. The operations must also specify the type of operation by passing a *mode* parameter to the **GroupManager**, and the file in which the groups database is stored.

Capabilities File

Since the runtime system requires transparent access to the meta-data, the **GroupManager** can be use to generate a *capabilities file* which is not password protected using an *output* operation. During the course of execution, this capabilities file will be manipulated by the runtime since it

needs to update the cryptographic hashes (used for integrity checks) of files that have been written to. To allow the groups database to reflect these changes, the content of a capabilities file can be transferred to a groups database using the *input* operation.

Group Listing

For convenience, the groups database can be interrogated with the *list* operation. If a specific group name is passed as a parameter, then the files which are a member of the group (if any) are listed. Alternatively, if no parameter is passed, then the list of currently defined protection groups is generated and emitted.

Altering Membership

Finally, a file may be added to a protection group with the *add* operation by specifying its current location in the filesystem. If the file has previously been added, the request will not alter the state of the meta-data. Files may not be added to more than one protection group. Files that would be members of the intersection of protection groups should be combined into a new, separate protection group of their own.

If the protection group does not exist, it is dynamically created, including a pair of public and private keys for sealing and unsealing its members' capabilities and hashes. A hash of the plain file is computed before encryption and is sealed with the group's public key. A new cryptographic capability (symmetric key) is generated for each file that is added. The file is encrypted with this key, after which the key is sealed with the group's public key.

The *remove* operation can be used to remove a file from a protection group of which it is currently a member. The file is decrypted using its cryptographic capability retrieved by unsealing it with the group's private key. Similarly, the file's integrity is verified by computing its hash and comparing it to the one stored in the meta-data (after unsealing the hash with the group's private key). All associated meta-data is then deleted. If the file was the only member of the protection group, then the group and its associated meta-data are also deleted.

4.5.3 Capability Manager

Platform

We implemented the **CapabilityManager** as a modification of Sun's Java Runtime Environment. The underlying implementation of all classes that provide an interface to files is through the use of the **java.io.FileInputStream** and **java.io.FileOutputStream** classes. Our implementation hence instruments these two classes' constructors and *close()* methods. (Version 1.4 of the Java Runtime Environment introduced a new subsystem for non-blocking input and output, which accesses the filesystem through native virtual machine calls. RICE does not support file manipulation with the **java.nio** subsystem.)

In principle, however, the design of the **CapabilityManager** supports the augmentation of multiple classes, not just the **java.io.FileInputStream** and **java.io.FileOutputStream** classes. This is because the only state that is stored in the class which invokes the **CapabilityManager** is the name of the file used in the constructor so that it can be passed back to the **CapabilityManager** after a file is closed to allow the file to be re-protected.

Hence, adding support to new classes only requires the addition of a single field to each and the instrumentation of the constructors and *close()* methods.

Initialization and Committal

The **CapabilityManager** takes two parameters. The first is the capabilities file referred to in Section 4.5.2. All meta-data for the virtual layer is stored and manipulated in this file. The second parameter is a location on disk where deltas are stored temporarily after they are computed for files that are modified by writes. They are transferred from this location to a remote node by an independent process.

When the runtime environment starts, the first time either a file read or write operation occurs, an attempt is made to load the **CapabilityManager**. If either required parameter is not provided or there is an error, the system will run without the **CapabilityManager** and files that are members of protection groups will only be accessible in the encrypted form. During initialization, the virtual layer is populated with meta-data read in from the capabilities file.

While the system is operating, if at any point all the files opened by applications are closed, the meta-data from the virtual layer is committed to the capabilities file. This choice allows the meta-data to be committed in a coherent state and assures that it is written out before the runtime shuts down.

Opening a File

When an application constructs a class that provides access to the filesystem, a call is made to the virtual machine's native *open()* method. We introduce code in the constructors to pass the filename as a parameter to the **CapabilityManager**'s *unsealFile()* method. The **CapabilityManager** inspects **Resources**' hashtable of all protected objects and determines if the file in question is being managed by RICE. If it is not, it simply returns the same filename. The virtual machine's native *open()* method is invoked with the filename as it would in the absence of the **CapabilityManager**.

If the **CapabilityManager** determines that the file is being managed by RICE, it looks up the **ObjectMetaData** for the file. With this it is able to check whether this file has been previously opened either by any executing thread (including the current one). If it has not been opened, then a check is done to see if the file's protection group's private key is available. If it is, then it is used to decrypt the file's cryptographic capability and sealed hash. The capability is used to decrypt the actual file, whose hash is computed and compared to the unsealed hash. If the hashes do not match the integrity check is deemed to have failed and is flagged. In addition a pristine copy of the file is made. The decrypted file is stored in a temporary location and it is this location that is returned by the **CapabilityManager**. If the **CapabilityManager** found that the file had been opened, then a decrypted file's location would already be present in the **ObjectMetaData** and this would be returned. In either case, the returned value is used as the parameter when calling the virtual machine's native *open()* method.

Since the **CapabilityManager** itself uses the filesystem, we introduce

a new constructor with an extra parameter. The parameter is used to determine whether the **CapabilityManager** will be used when opening the file. The standard constructor also calls the new constructor, passing it a value that indicates the **CapabilityManager** should be used. This is transparent to applications (unless they use reflection and depend on the fields stored in the class).

Closing a File

When an application finishes using a file, it invokes the *close()* method of the class with which it gained access to the file. This may be **java.io.-FileInputStream**, **java.io.FileOutputStream** or one of the classes which in turn use these classes, such as **java.io.FileReader** or **java.io.FileWriter**, to access files. We modify the *close()* method, allowing the normal operation to complete and then introduce a call to the **CapabilityManager**'s *sealFile()* method. Two parameters are passed, which are the filename and the *computeDelta* value which signifies whether the file was opened for reading or writing. The **CapabilityManager** inspects the relevant **Resources** hashtable to check if the file was protected by RICE. If not, it returns silently.

A count is maintained in each file's **ObjectMetaData** to keep track of how many instances of a file have been opened. Each time a file is opened, the count is increased and each time a file is closed, it is decreased. If this count reaches zero, no application is currently using the file. When this occurs, the **CapabilityManager** checks a flag to see if any instance of the file had been opened for writing. If not, then the decrypted version and pristine copy of the file are both deleted.

If the **CapabilityManager** found that the file had been opened for writing, it needs to commit the changes. It must first check to see if the file's protection group's public key exists. It then computes the delta of the file as the difference between the pristine copy and the current state of the unencrypted version. It computes the hash of both the file as well as the delta and seals each hash with the group's public key. The sealed hash of the file is stored in the file's **ObjectMetaData**, while the delta and its hash are written out to a location calculated as a function of the filename, it's *currentCheckpoint* and the parameter passed to the **CapabilityManager** at initialization. The *currentCheckpoint* is then incremented.

Each concurrent instance of a file that is opened is associated with a unique token which is stored in the *idempotency* set. When a file is closed, a check is performed to see if the token passed in as a parameter is in the idempotency set. If it is not, then this instance of the file was previously closed and the call is ignored, making *close()* an idempotent operation as required by conventional semantics. In addition, unauthorized sealing of the file is prevented since the *sealFile()* operation requires that the same token be passed as a parameter as the one that was passed to the corresponding *unsealFile()* operation that was invoked when opening the file. This assumes that the choice of the token is cryptographically random.

4.5.4 Runtime Protection

When the system is running, if an intrusion response engine determines that a group is under threat, it can opt to use RICE to cryptographically remove either write access or both read and write access.

To remove write access, it need only delete the protection group's public

key. Once this is done, files can still be written on the local filesystem, but the hashes of the new files and the deltas computed can not be sealed with the public key. When a system is investigated after a penetration, the changes that have not been signed can be deleted, restoring the last signed versions. In this manner, the filesystem can be restored to a state where all unauthorized writes are left out.

To remove read access, the response component only needs to invoke the *disable()* method and delete the private key used to unseal cryptographic capabilities. The *disable()* method iterates through the protection group's member's meta-data, deleting any unencrypted and pristine files that are defined. Once this is done, if a penetration occurs, there is no means (short of brute force key search) to gain access to the protected files (modulo covert channels such as the magnetic remanence of data).

To re-enable access to a group, the response component can call the *enable()* method. In this case, the group's name is added to a set. When an attempt is made to access any of the files in the set, the system will attempt to authenticate the user. If it succeeds all groups in the set will be re-enabled. The use of protection groups coupled with the process of combining multiple protection groups' re-authentication the minimizes impact on usability.

4.6 Evaluation

In the previous sections we have described the benefits of augmenting the runtime with RICE. However, the use of cryptography implemented in software introduces a computational overhead that slows down file operations.

To estimate the extent to which RICE affects performance we describe two sets of experiments.

Since the use of RICE only introduces an impact when a file is being opened or closed, the first experiments consists of micro-benchmarks that measure the cost it adds to *open()* and *close()* operations. The cryptographic overhead is a function of the size of the file that is being opened or closed. Hence, in the first experiment we vary the file's size and measure the time to open the file. This cost is independent of whether the file was opened for reading, writing or appending. The second, third and fourth experiments measure the cost to close a file, as a function of its size, after it has been opened for reading, writing or appending. The cost to close a file after reading is minimal since no encryption, hash or delta computation is needed. In the case that the file is opened for writing, the file is created with zero length and then filled so it reaches the expected size. The file opened for appending is already one which is the expected size and no extra data is written to it. Thus in both the write and append cases, the file that must be encrypted is the same size, but the append case requires less computation for constructing the delta which results in the operation being less expensive. The results of these experiments are displayed in Figure 4.2.

The micro-benchmarks show that the impact of RICE on opening and closing a file is significant. However, these operations constitute only a fraction of the cost of a typical workload. Therefore, we ran the SPECjvm98 [SPECjvm98] suite of applications to obtain a macro-benchmark which would provide an estimate of RICE's impact in context.

The SPECjvm98 suite includes a separate file for each application which

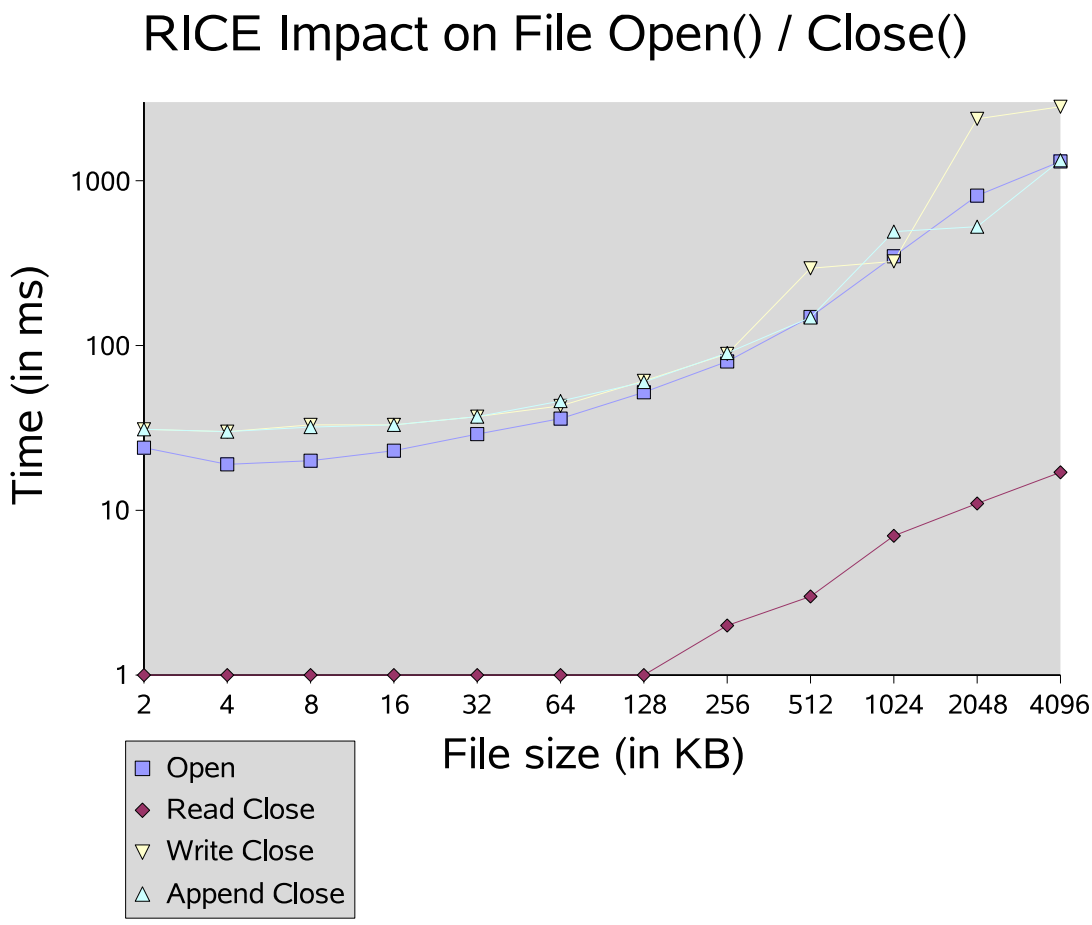

Figure 4.2: Cost of opening and closing a RICE protected file, measured as a function of file size. The cost of closing depends on whether the file was opened for reading, writing or appending.

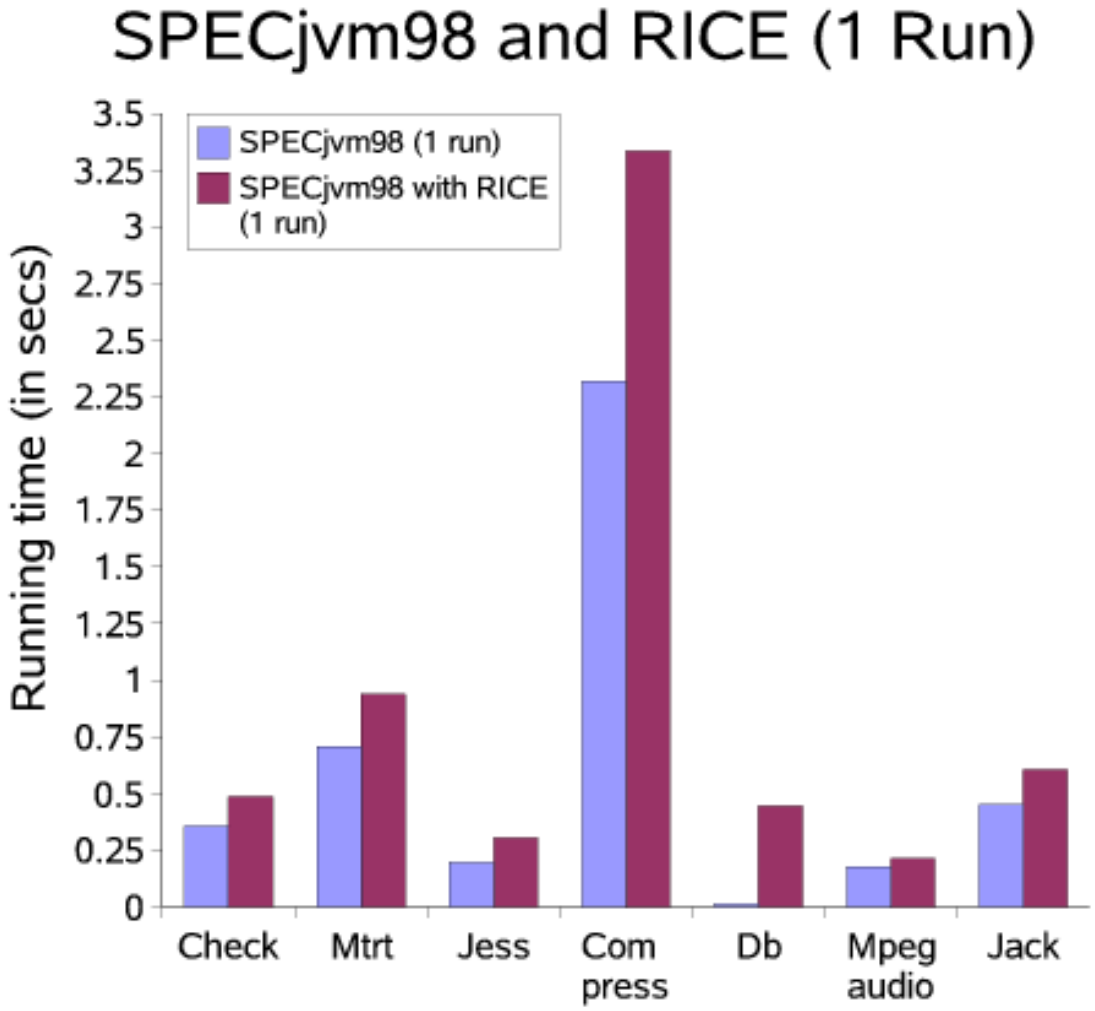

Figure 4.3: RICE imposes a noticeable impact on applications in SPECjvm98 that rely heavily on the filesystem if limited to a single run.

contains output results that are used to check that the program ran correctly. RICE protection is added to these files. In addition, all but one of the programs use one or more datasets that are stored in files. These files are protected with RICE as well. The files vary in size from 55 bytes to 3.5 megabytes. The result of a single run is shown in Figure 4.3.

compress is a Lempel-Ziv compressor. It is the worst affected in terms of absolute cost since it accesses the largest file in the workload. *db* per-

forms a series of add, delete, find and sort operations on a memory resident database. It is worst affected in percentage terms, since it uses multiple small files, with the result that the key management overhead is pronounced.

check exercises the virtual machine's core functionality such as subclassing, array creation, branching, bit operations, arithmetic operations. All the overhead introduced by RICE is from the cost of opening the file against which the output is matched for correctness.

jess is an expert system that solves puzzles using rules and a list of facts. *jack* is a lexical parser. *mtrt* is a ray tracer. *mpegaudio* is an MP3 decompressor. In each case, most of the overhead is from the capability management. In the case of *mtrt* the overhead is greater since it uses a significantly larger data set.

Since RICE is designed to take advantage of concurrent and repeated use of files, we undertook two more experiments where the applications are allowed to repeat a number of times. This allows us to see the benefit of RICE when the workload involves repeated access to the same files, either from a single process or multiple concurrent processes. The results of the experiment with 10 runs is shown in Figure 4.4. The cases where RICE's overhead was most pronounced, such as *db* show a marked improvement. Cases like *mpegaudio* are still dominated by key management since the data is streamed once and there is little re-use. Finally, the results of an experiment with 100 runs is shown in Figure 4.5. The impact of RICE is no longer significant in these results.

Thus, if the workload has enough reuse of the files, RICE is viable as is. RICE uses Java implementations of cryptographic subroutines. Native

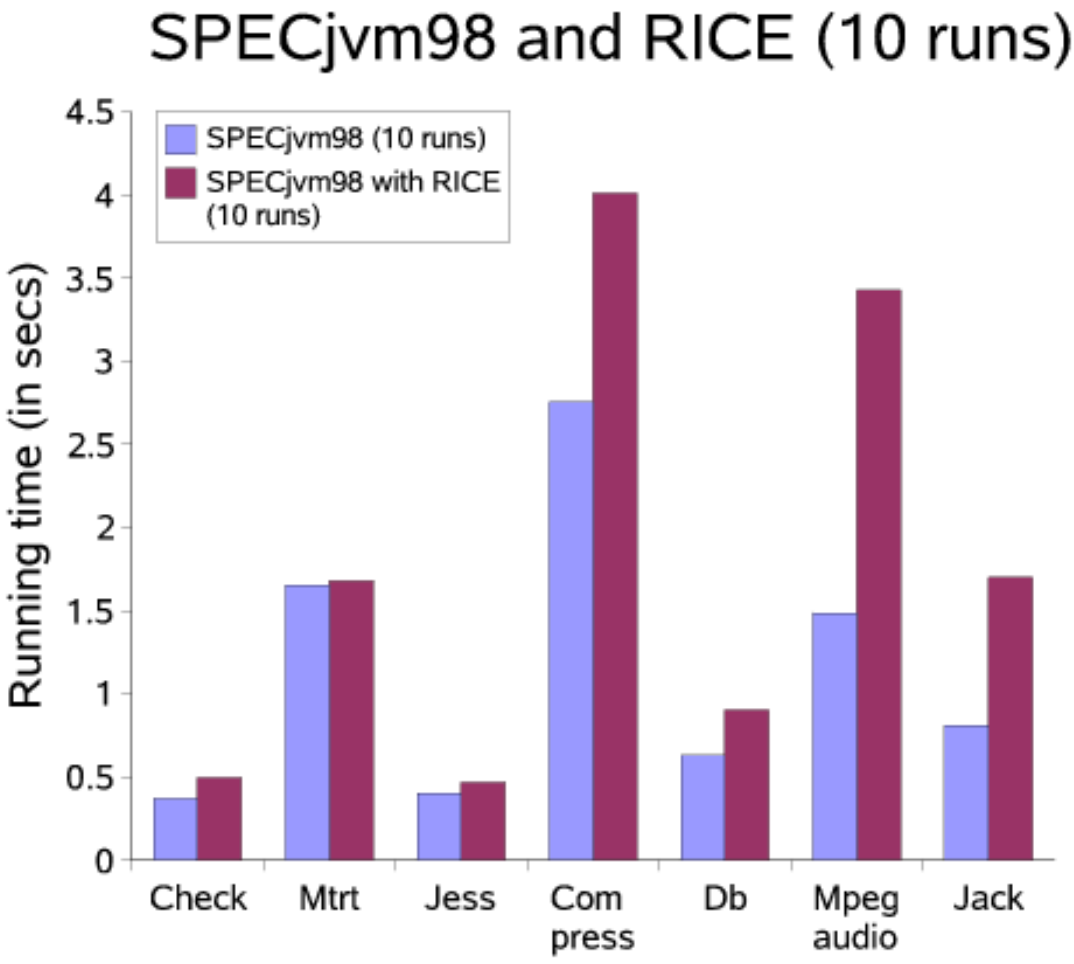

Figure 4.4: RICE's overhead is noticeably diminished when SPECjvm98 runs 10 times, due to the benefits of caching.

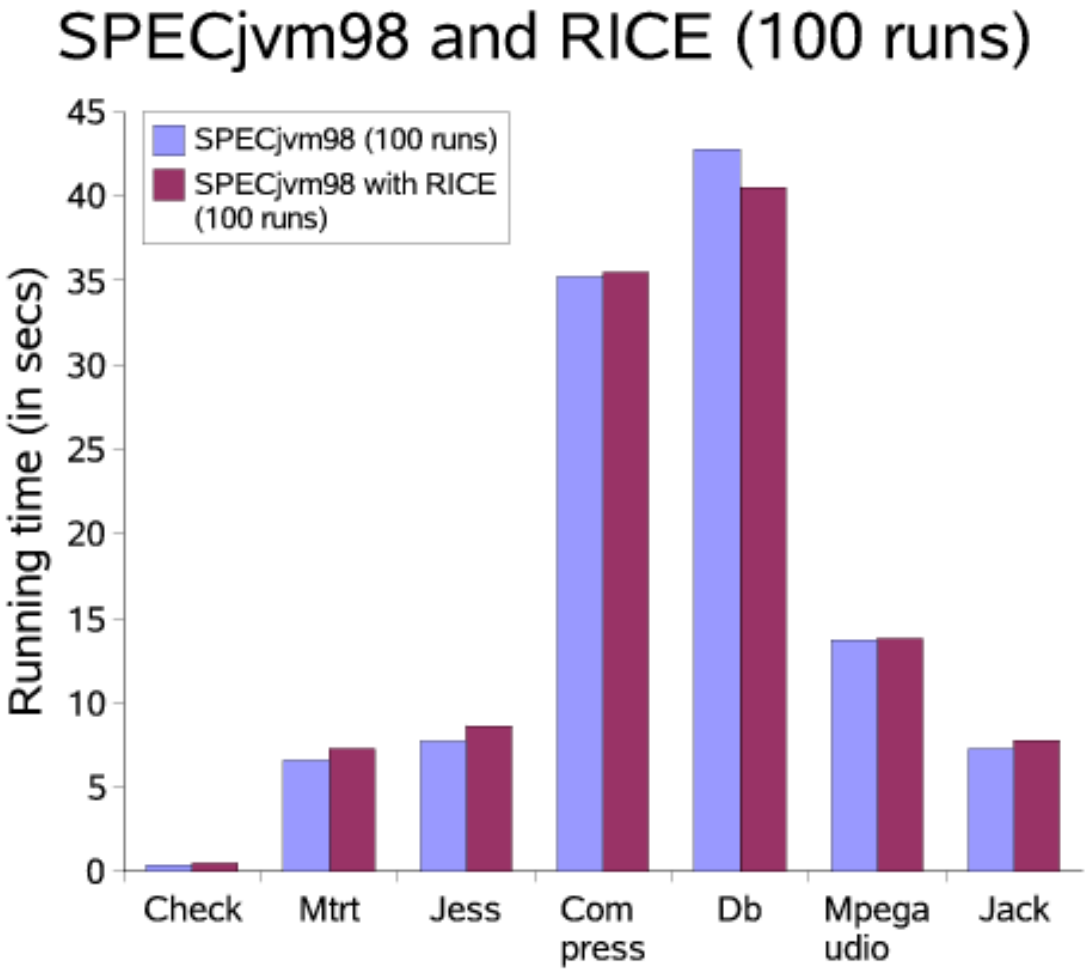

Figure 4.5: When SPECjvm98 runs a 100 times, the caching benefits compensate for the initial and final cryptographic operations to the point where the impact of using RICE is no longer significant.

ones will offer a significant performance improvement. The experiments were performed on a 700 MHz processor, while current generation ones run at speeds over 3 GHz. Since the bottleneck that increased the running time of the applications was the CPU-intensive cryptographic operations, it is likely to reduce significantly with the use of newer, faster CPUs. In addition, commodity processors will include dedicated hardware cryptographic acceleration in the near future. This which will address the issue.

4.7 Related Work

Several projects have used cryptography to control data access at file granularity. Each has a difference from what we propose which makes it unsuitable for application in the context we describe.

Cryptographic File System [Blaze93], Transparent Cryptographic File System [Cattaneo01], and Cryptfs [Zadok98] use only symmetric key cryptography. Guaranteeing a file's integrity requires a means to check that it has only been modified by an authorized party. If the key used to verify the hash of the file was of symmetric cipher then it could be used to modify the hash as well. An asymmetric cipher is required to allow verification of integrity without allowing changes. Framed in terms of file operations, we need a asymmetric cipher to be able to grant read access without write access.

Secure File System [Mazieres99] and Secure File System - Read Only [Fu00] use asymmetric ciphers to provide authentication but not confidentiality. Encrypting File System [MS99] uses an asymmetric cipher for authenticated access along with a symmetric cipher for confidentiality. It

does not sign hashes on writes which are verified on reads, so it is unable to guarantee integrity.

Secure File System [Hughes00] and Cepheus [Fu99] target distributed environments and rely on the network for gaining access to keys. Apart from the latency introduced by the network access, this introduces the weakness of allowing an attacker to cut off access to files by flooding the port used.

Finally, none of these systems aim to address the issue of availability of data after a successful attack. RICE addresses this by computing the changes made to files and storing them on a different host to allow file reconstruction should the original no longer be available after an intrusion. Additionally, RICE provides an interface for key manipulation to allow cryptographic guarantees to be added to read/write access denials with minimal overhead.

4.8 Summary

RICE provides a means to augment the runtime to provide data security guarantees. Using RICE, precautionary measures can be added to files so that in the event of an attack, the confidentiality, integrity and availablility of data can be maintained. By mapping read and write capabilities to their cryptographic analogues of confidentiality and integrity, and organizing key management appropriately, RICE allows access to the data to be removed rapidly by key deletion. File modifications result in deltas that are replicated to a safe node, thereby guaranteeing availability even after a penetration occurs.

Chapter 5:
System Snapshot Service

5.1 Overview

A forensic analyst seeks to reconstruct an incident based on the evidence available after the event occurs. If a site is considered to be a likely target, precautionary measures such as the installation of surveillance equipment is warranted. After an attack occurs, the recording made by the aforementioned device can be used to aid in the reconstruction of events. It can also guide the forensic examiner's choice of what further evidence to search for. Of particular utility is the fact that data of a forensic analyst's choosing can be gathered (by installing the relevant tools prior to exposing the site to attackers) rather than relying solely on the evidence unintentionally left by an intruder. An analogous approach is applicable to computer forensics as well. NOSCAM is a tool developed to provide a framework to facilitate the pro-active gathering of evidence and its subsequent examination.

5.2 Background

Despite the use of preventive measures, the number of security incidents reported [Howard97] is rising. As society becomes increasingly dependent on the computing infrastructure connected to the Internet, and technical measures are unable to resolve all security problems, it is likely to turn to the judiciary. Computer forensics is thus likely to play a greater role in dealing with security breaches. The software running, the types of at-

tacks, and the attack patterns are all becoming more complex. Relying on incidental evidence alone to reconstruct an attack is therefore increasingly harder.

One approach to alleviate this is to pro-actively invoke surveillance tools on a computer system to create activity records should a forensic analyst need them. This must be tempered due to performance and data storage considerations as well as privacy concerns. Our work is informed by these precepts.

5.3 Intermediate Certainty

The need to construct a tool that could be used by a forensic analyst to shed more light on the circumstances of an intrusion arose from the fact that not all attacks will be handled by the technical means developed to effect automated intrusion response.

Current intrusion detection technology does not yield a low enough false positive rate to allow unchecked active responses. Since there is a significant possibility of misidentifying activity as aggressive when it is not, taking offensive action against the hypothetical attacker can yield new unwarranted attacks rather than stopping current ones.

Instead, we identified as a target of opportunity the case when an intrusion detector has identified an attack with an intermediate level of certainty. If all such activity were flagged as intrusive, the detector would cumulatively trigger false alarms at an unacceptably high rate. Hence, the detector will opt to allow this activity to pass unchecked. However, we can opt to record the details of the circumstances in such a situation for they

may be useful if it subsequently determined that an attack had indeed taken place.

This goal gave rise to the tool NOSCAM that we now describe. Its scope is the automated gathering of runtime environmental data such that human analysis may be performed to determine the source and means of the attack. This will be needed to understand why the intrusion detector failed to recognize the attack with a high enough degree of certainty. It will also enable a legal response by gathering data useful to the forensic analyst.

5.4 Forensic Goals

Forensic analysis of computers has focused on drawing inferences based on the remnants of a system after an intrusion has occurred. For example, the Coroner's Toolkit [Farmer00] relies heavily on undeleted information in the Inodes of a Unix filesystem. Intruders, however, are likely to attempt to either hide or erase the trail they leave. This is seen, for instance, by the recent publication of tools [Anon02] to encrypt or delete Inode information to counter the Coroner's Toolkit. We seek to address this specific concern.

Our goal is the automated preservation of evidence before an intrusion occurs. Additionally, we wish to store it in a manner that is not susceptible to deletion by an attacker nor dependent on network connectivity for the correct operation of this functionality. This is since the intruder may attack the network functionality with ease. Further, we wish to obtain audit information from several points in time, so that the analyst is not forced to reason based only on the final state of the system. Finally, we

wish to allow the analyst to prescribe what data is to be audited rather than being forced to rely only on the logs of applications and the operating system meta-data that survives an attack.

In addition to preserving evidence, a sound forensic examination documents each step, is repeatable, and can be verified by an independent agent. Our design lends itself to achieving these goals.

5.5 Design

We now provide an overview of the design of the NOSCAM tool. There are three components. *noscam_db* is used to create and then manage the database in which the audit trail is stored. *noscam_audit* is responsible for gathering data. *noscam_run* is to be used to retrieve information.

5.5.1 Data Management

noscam_db is used to create the database in which the audit trail is stored. It creates a new database on the server with a table containing all the requisite fields. It is further used during initialization to commit a baseline version of the audit trail. It is subsequently used to handle the periodic commitment of the new entries to an immutable medium, typically a CD-R (recordable compact disc).

At a later point in time, if an intrusion is deemed to have occurred, *noscam_db* can be used to import the records from immutable media into an instance of NOSCAM on an uncompromised system. They can then be used to reconstitute an approximation of the runtime prior to intrusion. *noscam_db* can convert data gathered from multiple executions of

noscam_audit into a coherent database suitable for use with *noscam_run*.

The underlying database's functionality is leveraged to interleave the entries from multiple streams into an ordered sequence and to build an index to facilitate sorting and retrieving the data.

5.5.2 Audit Trail Generation

noscam_audit provides the framework with which runtime environmental data is sampled. The set of activity that it monitors is configurable. It allows the specification of which system utilities are to be invoked and what parameters are to be passed.

Each type of information is audited at an interval specific to its type. Information unlikely to change often can be audited less frequently. Other aspects are similarly configurable.

A second such factor is the priority level at which the utility should be invoked. This provides a simple interface by which auditing activity can be turned on an off.

The choice of whether the output should be stored on the immutable medium is a third aspect. This allows auditing data that is very voluminous not to be stored in the limited space of the immutable medium. (Instead a cryptographic hash could be stored to verify the integrity of such data.)

A fourth such aspect is the choice regarding whether the output should be stored, or whether a difference relative to a baseline initial invocation should be stored. In the case where the information generated is large in quantity but does not change much from invocation to invocation, saving only the differences is worth the extra computational overhead since it can

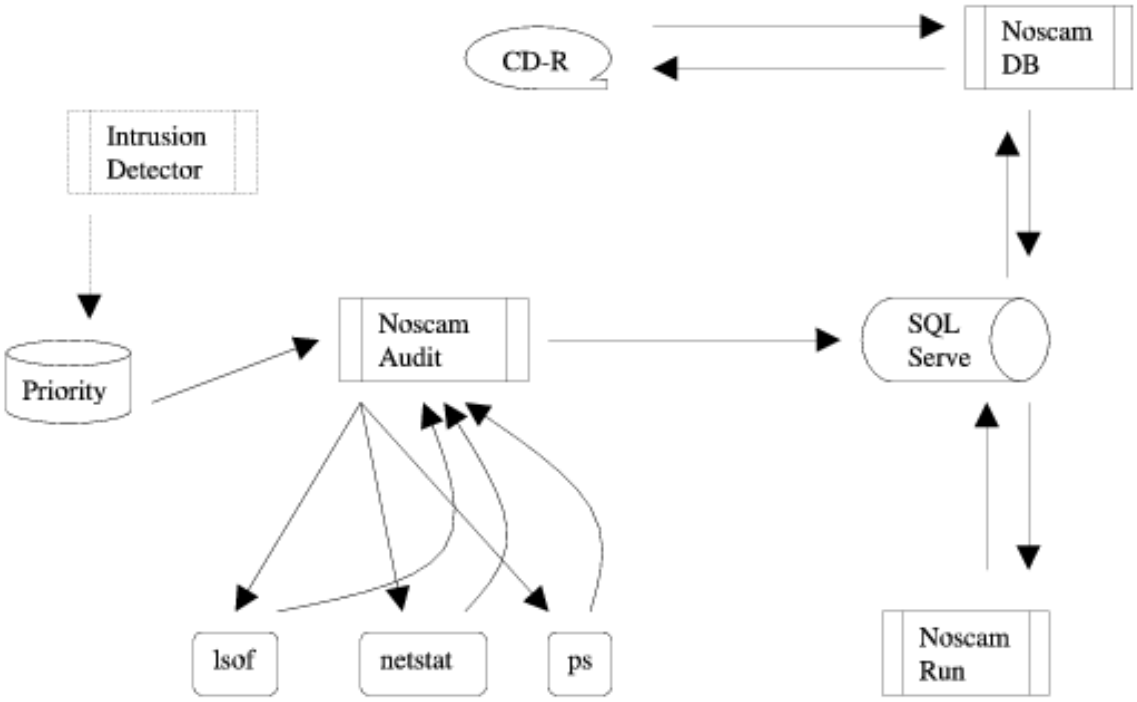

Figure 5.1: Architecture of NOSCAM, depicting the components *noscam_db*, *noscam_audit* and *noscam_run*. An intrusion detector specifies its estimate of the current threat level by writing a number in the priority file. NOSCAM Audit polls this file and starts and stops processes as needed. It captures their output and inserts it into a SQL database. Periodically, the database's content is backed up to a CD-R by NOSCAM DB. After an attack, it can reconstruct the database from the CD-R's. An analyst can use NOSCAM Run to query the database.

effect a significant saving of space on the immutable medium.

Finally, the output can be labeled with a category to facilitate retrieval during forensic analysis.

To minimize the need to replace the immutable medium, we expect that *noscam_audit* will be run as infrequently as possible. Instead, it is designed to be invoked when an intrusion detector has partially matched a rule or noticed moderately anomalous activity.

5.5.3 Forensic Querying

After a forensic analyst determines what specific information will be most pertinent to answering questions regarding an intrusion, they can use *noscam_run*. It is designed to provide an interface for efficiently extracting relevant output from the large collection of data gathered.

It affords the analyst the opportunity to run a query as if they were at the host at a point in time of their choosing before the system was successfully subverted. The analyst can do this by specifying the command and the time, and *noscam_run* will produce output corresponding to what would have been seen had they been logged in.

5.6 Implementation

We now describe some details regarding our implementation of NOSCAM.

5.6.1 Platform

The development is all currently being executed on a host that is running version 7.3 of Redhat's distribution of Linux as the operating system. MySQL serves as the backend database. Version 1.3 of the Java runtime environment from Sun is being used. Earlier versions of each software system will likely work since the primitives we use have had support in several prior versions.

All but one of the packages used in constructing NOSCAM are available for other Unix-like systems as well as Windows. *cdrecord* is a tool for writing to recordable compact disks that is limited to Unix-like environments. Since NOSCAM depends on it, this prevents porting to Windows.

Implicit in our discussion has been the requirement that the system on which NOSCAM is to be run must have a CD writer installed.

5.6.2 Intrusion Detector Interface

noscam_audit continuously monitors a 'priority' file. This file will always contain a single integer, which is the current priority level. An intrusion detector can communicate its current estimate of the likelihood of the system being under attack by editing this file and changing the value stored in it. When this is done, all commands in the configuration file that are designated below the current priority level will automatically be activated by *noscam_audit*, while all activities of a higher priority than the new value will be turned off. Auditing is halted by setting a low enough value in the 'priority' file. An alternate implementation exposes a program interface that accepts the priority level as a parameter.

5.6.3 Thread Management

When *noscam_audit* initializes, it reads the configuration file and spawns a thread for each variant of a command coupled with its parameters. Each thread consists of an inner loop that repeats at the specified interval and an outer loop which synchronizes on a lock that is shared with the main *noscam_audit* process. When a priority level changes, the main process iterates through its list of spawned threads, comparing their priority levels to the new level. If the new level is lower, then the thread is signaled to break the inner loop and the outer loop is forced to wait for the synchronization lock. (The lock will be released by the main process when it needs to restart the thread at a later point in time.) This allows *noscam_audit* to

use a constant size thread pool and avoid the overhead of spawning new threads.

5.6.4 Audit Trail Scope

The audit trail can extend to a range of runtime information. Examples are: the list of processes running; the creation, access and modification times for a pre-specified list of files; cryptographic hashes of these files; the routing table of the host; the list of open files and network connections; the disk usage; the firewall rules; the list of currently used kernel modules; traces of routes to hosts; port scans of hosts; and vulnerability scans of connected hosts. The system is extensible so that other audit data can be incorporated into the trail.

5.6.5 Query Utility Interface

NOSCAM aims to allow queries that mimic the execution of system utilities during a query-specific time frame. Consider an example where an intrusion detector's alarm is triggered at 12:10 pm. An analyst may wish to see what the output of 'netstat', a tool that prints network statistics, would have looked like at 12:05 pm, before the intrusion occurred. This would allow the analyst to see which remote hosts were connected to the compromised machine immediately before the attack.

noscam_run currently provides two functions. The first takes a date and two times as input. It produces a list of commands with options, the time they were executed and an entry number for each. The second function takes as input an entry number as produced by the previous function, and outputs the data written to the standard output and standard error

streams by the relevant command at the specified time.

5.6.6 Use of Immutable Media

noscam_db keeps track of the records that have already been committed to the immutable medium. Since the records are inserted with a monotonic entry number and there is a single instance of *noscam_db* with only one thread running at any time, this consists solely of keeping track of the last entry that was committed.

Periodically, it selects all the records that have their immutable field set and that have an entry number that is larger than the current maximum committed. These are then written to a file which is compressed. *cdrecord*, part of the Redhat distribution, is used to create a new session on the currently loaded CD-R in which this file is copied.

During recovery, all the relevant immutable media are mounted and have all their sessions' record files uncompressed. Each resulting file is then processed and its contents are inserted into the current live NOSCAM database, at which point it can be queried using *noscam_run*.

5.7 Evaluation

To evaluate the feasibility of writing the records to the CD-R without needing to change the media too often, we measured the space requirements for executing a set of commands. The configuration file corresponding to what we ran is attached in Appendix A.1. The system priority level was higher than all entries, so all commands in the configuration were active.

As can be seen from the plot in Figure 5.2, the space used over an

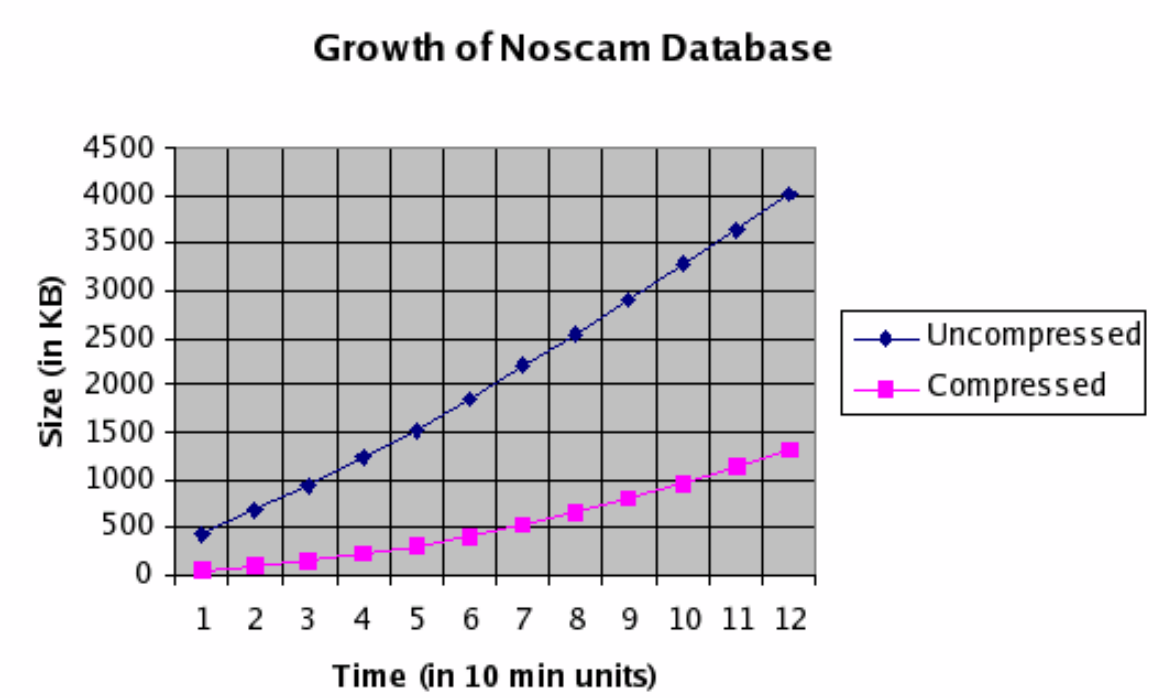

Figure 5.2: NOSCAM database size

execution of two hours was about 4 MB uncompressed and about 1.3MB compressed. If this level of activity was used on the system, then a 700MB CD-R would be able to hold over 6 weeks of audit trail data. A configuration which used more aggressive auditing by way of a large range of commands, files monitored and frequency of sampling would reduce this timeframe. It appears likely, however, that the capacity is enough for many usable configurations.

5.8 Related Work

Previous work that is related to NOSCAM falls into two broad categories. The first consists of tools used to prepare evidence for a legal proceeding. These are of several types. The second is auditing.

5.8.1 Forensic Utilities

The first class includes those that aim to preserve pristine copies of a resource such as a hard disk. This is either done by attempting to make a bit by bit replica of the original on hardware that is identical to that of the source, or by creating a digital image of the bitstream on an intermediate resource. If the latter is done, then it is either followed by a process that uses the image to recreate the data on other hardware or operated on by software that provides an interface to the internal contents of the image. These tasks can be achieved under a Unix-like operating system through the use of the system utility *dd*.

The next class of utilities aim to gather data from areas not accessible through an operating systems system call interface. Two examples of this are the recovery of content in slack space and free space. The former refers to data from an erased file that exist in a disk page that has been reused for a new file but whose unused portion has not been deleted. The latter refers to entire pages that have been freed and not reallocated. Both result from the design decision made in most operating systems not to actually delete the content of disk pages when the file is deleted, in the interest of achieving faster delete operations. Such utilities do not run on live resources and are hence outside the scope of NOSCAM.

The third class of tools is those that are used to analyze an extent of data. The most frequent instantiation of this class involves the creation of a list of keywords of interest to a forensic analyst, the construction of a database of the occurrence of these words in the gathered data, and ability to query the database for pointers to locations where specific words occur.

More general tools such as glimpse [Manber94] serve this need on Unix systems.

The fourth class is the verification of the integrity of the evidence. The typical means of effecting this is to compute a checksum over the data in question, storing the value and repeating the process subsequently, checking if the computation output matches. On a Unix system, the *md5sum* utility which computes an MD5 cryptographic hash may be used.

Most of the tools that fall into the above categories are complementary to NOSCAM and can be invoked from our tool to effect their functionality within our framework.

5.8.2 Auditing Utilities

The second broad category of related work is that done on auditing. Since there have been numerous research efforts in the area, we describe below the aspects that distinguish our tool from others available.

NOSCAM differs from prior work on auditing in several respects. It opts to audit a broad set of events in a limited temporal window, as opposed to using a narrow definition of security-related events over a long period of time as in the NIST specification [Barkley94]. This is to allow the forensic analyst enough contextual information to reconstruct an intrusion.

Further, NOSCAM seeks to prevent the modification of the audit trail, rather than just detect the existence of changes as effected by integrity checks, such as *cryptographic hash chains* [Schneier99].

Additionally, NOSCAM pro-actively invokes other system utilities and records their output, rather than relying on applications to report audit information as does the *syslog* tool [Lonvick01].

Finally, NOSCAM does not rely on network connectivity for the safe deposition of data when the host is under attack as do tools like syslog [Lonvick01] and Remote Access Service [MS96].

5.9 Summary

NOSCAM provides the forensic analyst with a flexible means of adding a wide range of auditing capabilities, including multiple temporal versions of each. Additionally, it allows parts of the audit trail to be stored on a tamper-resistant medium, guarding against the common problem of deletion of the evidence upon which a forensic analysis is dependent.

Chapter 6:
Rapid Runtime Response

6.1 Overview

A host-based intrusion detection system monitors the events in the operating system watching for patterns that signify an attack. If it is complemented by an intrusion response system, then it can initiate a defensive course of action immediately. This reduces the window of exposure during which an attacker can effect damage. Previous work on automated response focused on developing primitives for use in specific scenarios, such as disabling a user account after multiple authentication failures. We introduce a general model for real-time analysis and management of the risk posed to a host. It requires and utilizes operating system support for dynamically altering the access control, data protection and auditing subsystem configurations. It searches, selects and implements a course of action that reduces the risk below the threshold of tolerance while minimizing the impact on performance. Finally, we describe RheoStat, our prototype detection and response engine.

6.2 Background

If a system is simple, its properties can be completely ascertained, either analytically or empirically. When a system is complex, analytical tools can not address all issues and empirical techniques require more resources than can typically be devoted to the task of verification. To secure an information processing system, it is necessary to ensure that it obeys a

set of rules and maintains a set of properties. Modern computing systems are complex. This makes it infeasible to address the task empirically. Analytical techniques can alleviate the issue, but they can not resolve it completely [Harrison76]. In this context, *risk* serves as a measure of the extent to which the security of the system is likely to be violated.

Early approaches to computer security *risk management* employed static strategies, such as the use of passwords, discretionary or mandatory access control, and encrypted network connections [Fletcher95]. These approaches did not provide the flexibility needed to allow risk managers to alter the levels of risk they were willing to tolerate in exchange for commensurate costs. Hence, techniques to dynamically vary the risk were developed. Inherent in the new approach was the need for *risk analysis* to quantify the level of risk present in each configuration of a system.

Large data processing centers started to use the Annual Loss Expectancy (ALE) metric [FIPS31], [FIPS65]. To compute it, the set of all possible *hazards* that could impact the system over the course of a year was enumerated as $H = \{h_1, h_2, \ldots, h_n\}$. The loss associated with each hazard h_α was denoted by $l(h_\alpha)$ and the frequency with which it was likely to occur was denoted by $f(h_\alpha)$. The risk was then calculated using:

$$ALE = \sum_{\alpha=1}^{\alpha=n} f(h_\alpha) \times l(h_\alpha) \tag{6.1}$$

The utilization of the paradigm by a number of commercial tools [NIST91] coupled with a focused research effort [CSRMMBW88], [CSRMMBW89], [CSRMMBW90], [CSRMMBW91] resulted in a number of improvements. The hazard construct was decomposed into threat and vulnerability components. The likelihood of a threat being present was added as a factor,

replacing the hazard frequency. Each vulnerability was coupled with an associated safeguard. The loss from a hazard's occurrence was modeled as a set of consequences that could affect each asset under consideration. Finally, risk management was formulated as the maintenance of a set of requirements framed as constraints on the aforementioned factors [NIST800-12].

6.3 Runtime Risk Management

The primary goal of an intrusion response system is to guard against attacks. Primitives that address specific threats have been developed. However, invoking these arbitrarily may safeguard part of the system but leave other weaker areas exposed. Thus, to effect a rational response, it is necessary to weigh all the possible alternatives. A course of action must then be chosen which will result in the least damage, while simultaneously assuring that cost constraints are respected. This is the very problem that risk management addresses.

6.3.1 Risk Factors

Analyzing the risk that a system is faced with requires knowledge of a number of factors. Below we describe each of these factors along with its associated semantics. We define these in the context of the operating system paradigm since our goal is host-based response.

The paradigm assumes the existence of an operating system with a trusted reference monitor that mediates access by subjects to objects in the system. In addition, the file system and auditing subsystem are as-

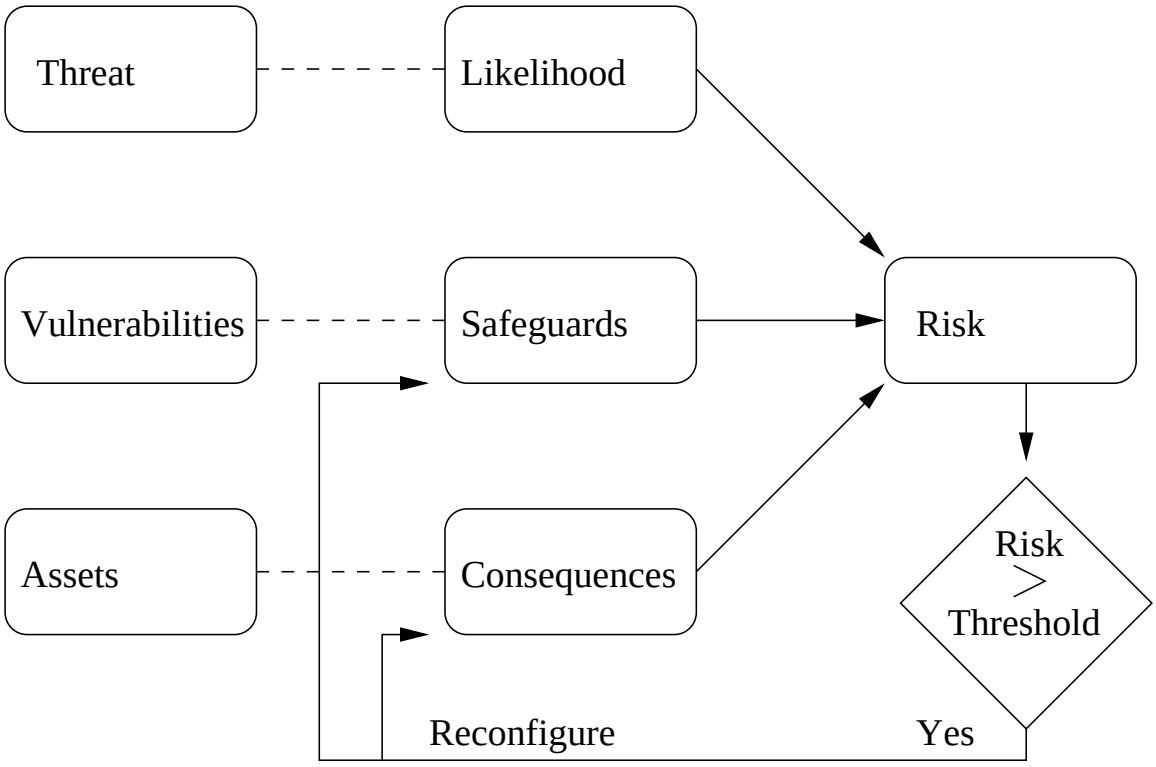

Figure 6.1: Risk can be analyzed as a function of the threats, their likelihood, the vulnerabilities, the safeguards, the assets and the consequences. Risk can be managed by using the safeguards to control the exposure of vulnerabilities and manipulating the assets to limit the consequences.

sumed to be trusted components in the operating system. Finally, a host-based intrusion detection system is assumed to be present and operational.

Threats A *threat* is an agent that can cause harm to an asset in the system. We define a threat to be a specific attack against any of the application or system software that is running on the host. It is characterized by an intrusion detection signature. The set of threats is denoted by $T = \{t_1, t_2, \ldots\}$, where $t_\alpha \in T$ is an intrusion detection signature. Since t_α is a host-based signature, it is comprised of an *ordered set* of events $S(t_\alpha) = \{s_1, s_2, \ldots\}$. If this set occurs in the order recognized by the rules of the intrusion detector, it signifies the presence of an attack.

Likelihood The *likelihood* of a threat is the hypothetical probability of it occurring. If a signature has been partially matched, the extent of the match serves as a predictor of the chance that it will subsequently be completely matched. A function μ is used to compute the likelihood of threat t_α. μ can be threat specific and will depend on the history of system events that are relevant to the intrusion signature. Thus, if $E = \{e_1, e_2, \ldots\}$ denotes the ordered set of all events that have occurred, then:

$$\mathcal{T}(t_\alpha) = \mu(t_\alpha, E \overset{\prec}{\cap} S(t_\alpha)) \tag{6.2}$$

where $\overset{\prec}{\cap}$ yields the set of all events that occur *in the same order* in each input set.

Assets An *asset* is an item that has value. We define the assets to be the data stored in the system. In particular, each file is considered

a separate object $o_\beta \in O$, where $O = \{o_1, o_2, \ldots\}$ is the set of assets. A set of objects $A(t_\alpha) \subseteq O$ is associated with each threat t_α. Only objects $o_\beta \in A(t_\alpha)$ can be harmed if the attack that is characterized by t_α succeeds.

Consequences A *consequence* is a type of harm that an asset may suffer. Three types of consequences can impact the data. These are the loss of confidentiality, integrity and availability. If an object $o_\beta \in A(t_\alpha)$ is affected by the threat t_α, then the resulting costs due to the loss of confidentiality, integrity and availability are denoted by $c(o_\beta)$, $i(o_\beta)$, and $a(o_\beta)$ respectively. Any of these values may be 0 if the attack can not effect the relevant consequence. However, all three values associated with a single object can not be 0 since in that case $o_\beta \in A(t_\alpha)$ would not hold. Thus, the consequence of a threat t_α is:

$$\mathcal{C}(t_\alpha) = \sum_{o_\beta \in A(t_\alpha)} c(o_\beta) + i(o_\beta) + a(o_\beta) \tag{6.3}$$

By removing an asset from the system, the consequences it faces can be curtailed. In the case of data availability, replication serves this purpose, while in the case of confidentiality and integrity, cryptographic operations can be used. For the purpose of estimating risk, a consequence *curtailment* effectively removes the asset from the analysis.

Vulnerabilities A *vulnerability* is a weakness in the system. It results from an error in the design, implementation or configuration of either the operating system or application software. The set of vulnerabilities present in the system is denoted by $W = \{w_1, w_2, \ldots\}$. $W(t_\alpha) \subseteq W$

is the set of weaknesses exploited by the threat t_α to subvert the security policy.

Safeguards A *safeguard* is a mechanism that controls the exposure of the system's assets. The reference monitor's set of permission checks $P = \{p_1, p_2, \ldots\}$ serve as safeguards in an operating system. Since the reference monitor mediates access to all objects, a vulnerability's exposure can be limited by denying the relevant permissions. The set $P(w_\gamma) \subseteq P$ contains all the permissions that are requested in the process of exploiting vulnerability w_γ. The static configuration of a conventional reference monitor either grants or denies access to a permission p_λ. This *exposure* is denoted by $v(p_\lambda)$, with the value being either 0 or 1. The active reference monitor can reduce the exposure of a statically granted permission to $v'(p_\lambda)$, a value in the range $[0, 1]$. This reflects the nuance that results from evaluating predicates as *auxiliary safeguards*.)

Thus, if all auxiliary safeguards are utilized, the total exposure to a threat t_α is:

$$\mathcal{V}(t_\alpha) = \sum_{p_\lambda \in \hat{P}(t_\alpha)} \frac{v(p_\lambda) \times v'(p_\lambda)}{|\hat{P}(t_\alpha)|} \tag{6.4}$$

where:

$$\hat{P}(t_\alpha) = \bigcup_{w_\gamma \in W(t_\alpha)} P(w_\gamma) \tag{6.5}$$

6.3.2 Risk Analysis

The risk to the host is the sum of the risks that result from each of the threats that it faces. The risk from a single threat is the product of the

chance that the attack will occur, the exposure of the system to the attack, and the cost of the consequences of the attack succeeding [NIST800-12]. Thus, the cumulative risk faced by the system is:

$$\mathcal{R} = \sum_{t_\alpha \in T} \mathcal{T}(t_\alpha) \times \mathcal{V}(t_\alpha) \times \mathcal{C}(t_\alpha) \tag{6.6}$$

6.3.3 Risk Management

If the risk posed to the system is to be managed, the current level must be continuously monitored. When the risk rises past the threshold that the host can tolerate, the system's security must be tightened. Similarly, when the risk decreases, the restrictions can be relaxed to improve performance and usability. This process is elucidated below.

The system's risk can be reduced either by reducing the exposure of vulnerabilities or limiting the consequences to the data in the event of a successful attack. The former is effected through the use of auxiliary safeguards prior granting a permission. The latter is realized by cryptographically protecting threatened files. Additionally, both approaches may be used simultaneously. Similarly, if the threat reduces, the restrictive permission checks and data protection can be relaxed.

Managed Risk

The set of permissions P is kept partitioned into two disjoint sets, $\Psi(P)$ and $\Omega(P)$, that is $\Psi(P) \cap \Omega(P) = \phi$ and $\Psi(P) \cup \Omega(P) = P$. The set $\Psi(P) \subseteq P$ contains the permissions for which auxiliary safeguards are currently active. The remaining permissions $\Omega(P) \subseteq P$ are handled conventionally by the reference monitor, using only static lookups rather than evaluating as-

sociated predicates prior to granting these permissions. Similarly, the set of files O is kept partitioned into two disjoint sets, $\Psi(O)$ and $\Omega(O)$, where $\Psi(O) \cap \Omega(O) = \phi$ and $\Psi(O) \cup \Omega(O) = O$. The set $\Psi(O) \subseteq O$ contains the files that are currently inaccessible and unmodifiable due to their cryptographic encapsulation. The remaining files $\Omega(O) \subseteq O$ are transparently accessible and modifiable.

At any given point, when safeguards $\Psi(P)$ and curtailments $\Psi(O)$ are in use, the current risk $\mathcal{R}'$ is calculated with:

$$\mathcal{R}' = \sum_{t_\alpha \in T} \mathcal{T}(t_\alpha) \times \mathcal{V}'(t_\alpha) \times \mathcal{C}'(t_\alpha) \tag{6.7}$$

where:

$$\mathcal{V}'(t_\alpha) = \sum_{p_\lambda \in \hat{P}(t_\alpha) \cap \Omega(P)} \frac{v(p_\lambda)}{|\hat{P}(t_\alpha)|} + \sum_{p_\lambda \in \hat{P}(t_\alpha) \cap \Psi(P)} \frac{v(p_\lambda) \times v'(p_\lambda)}{|\hat{P}(t_\alpha)|} \tag{6.8}$$

and:

$$\mathcal{C}'(t_\alpha) = \sum_{o_\beta \in A(t_\alpha) \cap \Omega(O)} c(o_\beta) + i(o_\beta) + a(o_\beta) \tag{6.9}$$

Risk Tolerance

While the risk must be monitored continuously, there is a computational cost incurred each time it is recalculated. Therefore, the frequency with which the risk is estimated must be minimized to the extent possible. Instead of calculating the risk synchronously at fixed intervals in time, we exploit the fact that the risk level only changes when the threat to the system is altered.

An intrusion detector is assumed to be monitoring the system's activity. Each time it detects an event that changes the extent to which a signature

has been matched, it passes the event e to the intrusion response subsystem. The level of risk $\mathcal{R}_b$ before e occurred is noted, and then the level of risk $\mathcal{R}_a$ after e occurred is calculated. Thus, $\mathcal{R}_a = \mathcal{R}_b + \epsilon$, where ϵ denotes the change in the risk. Since the risk is recalculated only when it actually changes, the computational cost of monitoring it is minimized.

Each time an event e occurs, either the risk decreases, stays the same or increases. Each host is configured to tolerate risk upto a threshold, denoted by $\mathcal{R}_0$. After each event e, the system's response guarantees that the risk will return to a level below this threshold. As a result, $\mathcal{R}_b < \mathcal{R}_0$ always holds. If $\epsilon = 0$, then no further risk management steps are required.

If $\epsilon < 0$, then $\mathcal{R}_a < \mathcal{R}_0$ since $\mathcal{R}_a = \mathcal{R}_b + \epsilon < \mathcal{R}_b < \mathcal{R}_0$. At this point, the system's security configuration is more restrictive than it needs to be. To improve system usability and performance, the response system must deactivate appropriate safeguards and curtailments, while ensuring that the risk level does not rise past the threshold $\mathcal{R}_0$.

If $\epsilon > 0$ and $\mathcal{R}_a \leq \mathcal{R}_0$, then no action needs to be taken. Even though the risk has increased, it is below the threshold that the system can tolerate, so no further safeguards or curtailments need to be introduced. In addition, the system will not be able to find any set of unused safeguards and curtailments whose removal will increase the risk by less than $\mathcal{R}_0 - \mathcal{R}_b - \epsilon$, since the presence of such a combination would also mean that the set existed before e occurred. It is not possible that such a combination of safeguards and curtailments existed before e occurred since they would also have satisfied the condition of being less than $\mathcal{R}_0 - \mathcal{R}_b$ and would have been utilized before e occurred in the process of minimizing the impact on performance in the previous step.

If $\epsilon > 0$ and $\mathcal{R}_a > \mathcal{R}_0$, then action is required to reduce the risk to a level below the threshold of tolerance. The response system must search for and implement a set of safeguards and curtailments to this end.

Recalculating Risk

When the risk is calculated the first time, Equation 6.6 is used. Therefore, the cost is $O(|T| \times |P| \times |O|)$. Since the change in the risk must be repeatedly evaluated during real-time reconfiguration of the runtime environment, it is imperative the cost is minimized. This is achieved by caching all the values $\mathcal{V}'(t_\alpha) \times \mathcal{C}'(t_\alpha)$ associated with threats $t_\alpha \in T$ during the evaluation of Equation 6.6. Subsequently, when an event e occurs, the change in the risk $\epsilon = \delta(\mathcal{R}', e)$ can be calculated with cost $O(|T|)$ as described below.

The ordered set E refers to all the events that have occurred in the system prior to the event e. The change in the likelihood of a threat t_α due to e is:

$$\delta(\mathcal{T}(t_\alpha), e) = \mu(t_\alpha, (E \cup e) \overset{\curvearrowleft}{\cap} S(t_\alpha)) - \mu(t_\alpha, E \overset{\curvearrowleft}{\cap} S(t_\alpha)) \tag{6.10}$$

The set of threats affected by e is denoted by $\Delta(T, e)$. A threat $t_\alpha \in \Delta(T, e)$ is considered to be affected by e if $\delta(\mathcal{T}(t_\alpha), e) \neq 0$, that is its likelihood changed due to the event e. The resultant change in the risk level is:

$$\delta(\mathcal{R}', e) = \sum_{t_\alpha \in \Delta(T,e)} \delta(\mathcal{T}(t_\alpha), e) \times \mathcal{V}'(t_\alpha) \times \mathcal{C}'(t_\alpha) \tag{6.11}$$

6.3.4 Cost/Benefit Analysis

After an event e occurs, if the risk level $\mathcal{R}_a$ increases past the threshold of risk tolerance $\mathcal{R}_0$, the goal of the response engine is to reduce the risk by

$\delta_g \geq \mathcal{R}_a - \mathcal{R}_0$ to a level below the threshold. To do this, it must select a subset of permissions $\rho(\Omega(P)) \subseteq \Omega(P)$ and a subset of objects $\rho(\Omega(O)) \subseteq \Omega(O)$, such that adding safeguards and curtailments respectively to the two sets will reduce the risk to the desired level. By ensuring that the permissions in $\rho(\Omega(P))$ are granted only after relevant predicates are verified and files in $\rho(\Omega(O))$ are cryptographically protected, the resulting risk level is reduced to:

$$\mathcal{R}'' = \sum_{t_\alpha \in T} \mathcal{T}(t_\alpha) \times \mathcal{V}''(t_\alpha) \times \mathcal{C}''(t_\alpha) \tag{6.12}$$

where the new vulnerability measure, based on Equation 6.4, is:

$$\mathcal{V}''(t_\alpha) = \sum_{p_\lambda \in (\hat{P}(t_\alpha) \cap \Omega(P) - \rho(\Omega(P)))} \frac{v(p_\lambda)}{|\hat{P}(t_\alpha)|} + \sum_{p_\lambda \in (\hat{P}(t_\alpha) \cap \Psi(P) \cup \rho(\Omega(P)))} \frac{v(p_\lambda) \times v'(p_\lambda)}{|\hat{P}(t_\alpha)|} \tag{6.13}$$

and the new consequence measure, based on Equation 6.3, is:

$$\mathcal{C}''(t_\alpha) = \sum_{o_\beta \in (A(t_\alpha) \cap \Omega(O) - \rho(\Omega(O)))} c(o_\beta) + i(o_\beta) + a(o_\beta) \tag{6.14}$$

Instead, after an event e occurs, if the risk level $\mathcal{R}_a$ decreases, the goal of the response engine is to allow the risk to rise by $\delta_g \leq \mathcal{R}_0 - \mathcal{R}_a$ to a level below the threshold of risk tolerance $\mathcal{R}_0$. To do this, it must select a subset of permissions $\rho(\Psi(P)) \subseteq \Psi(P)$ and a subset of objects $\rho(\Psi(O)) \subseteq \Psi(O)$, such that removing the safeguards and curtailments currently in use for these two sets will yield the maximum improvement to runtime performance. After the safeguards and curtailments are relaxed, the risk level will rise to:

$$\mathcal{R}'' = \sum_{t_\alpha \in T} \mathcal{T}(t_\alpha) \times \mathcal{V}''(t_\alpha) \times \mathcal{C}''(t_\alpha) \tag{6.15}$$

where the new vulnerability measure, based on Equation 6.4, is:

$$\mathcal{V}''(t_\alpha) = \sum_{p_\lambda \in \hat{P}(t_\alpha) \cap \Omega(P) \cup \rho(\Psi(P))} \frac{v(p_\lambda)}{|\hat{P}(t_\alpha)|} + \sum_{p_\lambda \in \hat{P}(t_\alpha) \cap \Psi(P) - \rho(\Psi(P))} \frac{v(p_\lambda \times v'(p_\lambda))}{|\hat{P}(t_\alpha)|} \quad (6.16)$$

and the new consequence measure, based on Equation 6.3, is:

$$\mathcal{C}''(t_\alpha) = \sum_{o_\beta \in A(t_\alpha) \cap \Omega(O) \cup \rho(\Psi(O))} c(o_\beta) + i(o_\beta) + a(o_\beta) \quad (6.17)$$

There are $O(2^{(|P|+|O|)})$ ways of choosing subsets $\rho(\Omega(P)) \subseteq \Omega(P)$ and $\rho(\Omega(O)) \subseteq \Omega(O)$ for risk reduction or subsets $\rho(\Psi(P)) \subseteq \Psi(P)$ and $\rho(\Psi(O)) \subseteq \Psi(O)$ for risk relaxation. When selecting from the possibilities, the primary constraint is the maintenance of the bound $\mathcal{R}'' < \mathcal{R}_0$, where $\mathcal{R}'' = \mathcal{R}_a - \delta_g$ in the case of risk reduction, and $\mathcal{R}'' = \mathcal{R}_a + \delta_g$ in the case of risk relaxation.

The choice of safeguards and curtailments also impacts the performance of the system. Evaluating predicates prior to granting permissions introduces latency in system calls. Cryptographically protecting objects decreases usability. Hence, the choice of subsets $\rho(\Omega(P))$ and $\rho(\Omega(O))$ or subsets $\rho(\Psi(P))$ and $\rho(\Psi(O))$ is subject to the secondary goal of minimizing the overhead introduced.

The adverse impact of a safeguard or curtailment is proportional to the frequency with which it is utilized in the system's workload. Given a typical workload, we can count the frequency $f(p_\lambda)$ with which permission p_λ is requested in the workload. Similarly, we can count the frequency $f(o_\beta)$ with which file o_β is accessed in the workload. This can be done for all permissions and files. The cost of utilizing subsets $\rho(\Omega(P))$ and $\rho(\Omega(O))$ for risk reduction can then be calculated with:

$$\zeta(\rho(\Omega(P)), \rho(\Omega(O))) = \sum_{p_\lambda \in \rho(\Omega(P))} f(p_\lambda) + \sum_{o_\beta \in \rho(\Omega(O))} f(o_\beta) \quad (6.18)$$

Similarly, if the safeguards of subset $\rho(\Psi(P))$ and the curtailments of consequences to assets in subset $\rho(\Psi(O))$ are relaxed, the resulting reduction in runtime cost can be calculated with:

$$\zeta(\rho(\Psi(P)), \rho(\Psi(O))) = \sum_{p_\lambda \in \rho(\Psi(P))} f(p_\lambda) + \sum_{o_\beta \in \rho(\Psi(O))} f(o_\beta) \tag{6.19}$$

The ideal choice of safeguards and curtailments will minimize the safeguards' and curtailments' impact on performance, while simultaneously ensuring that the risk remains below the threshold of tolerance. Thus, for risk reduction we wish to find:

$$\min \zeta(\rho(\Omega(P)), \rho(\Omega(O))), \quad \mathcal{R}'' \leq \mathcal{R}_0 \tag{6.20}$$

In the context of risk relaxation, we wish to find:

$$\max \zeta(\rho(\Psi(P)), \rho(\Psi(O))), \quad \mathcal{R}'' \leq \mathcal{R}_0 \tag{6.21}$$

6.3.5 Complexity

We note that the semantics of risk management require that at each step the risk must be reduced below the threshold of tolerance. This precludes optimization strategies such as minimizing a weighted sum of risk and runtime performance. We conclude that runtime risk management is a $0-1$ integer non-linear programming problem with a linear objective function and quadratic constraint. The decision problem that corresponds to the aforementioned optimization problem is NP-hard [Garey79] as argued below.

Optimization Problem

Risk reduction can viewed as selecting a set of vertices in a vertex-weighted, edge-weighted bipartite graph, such that the sum of the weights of the

vertices selected is minimized, subject to the constraint that the sum of the weights of the edges present in the subgraph induced by the selected vertices is greater than a fixed threshold. The vertices in one partition correspond to safeguards, while the vertices in the other partition correspond to curtailments. The weight of each vertex is the frequency of the corresponding permission or object in the workload. The weight of an edge between a safeguard and a curtailment is the contribution to the total risk that results from the exposure of the corresponding permission and the cost of the corresponding object's security being subverted. The fixed threshold is the risk tolerance. Risk relaxation is similar, with the exception that the sum of weights of the chosen vertices must be maximized, while the sum of the weights of the edges in the induced subgraph must remain below a fixed threshold. The two optimization problems are equivalent.

Decision Problem

The decision problem for risk reduction takes as input: (i) a bipartite graph of the form described above, (ii) a fixed threshold which is the risk tolerance, and (iii) the sum of the weights of the subset of vertices to be chosen, which corresponds to the runtime cost of the primitives in a proposed response. The output is only `true` if the algorithm is able to find a subset of vertices whose weights add up to the specified total, while the sum of the weights of the edges in the induced subgraph is at least the specified threshold.

NP-Hard

Given an algorithm for the risk reduction optimization problem, the decision problem can be solved by checking if the target sum of vertices' weights is less than, equal or greater than the minimum cost output by the optimization algorithm.

Given a decision algorithm for risk reduction, we can solve the *maximum edge biclique problem* which takes a bipartite graph and a threshold as inputs and outputs whether the graph includes a biclique that is the size of the threshold or larger. The problem is known to be NP-complete [Peeters03].

To solve the maximum edge biclique problem, we repeatedly invoke the risk reduction decision algorithm. The number of invocations is bounded above by the size of the vertex set in the graph. The input is the bipartite graph (with all vertices and edges weighted 1), the threshold and a target number of vertices that ranges during invocations from 1 to the total number of vertices in the bipartite graph. The first time the output is `true`, we stop and output `true` for the maximum edge bliclique problem.

Since the risk reduction decision algorithm output `true`, the sum of the weights of the vertices (which is equal to the number of vertices since all the weights were set to 1) is the least possible such that the sum of the weights of the edges (which is equal to the number of edges since the weights of all the edges were set to 1) was at least the threshold specified. The total number of edges in a biclique is the maximum possible for a subset of vertices of the biclique's size. Thus, the smallest subset of vertices that will contain a specified number of edges is a biclique.

If all the invocations of the risk reduction decision problem produced an output of `false`, then the output for the maximum edge biclique problem is also `false`. This completes the reduction. If the risk reduction decision problem were *tractable*, then the maximum edge biclique problem would also be tractable, but it is known to be NP-complete. Therefore, the risk reduction (and risk relaxation since it is analogous) decision problems are NP-hard.

6.4 Design

While JStat's purpose was limited to intrusion detection, RheoStat's goal extends to automated response. Since the choice of response depends on runtime risk estimates, a new component is required for managing this functionality. In addition, since risk can decrease over time, RheoStat must be augmented to model this. These two changes are described below.

6.4.1 Threat Expiration

JStat's design was based on that of STAT, in which an intrusion is modeled as the system proceeding through a series of states. In each state, the only transition possible is to a state which is one step closer to the final state. For the purpose of intrusion detection, where the only result of consequence is whether the rule has entered the final state or not, this sufficed. However, it is also possible for unrelated inoffensive processes to coincidentally generate the requisite sequence of system states if the system is monitored for an extended period of time. Other intrusion detectors address this issue by requiring that the requisite events occur within a specified temporal window.

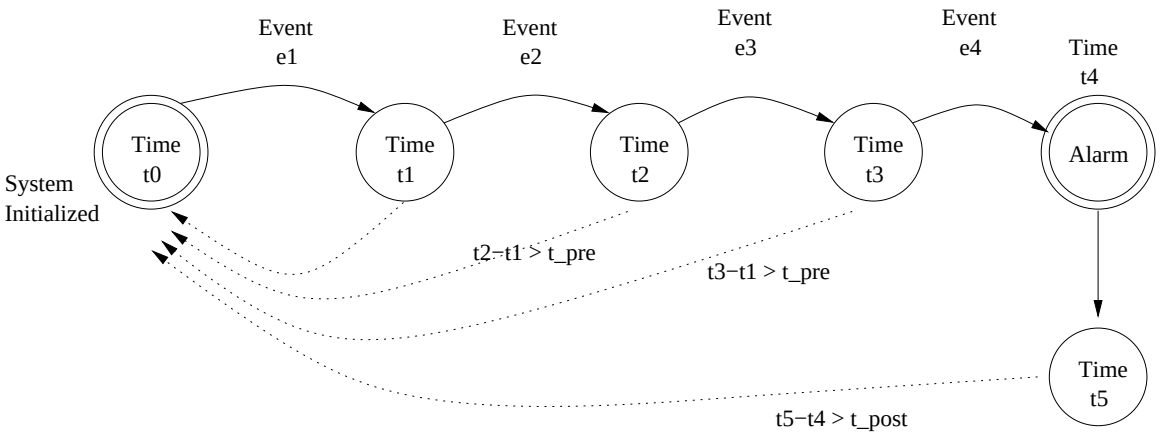

Figure 6.2: Two timers are associated with each signature. When the first event in the signature occurs, the *pre-match timer* is activated. If it expires before the rule enters the final state, it resets the rule to the uninitialized configuration. The *post-match timer* is activated after a rule has been matched. It serves as the delay before the system assumes the detected threat has passed.

Pre-Match Timer

RheoStat uses the partial match information as an estimate of the current threat level. Thus, if it is possible to determine that the partial match of a rule was caused by innocuous system activity, the rule should be reset so that the corresponding threat estimate can be corrected. This is achieved by starting a timer once the first state of the rule is matched. If the timer expires before the rule transitions to the final state, then the rule is reset to the original configuration when no events had occurred.

Post-Match Timer

During the course of an intrusion rule being matched, RheoStat attempts to take precautionary measures to limit the effect of the attack. Thus, a complete signature match does not signify the system being compromised, though the safeguards and curtailments instituted may signifi-

cantly impede the system's usability. This is particularly likely in the case of anomaly detection rules since they are an indicator of a probable intrusion rather than a domain specific definition of a compromise. Once an attack has been warded off, the system can opt to withdraw the relevant safeguards and curtailments. If this is done immediately, however, an attacker is likely to attempt to reuse the same exploit. To prevent this, a delay is introduced for the duration that the threat is likely to persist, before the rule is reset to its original configuration.

Interval choices

When determining either the length of time that will be allowed for a signature to match completely before it is reset, or the length of time that will be allowed to pass after a completed match before the threat is deemed to have passed, a tradeoff is involved. The greater the length, the less likely it is that an attacker can compromise the system since it is more likely to have safeguards and curtailments in place. During this period, the system usability is also decreased, so the delay before the relevant safeguards and curtailments are withdrawn should be minimized to the extent possible. Since the potential for harm is lower in the case of a partial match, its value will typically be small. Similarly, the delay after an intrusion signature matches completely will be large since it brings with it the possibility of more severe consequences.

6.4.2 Response Choice

Determining the optimal choice of safeguards and curtailments for risk management corresponds to an NP-hard problem, as argued in Section

6.3.5. Since the choice is to be made in real-time, we will use a heuristic which guarantees that the risk threshold is maintained. The heuristic uses the greedy strategy of picking the response primitive with the highest benefit-to-cost ratio repeatedly till the constraint is satisfied. By maintaining the choices in a *heap* data structure keyed on the benefit-to-cost ratio, the first primitive in the response set can be chosen in $O(1)$ time. This is significant since implementing a single response primitive is often sufficient for disrupting an attack in progress.

Since the benefit associated with each unutilized safeguard or curtailment is the degree to which the risk will be reduced if it is used, this is a function of other safeguards or curtailments related to the threats that it affects. Similarly, since the loss of benefit associated with each currently utilized safeguard or curtailment is the degree to which the risk will increase if it is used, this is also a function of the other safeguards or curtailments associated with the threats that it affects. As a result, the benefit of adding or removing each safeguard or curtailment must be recalculated each time other safeguards or curtailments are added or removed.

Risk Reduction

We outline the algorithm for the case where the risk needs to be reduced. The first two steps constitute pre-processing and therefore only occur during system initialization.

Step 1 The benefit-to-cost ratio of each candidate safeguard permis-

sion $p_\lambda \in \Omega(P)$ can be calculated by:

$$\kappa(p_\lambda) = \frac{\sum_{t_\alpha : p_\lambda \in (\hat{P}(t_\alpha) \cap \Omega(P))} \mathcal{T}(t_\alpha) \times \frac{v(p_\lambda) \times (1 - v'(p_\lambda))}{|\hat{P}(t_\alpha)|} \times \mathcal{C}'(t_\alpha)}{f(p_\lambda)} \quad (6.22)$$

Step 2 Similarly, the benefit-to-cost ratio of protecting an object $o_\beta \in \Omega(O)$ can be calculated by:

$$\kappa(o_\beta) = \frac{(c(o_\beta) + i(o_\beta) + a(o_\beta)) \quad \times \sum_{t_\alpha : o_\beta \in (A(t_\alpha) \cap \Omega(O))} \mathcal{T}(t_\alpha) \times \mathcal{V}'(t_\alpha)}{f(o_\beta)} \quad (6.23)$$

Step 3 The response sets are defined as empty, that is $\rho(\Omega(P)) = \rho(\Omega(O)) = \phi$.

Step 4 The single risk reducing measure with the highest benefit-to-cost can be selected, that is:

$$\max\ p_{max}, o_{max} \quad where: \quad (6.24)$$

$$p_{max} = \max\ \kappa(p_\lambda), \quad p_\lambda \in \Omega(P)$$

$$o_{max} = \max\ \kappa(o_\beta), \quad o_\beta \in \Omega(O)$$

If it is a permission it is added to $\rho(\Omega(P))$ and if it is an object it can be added to $\rho(\Omega(O))$.

Step 5 If the choice was a permission p_λ, then the value $\kappa(o_\beta)$ must be recalculated for all objects o_β that are affected by threats which utilize p_λ in the course of their attacks. Thus, each $\kappa(o_\beta)$ must be updated if:

$$o_\beta \in \bigcup_{t_\alpha : p_\lambda \in \hat{P}(t_\alpha)} A(t_\alpha) \quad (6.25)$$

Instead, if the choice was an object o_β, then the value $\kappa(p_\lambda)$ must be recalculated for all permissions p_λ that are utilized in the course of an attack that affects object o_β. Thus, each $\kappa(p_\lambda)$ must be updated if:

$$p_\lambda \in \bigcup_{t_\alpha : o_\beta \in A(t_\alpha)} \hat{P}(t_\alpha) \tag{6.26}$$

Step 6 The risk before the candidate responses were utilized is $\mathcal{R}_a$. If the responses were activated the resulting risk $\mathcal{R}''$ is given by:

$$\mathcal{R}'' = \mathcal{R}_a \quad - \sum_{p_\lambda \in \rho(\Omega(P))} \kappa(p_\lambda) \times f(p_\lambda) \quad - \sum_{o_\beta \in \rho(\Omega(O))} \kappa(o_\beta) \times f(o_\beta) \tag{6.27}$$

This is equivalent to using Equations 6.12, 6.13 and 6.14. While the worst case complexity is the same, when few protective measures are added the cost of the above calculation is significantly lower.

Step 7 If $\mathcal{R}'' > \mathcal{R}_0$ then the system repeats the above from Step 4 onwards. However, if $\mathcal{R}'' \leq \mathcal{R}_0$ then the set of safeguards $\rho(\Omega(P))$ and the set consequence curtailing measures $\rho(\Omega(O))$ must be applied. $\rho(\Omega(P))$ should be transferred from $\Omega(P)$ to $\Psi(P)$ and $\rho(\Omega(O))$ should be transferred from $\Omega(O)$ to $\Psi(O)$. Then the response sets should be reset so $\rho(\Omega(P)) = \rho(\Omega(O)) = \phi$.

The time complexity is:

$$O((\rho(\Omega(P)) + \rho(\Omega(O))) \times (\log|P| + \log|O| + \sum_{t_\alpha \in T} (|\hat{P}(t_\alpha)| + |A(t_\alpha)|))) \tag{6.28}$$

In the worst case, this is $O(|P| + |O|)^2$. Unless a large variety of attacks are simultaneously launched against the target, the first factor will re-

main small. Additionally, if there is a strong correlation between the exposures and the consequences, then the second factor will also remain small. Thus, in practice it is likely to achieve acceptable results.

Risk Relaxation

In the case of risk relaxation, the algorithm becomes:

Step 1 For $p_\lambda \in \Psi(P)$ calculate:

$$\kappa(p_\lambda) = \frac{\sum_{t_\alpha : p_\lambda \in (\hat{P}(t_\alpha) \cap \Psi(P))} \mathcal{T}(t_\alpha) \times \frac{v(p_\lambda) \times (1 - v'(p_\lambda))}{|\hat{P}(t_\alpha)|} \times \mathcal{C}'(t_\alpha)}{f(p_\lambda)} \quad (6.29)$$

Step 2 For $o_\beta \in \Psi(O)$ calculate:

$$\kappa(o_\beta) = \frac{(c(o_\beta) + i(o_\beta) + a(o_\beta)) \quad \times \sum_{t_\alpha : o_\beta \in (A(t_\alpha) \cap \Psi(O))} \mathcal{T}(t_\alpha) \times \mathcal{V}'(t_\alpha)}{f(o_\beta)} \quad (6.30)$$

Step 3 Set $\rho(\Psi(P)) = \rho(\Psi(O)) = \phi$.

Step 4 Find the safeguard or curtailment which yields the least risk reduction per instance of use:

$$\min\ p_{min}, o_{min} \quad where: \quad (6.31)$$

$$p_{min} = \min\ \kappa(p_\lambda), \quad p_\lambda \in \Psi(P)$$

$$o_{min} = \min\ \kappa(o_\beta), \quad o_\beta \in \Psi(O)$$

Add it to $\rho(\Psi(P))$ if it is a permission. Instead, add it to $\rho(\Psi(O))$ if it is a file.

Step 5 Update $\kappa(o_\beta)$ if:

$$o_\beta \in \bigcup_{t_\alpha : p_\lambda \in \hat{P}(t_\alpha)} A(t_\alpha) \tag{6.32}$$

or update $\kappa(p_\lambda)$ if:

$$p_\lambda \in \bigcup_{t_\alpha : o_\beta \in A(t_\alpha)} \hat{P}(t_\alpha) \tag{6.33}$$

depending on whether a permission or a file was chosen in the previous step.

Step 6 Calculate $\mathcal{R}''$:

$$\mathcal{R}'' = \mathcal{R}_a \quad + \sum_{p_\lambda \in \rho(\Psi(P))} \kappa(p_\lambda) \times f(p_\lambda) \quad + \sum_{o_\beta \in \rho(\Psi(O))} \kappa(o_\beta) \times f(o_\beta) \tag{6.34}$$

Step 7 If $\mathcal{R}'' < \mathcal{R}_0$, repeat from Step 4. If $\mathcal{R}'' = \mathcal{R}_0$, proceed to next step. If $\mathcal{R}'' > \mathcal{R}_0$, undo last iteration of Step 4.

Step 8 Relax all measures in $\rho(\Psi(P))$ and $\rho(\Psi(O))$ and transfer them to $\Omega(P)$ and $\Omega(O)$, respectively. Set $\rho(\Psi(P)) = \rho(\Psi(O)) = \phi$.

6.5 Implementation

We now describe how we implemented Support for Automated Host-based Passive Intrusion Response (SAPHIRe). The prototype augments Sun's Java Runtime Environment (version 1.4.2) running on Redhat Linux 9 (with kernel 2.4.20). We first describe the integration of the components and then explain how the **RiskManager** coordinates their activity.

6.5.1 Initialization

Security in the Java 2 model is handled by the **java.security.Access-Controller** class, which in turn invokes the legacy **java.lang.Security-**

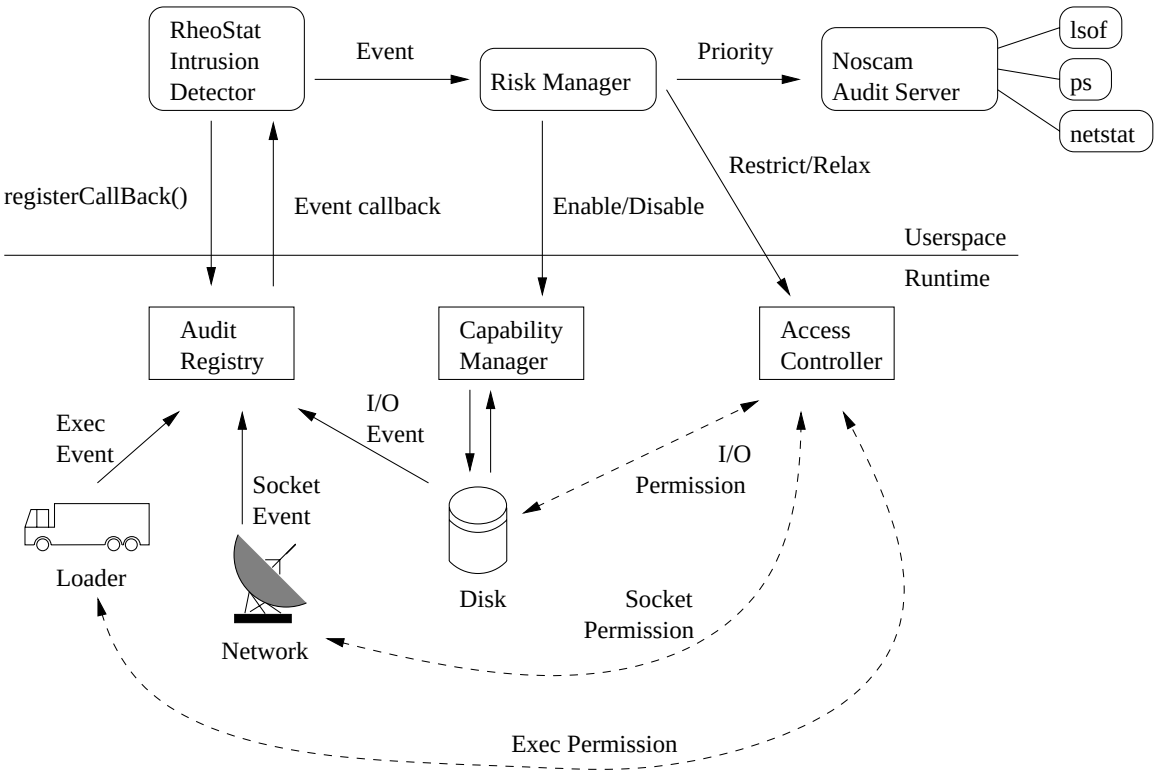

Figure 6.3: The *Risk Manager* receives event information from the intrusion detector. Based on the risk level it adjusts the permissions to access system resources, cryptographic protection of data and auditing activity.

Manager. We instrumented the latter to invoke an **Initialize** class where we create and activate an instance of the state-based intrusion detection and response engine, **RheoStat** and the pro-active auditing subsystem **Noscam**. Shutdown hooks are also registered with the security manager so that these two applications are terminated when the user application exits.

When **RheoStat** initializes, it registers for callbacks from the SADDLE subsystem's **AuditRegistry**, which is activated independently when any auditable activity occurs in the runtime. Audit points have been inserted to report on classloading, filesystem and network activity.

The **java.security.AccessController**'s *checkPermission()* method is in-

strumented to consult the ARM subsystem prior to looking up the conventional access control matrix for granting a permission. ARM's **Active-Monitor** is responsible for performing auxiliary runtime checks. ARM is activated the first time a permission check is performed.

All Java filesystem activity is routed through the **java.io.FileInput-Stream** and **java.io.FileOutputStream** classes (except for that of the recently introduced **java.nio.*** subsystem). These classes' *open* and *close* methods are instrumented to allow the RICE subsystem's **CapabilityManager** the opportunity to cryptographically transform files if necessary. In addition, when writes are performed, the same subsystem handles calculating deltas which can then be replicated to a remote node by an independent process. RICE is activated transparently when the first filesystem activity occurs.

6.5.2 Risk Manager

Once the SAPHIRe subsystems are active, **RheoStat** registers for events from its set of intrusion detection signatures as needed. When it receives a callback from the **AuditRegistry** it updates its intrusion detection signatures and then passes the event to the **RiskManager**. Here the event is used to update the threat level of all relevant intrusion signatures. The μ matching function, described in Section 6.3.1, is used for estimating the threat from a partial match. We use:

$$\mu(t_\alpha, E \tilde{\cap} S(t_\alpha)) = \frac{|E \tilde{\cap} S(t_\alpha)|}{|S(t_\alpha)|} \tag{6.35}$$

With the updated threat levels, the new risk level is calculated. If it crosses a threshold, a call is made to **Noscam**'s *setAuditLevel()* method to alter its

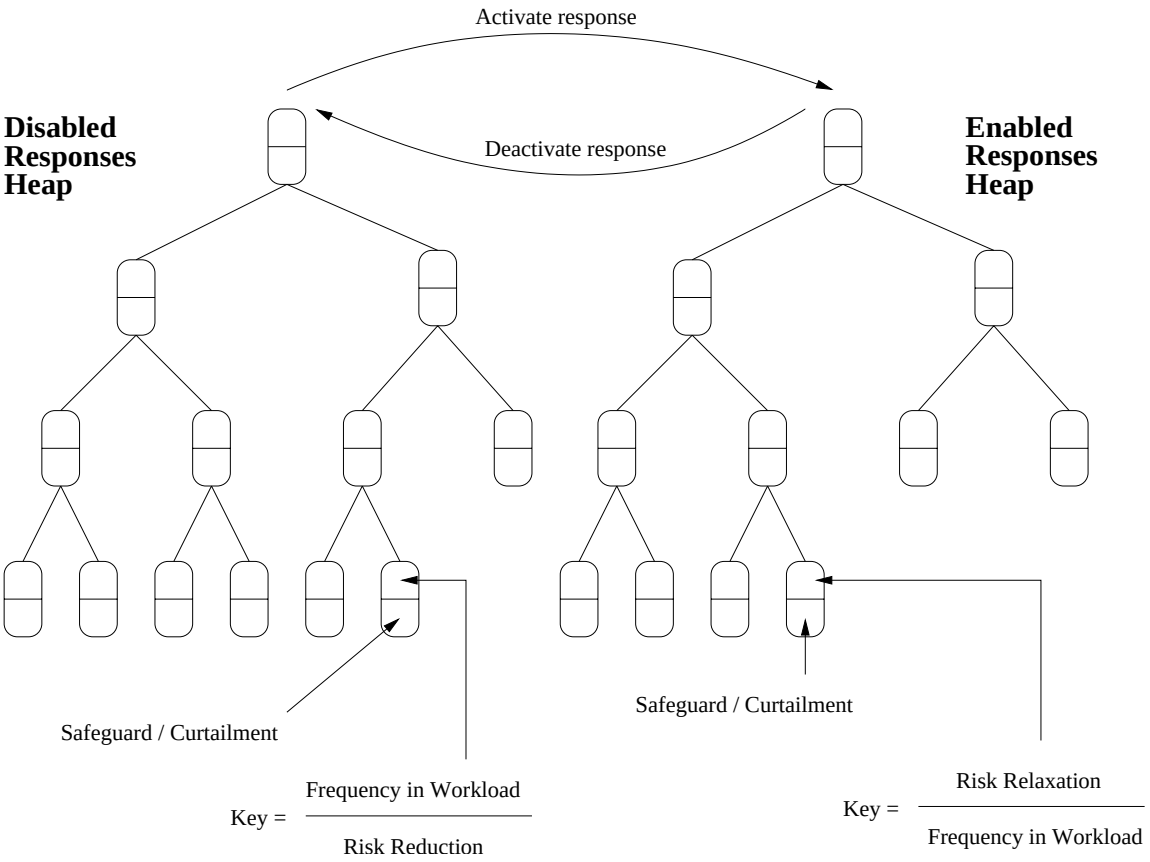

Figure 6.4: When the Risk Manager needs to activate a response to effect risk reduction, it attempts to select the one which will minimize the runtime overhead while maximizing the risk reduction. When the Risk Manager needs to deactivate a response to effect risk relaxation, it attempts to seek one which improve the runtime performance the most while maintaining the risk below the threshold of tolerance.

current priority level, which in turn activates or deactivates auditing activity appropriately. In addition, if the risk level increases above or decreases below the risk tolerance threshold, the following course of action occurs.

The **RiskManager** maintains two heap data structures. The first one contains all the permissions for which the **ActiveMonitor** has predicates but are currently unused, and **ObjectGroup**s that can currently be transparently accessed through the **CapabilityManager**'s transformations. The objects are stored in the heap using the cost-to-benefit ratios as the keys.

The second heap contains all the permissions for which the **ActiveMonitor** is currently evaluating predicates before it grants permissions, and **ObjectGroup**s that can not be used since the **CapabilityManger** has deleted the relevant keys. The objects in this heap are keyed by the benefit-to-cost ratios. When the risk level rises, the **RiskManager** extracts the minimum value element from the first heap, and inserts it into the second heap. If it was a permission, the corresponding predicate evaluation is activated. If it was an **ObjectGroup**, a call is made to the **CapabilityManager** to disable it. The risk level is updated. If it remains above the risk tolerance threshold the process is repeated until the risk has reduced sufficiently. Similarly, when an event causes the risk to drop, the **RiskManager** extracts the minimum element repeatedly from the second heap, inserting it into the first heap, disabling predicate checks for permissions and activating access to **ObjectGroup**s, while the risk remains below the threshold of tolerance. In this manner, the system is abled to adapt its security posture continuously.

Finally, we note that all discussions of curtailments apply at any data granularity. Thus, the model applies when each o_β is a protection group rather than a file. The implementation operates at object group granularity since this reduces the number of response options. The adverse effect is that each curtailment results in a larger change in risk, potentially causing over-corrections. In practice, this can be minimized by choosing each object group such that its member files' protection needs are correlated.

6.6 Evaluation

The NIST ICAT database [ICAT] contains information on over $6,200$ vulnerabilities in application and operating system software from a range of sources. These are primarily classified into seven categories. Based on the database, we have constructed a suite of attacks, with each attack illustrating the exploitation of a vulnerability from a different category. In each case, the system component which includes the vulnerability is a Java servlet that we have created and installed in the W3C's Jigsaw web server (version 2.2.2) [Jigsaw]. We describe below a scenario that corresponds to each attack, including a description of the vulnerability that it exploits, the intrusion signature used to detect it and the way RheoStat responds. The global risk tolerance threshold is set at 20.

6.6.1 Access Validation Error

An *access validation error* is a fault in the implementation of the access control mechanism. Although the access control has been configured correctly, it can be bypassed. In our example, the servlet implements logic to restrict access to certain documents based on the source IP address. However, if a non-canonical version of the path is used, the servlet fails to implement the restriction on the source IP address. This access control implementation flaw allows the policy to be violated despite a correct configuration.

When the following sequence of events is detected, an attack that exploits this vulnerability is deemed to have occurred. First, the web server accepts a connection to port 8001 (the default port that Jigsaw listens on).

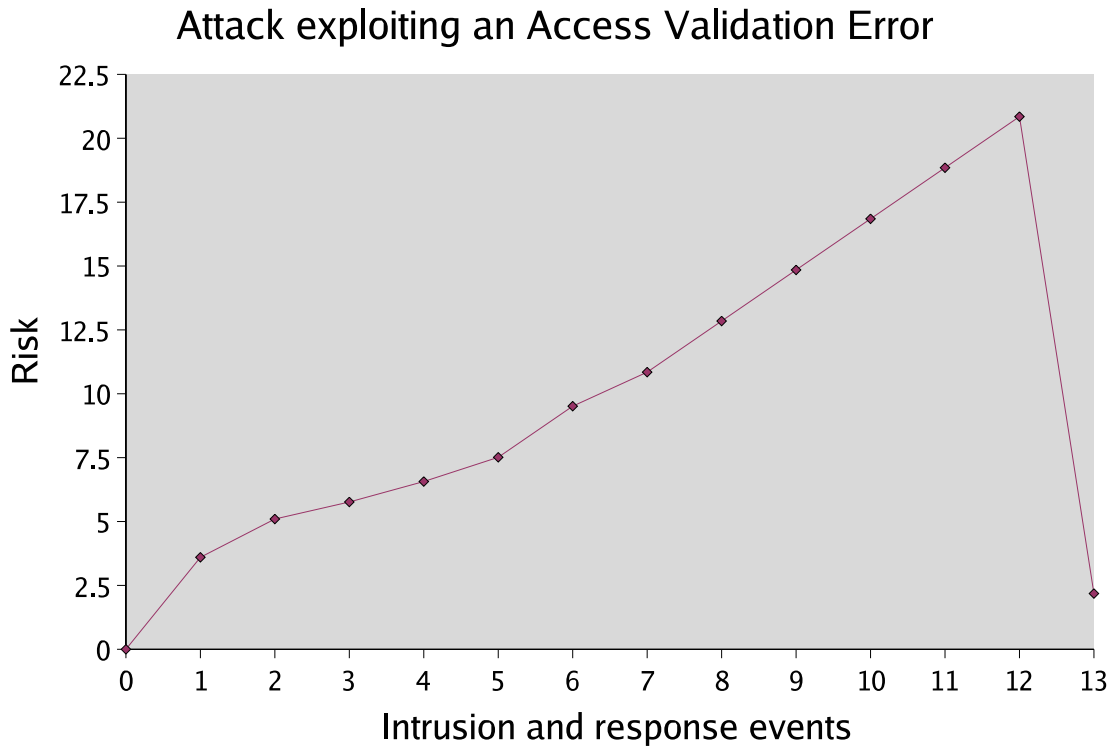

Figure 6.5: Attack exploiting an access validation error.

Second, it serves the specific HTML document which includes the form which must be filled to request a file. Third, the server accepts another connection. Fourth, it executes the servlet that verifies if the file can be served to the client, based on its IP address. Fifth, the decision to deny the request is logged. Sixth, despite the choice to deny the request, the file is served (due to the non-canonical path not being classified correctly). The events must all occur within the pre-match timeout of the signature, which is 1 minute.

In Figure 6.5, event 6 and events $8-12$ correspond to this signature. Events $1-5$ and event 7 are matches of other signatures which cause the global system risk to rise. They occur since the events in this signature overlap with those of other signatures. After event 12, the risk has risen above 20, the threshold of risk tolerance. As a result, the **RiskManager** searches for and finds the risk reduction measure which has the lowest cost-benefit ratio. The measure it selects cryptographically curtails access to the 'Documents' object group which is data that is listed as being affected as a consequence of this attack. This causes the risk to drop in event 13.

6.6.2 Configuration Error

A *configuration error* introduces a vulnerability into the system due to setting that are controlled by the user. Although the configuration is implemented faithfully by the system, it allows the security policy to be subverted. In our example, the servlet authenticates the user before granting access to certain documents. The password used is a permutation of the username. As a result, an attacker can guess the password after a small

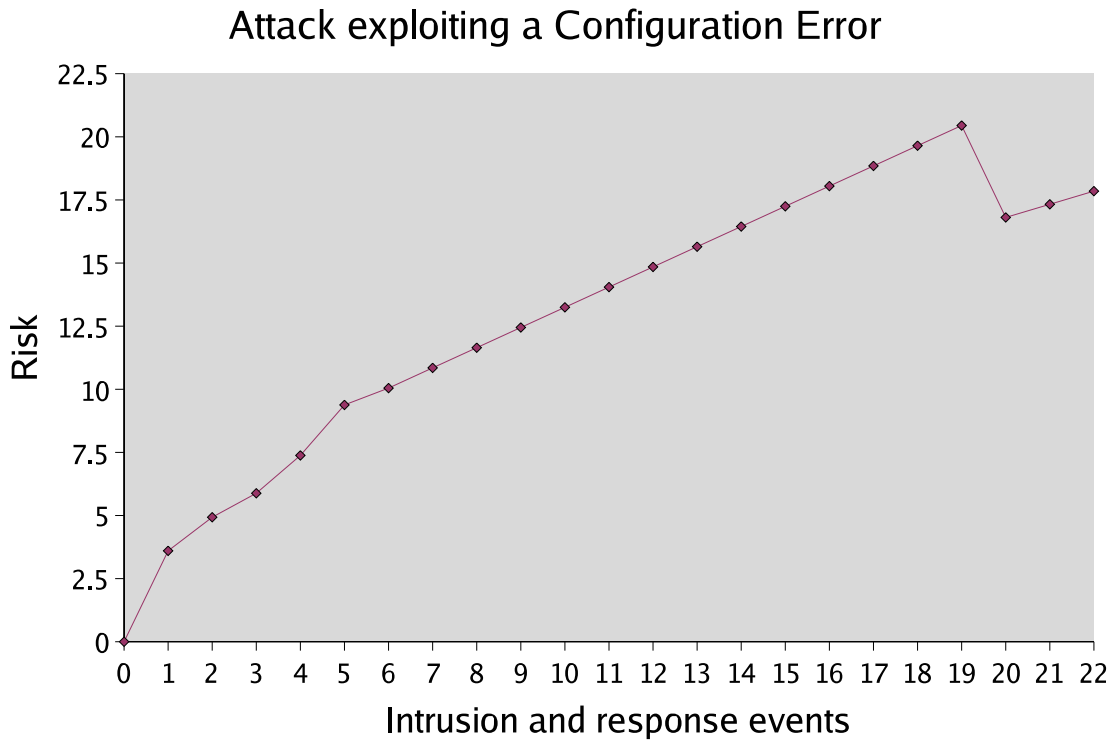

Figure 6.6: Attack exploiting a configuration error.

number of attempts. The flaw here is the weak configuration.

When the following sequence of events is detected, an attack that exploits this vulnerability is deemed to have occurred. First, the web server accepts a connection to port 8001. Second, it serves the specific HTML document which includes the form which requests authentication information as well as the desired document. Third, the server receives another connection. Fourth, the servlet that verifies if the file can be served to the client, based on the authentication information provided, will execute. Fifth, the decision to deny the request is logged. If this sequence of events repeats twice again within the pre-match timeout of the signature, which is 1 minute, an intrusion attempt is deemed to have occurred.

In Figure 6.6, events $7-18$ and $20-22$ correspond to this signature.

Events 1–6 are of other signatures that cause the risk level to rise. Event 18 causes the risk threshold to be crossed. The system responds by enabling a predicate for the permission that controls whether the servlet can be executed. This is event 19 and reduces the risk. The predicate checks whether the current time is within the range of business operating hours. It allows the permission to be granted only if it evaluates to `true`. During operating hours, it is likely that the intrusion will be flagged and seen by an administrator and it is possible that the event sequence occurred accidentally, so the permission continues to be granted. Outside those hours, it is likely that this is an attack attempt and no administrator is present, so the permission is denied thereafter, till the post-match timer expires after one hour and the threat is reset.

6.6.3 Design Error

A *design error* is a flaw that introduces a weakness in the system despite a safe configuration and correct implementation. In our example, the servlet allows a remote node to upload data to the server. The configuration specifies the maximum size file that can be uploaded. The servlet implementation ensures that each file uploaded is limited to the size specified in the configuration. However, the design of the restriction did not account for the fact that repeated uploads can be performed by the same remote node. This effectively allows an attacker to launch the very denial-of-service (that results when the disk is filled) that was being guarded against when the upload file size was limited.

When the following sequence of events is detected, an attack that exploits this vulnerability is deemed to have occurred. First, the web server

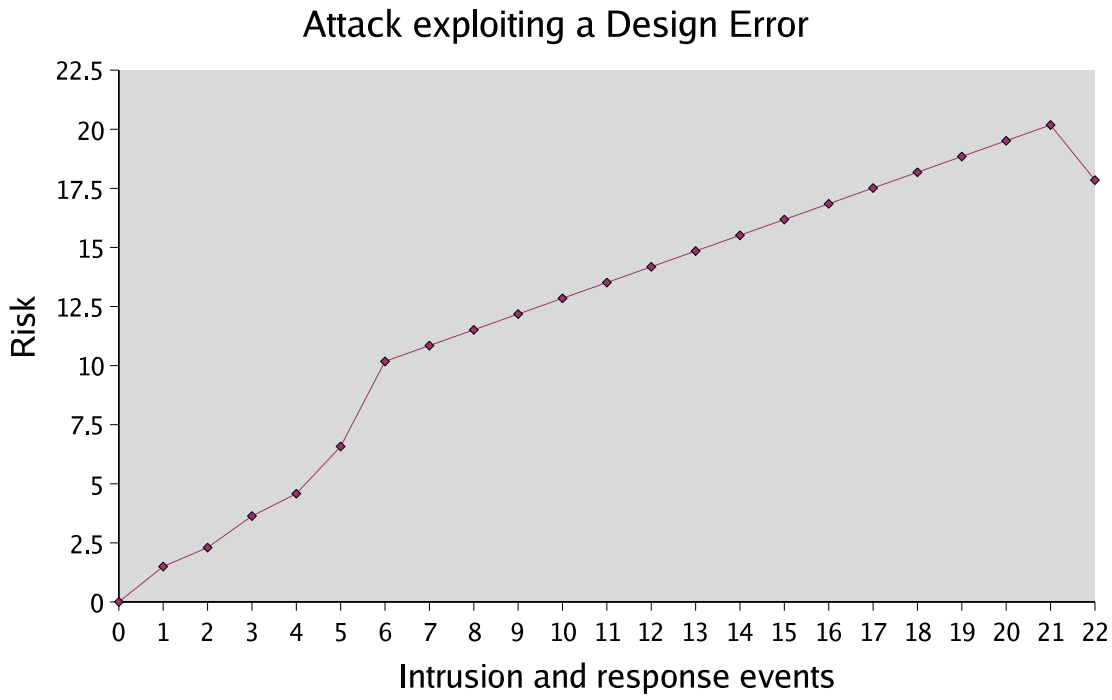

Figure 6.7: Attack exploiting a design error.

accepts a connection to port 8001. Second, it serves the specific HTML document which includes the form that allows uploads. Third, the server receives another connection. Fourth, it executes the servlet that accepts the upload and limits its size. Fifth, a file is written to the uploads directory. If this sequence of events repeats twice again within the pre-match timeout of the signature, which is 1 minute, an intrusion attempt is deemed to have occurred.

In Figure 6.7, events $7-21$ correspond to this signature. Events $1-6$ are of other signatures that cause the risk level to rise. Event 21 causes the risk threshold to be crossed. The system responds by enabling a predicate for the permission that controls whether files can be written to the uploads directory. This is event 22 and reduces the risk. The predicate checks whether the current time is within the range of business operating

hours. It allows the permission to be granted only if it evaluates to `true`. During operating hours, it is likely that the denial-of-service attempt will be flagged and seen by an administrator, so the permission continues to be granted on the assumption that manual response will occur. Outside those hours, it is likely that no administrator is present, so the permission is denied thereafter, till the post-match timer expires after one hour and the threat is reset.

6.6.4 Environment Error

An *environment error* is one where an assumption is made about the run-time environment which does not hold. In our example, the servlet authenticates a user, then stores the user's directory in a cookie that is returned to the client. Subsequent responses utilize the cookie to determine where to serve files from. The flaw here is that the server assumes the environment of the cookie is safe, which it is not since it is exposed to manipulation by the client. An attacker can exploit this by altering the cookie's value to reflect a directory that they should not have access to.

When the following sequence of events is detected, an attack that exploits this vulnerability is deemed to have occurred. First, the web server accepts a connection to port 8001. Second, it serves the specific HTML document which includes the form that authenticates a user. Third, the server receives another connection. Fourth, it executes the servlet that authenticates the user and maps users to the directories that they are allowed to access. It sets a cookie which includes the directory from which files will be retrieved for further requests. Fifth, the server receives another connection. Sixth, it serves the specific HTML document that includes

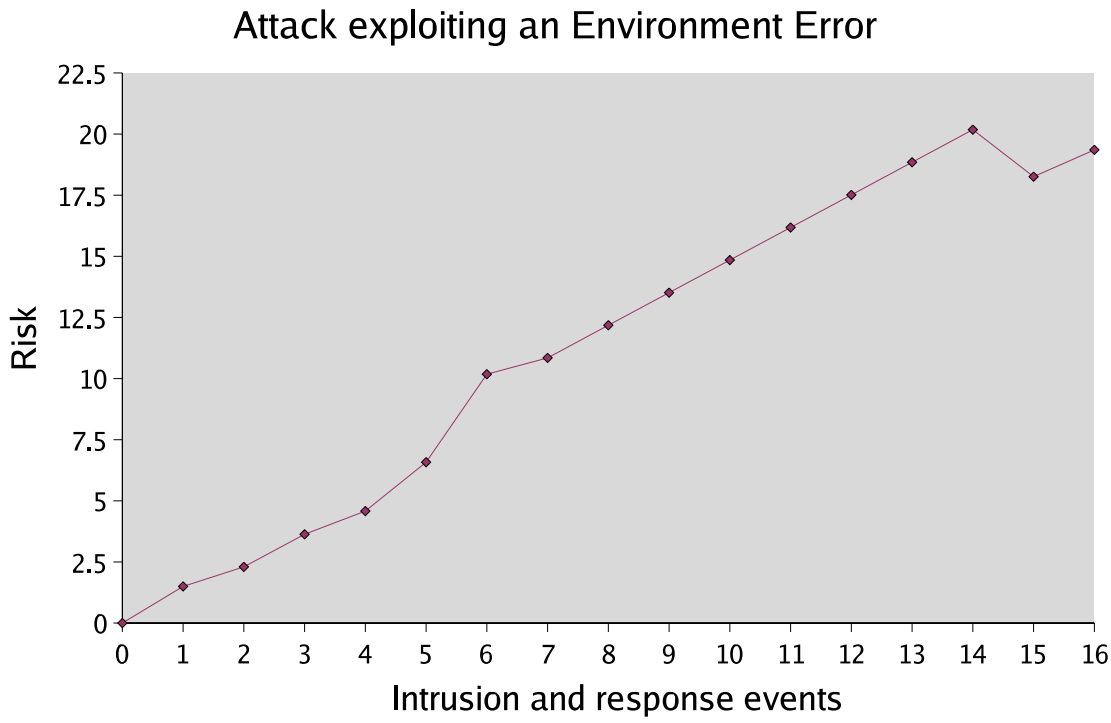

Figure 6.8: Attack exploiting an environment error.

the form which accepts the file request. Seventh, the server receives another connection. Eighth, the servlet that processes the request, based on the form input as well as the cookie data, is executed. Ninth, a file is served from a directory that was not supposed to be accessible to the user. The events must all occur within the pre-match timeout of the signature, which is 1 minute.

In Figure 6.8, event 3, events $8-14$ and event 16 correspond to this signature. Events $1-2$ and $4-7$ are of other signatures. Event 14 causes the risk threshold to be crossed. The system responds by enabling a predicate for the permission that controls whether the file download servlet can be executed. This is event 15 and reduces the risk. The predicate simply denies the permission. As a result, the attack can not complete since no

more files can be downloaded till the safeguard is removed when the risk reduces at a later point in time (when a threat's timer expires).

6.6.5 Exceptional Condition Handling Error

An *exceptional condition handling error* can result when the system is left in an exposed state after an unexpected event occurs. It is due to the failure to explicitly design the system to fall back into a safe state when unplanned eventualities are realized. In our example, when the servlet is authenticating the user, it checks a list of revoked accounts on another site. If does not receive a response after multiple queries, it grants access (on the assumption that there is an error in the revocation server's functioning). The resulting attack is mounted by first flooding the revocation server's network connection so as to assure that it can not respond, then utilizing an expired account to gain access.

When the following sequence of events is detected, an attack that exploits this vulnerability is deemed to have occurred. First, the web server accepts a connection to port 8001. Second, it serves the specific HTML document which includes the form which requests authentication information as well as the desired document. Third, the server receives another connection. Fourth, it executes the servlet that checks the authentication information provided. Next, an attempt is made to contact the revocation server to check that the credentials have not been revoked. Since the revocation server's network connectivity is under attack, the connection to it will timeout. After a total of three attempts, the check will fail, incorrectly allowing access instead of denying it. This results in the fifth, sixth and seventh events being network exceptions, while the eighth is the comple-

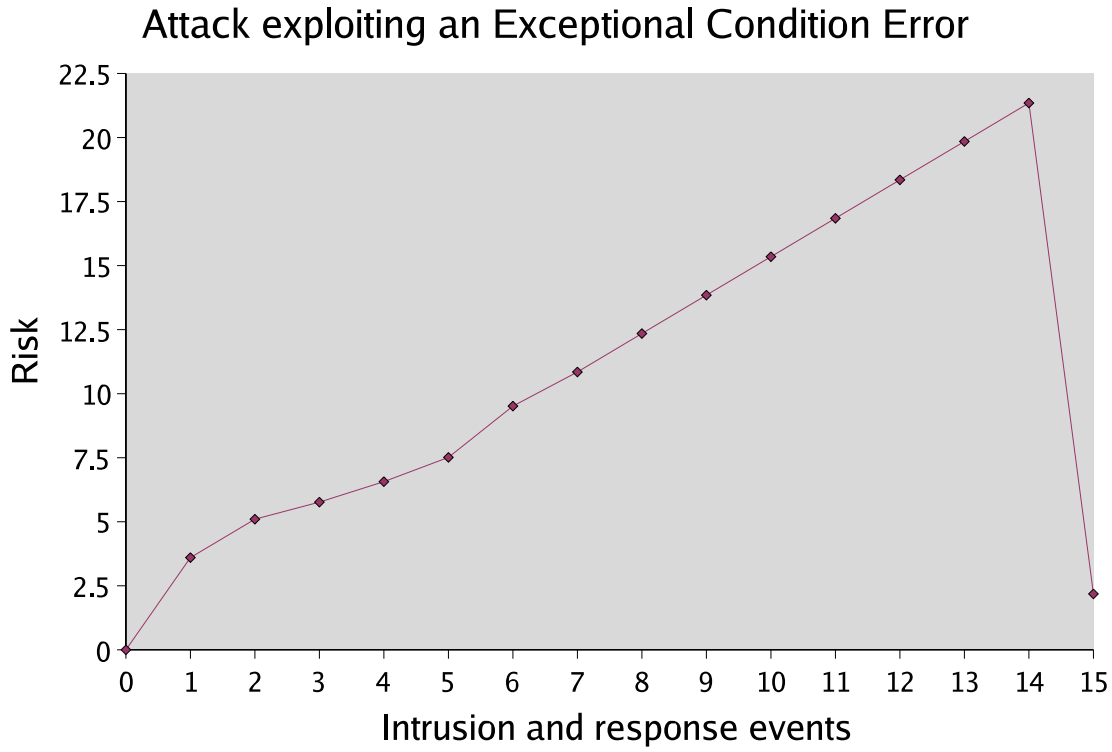

Figure 6.9: Attack exploiting an exceptional condition handling error.

tion of the file request. The events must all occur within the pre-match timeout of the signature, which is 2 minutes.

In Figure 6.9, event 2 and events $8-14$ correspond to this signature. Event 1 and events $3-7$ are of other signatures. Event 14 causes the risk threshold to be crossed. The system searches for a risk reduction measure and opts to cryptographically curtail access to the 'Documents' object group which is data that is listed as being affected as a consequence of this attack. This causes the risk to drop in event 15.

6.6.6 Input Validation Error

An *input validation error* is one that results from the failure to conduct necessary checks on the data. A common example of this type of error is the failure to check that the data passed in is of length no greater than that off the buffer in which it is stored. The result is a buffer overflow which can be exploited in a variety of ways. In our example, the servlet allows a file on the server to be updated remotely. The path of the target file is parsed and a check is performed to verify that it is in a directory that can be updated. The file 'Password.cfg' is used in each directory to describe which users may access it. By uploading a file named 'Password.cfg', an attacker can overwrite and alter the access configuration of the directory. As a result, they can gain unlimited access to the other data in the directory.

When the following sequence of events is detected, an attack that exploits this vulnerability is deemed to have occurred. First, the web server accepts a connection to port 8001. Second, it serves the specific HTML document which includes the form that allows uploads to selected directories. Third, the server receives another connection. Fourth, it executes

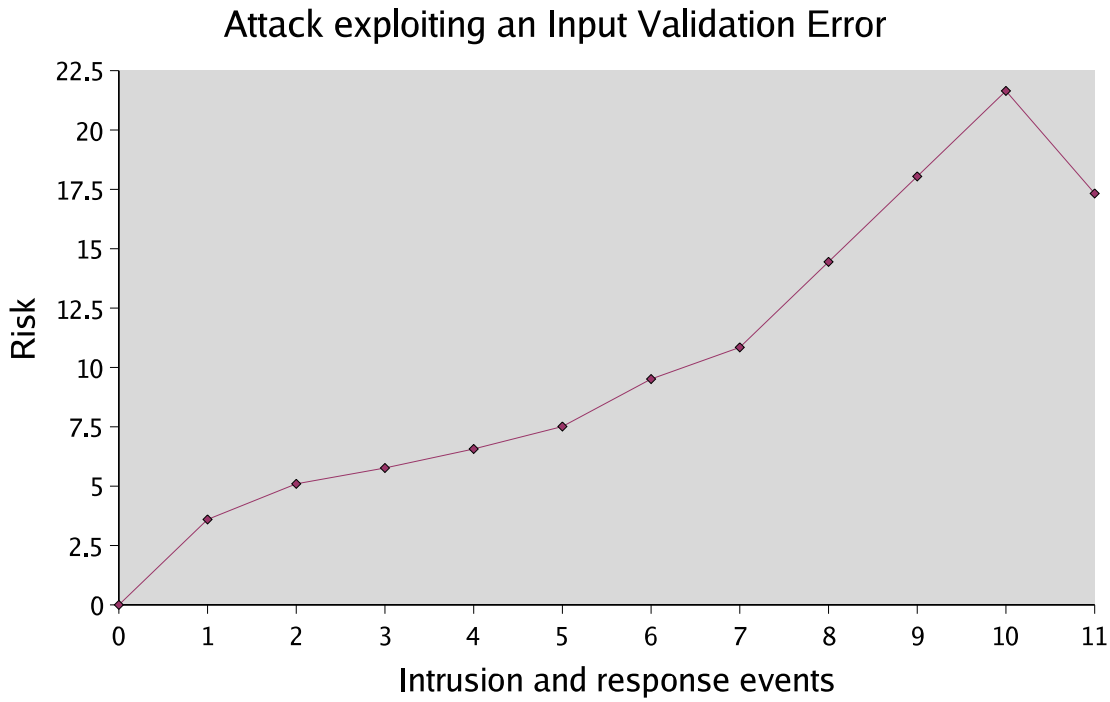

Figure 6.10: Attack exploiting an input validation error.

the servlet that checks that the uploaded file is going to a legal directory. Fifth, the 'Passwords.cfg' file in the uploads directory is written to. The events must all occur within the pre-match timeout of the signature, which is 1 minute.

In Figure 6.10, event 1 and events $7-10$ correspond to this signature. Events $2-6$ are of other signatures. Event 10 causes the risk threshold to be crossed. The system responds by enabling a predicate for the permission that controls write access to the 'Passwords.cfg' file in the uploads directory. This is event 11 and reduces the risk. The predicate simply denies the permission. As a result, the attack can not complete since the last step requires this permission to upload and overwrite the 'Passwords.cfg' file. Enabling this safeguard does not affect legitimate uploads since they do not need to write to this file.

6.6.7 Race Condition Error

A *race condition error* results due to the system performing a security operation in multiple steps, while assuming that the sequence is being performed atomically. In our example, a servlet allows a user to create an account by providing a username and password. The servlet creates a writable copy of the password file in a temporary directory, to which it appends the new account information before removing write permission and moving the file to the password file's usual location. The design assumes the copy, append, change permission and move operations all occur atomically. An attacker uses a file upload servlet running on the same host that has access to the temporary directory, by initiating the creation of a new account while repeatedly uploading a spurious password file to the temporary location. By continuously, repeatedly uploading the file, when the legitimate one appears it is overwritten by the spurious one. This can be used to grant greater privileges than they would have been allowed as a new user having just created an account.

When the following sequence of events is detected, an attack that exploits this vulnerability is deemed to have occurred. First, the web server accepts a connection to port 8001. Second, it serves a specific HTML document that includes a form for checking whether a username already exists in the system. Third, it receives another connection. Fourth, it executes the servlet that checks whether the username is in use. Fifth, the password file is opened for copying to a temporary location. Sixth, a temporary copy is written out. Seventh, the HTML document which includes a form for selecting a username and password is served. Eighth, the server re-

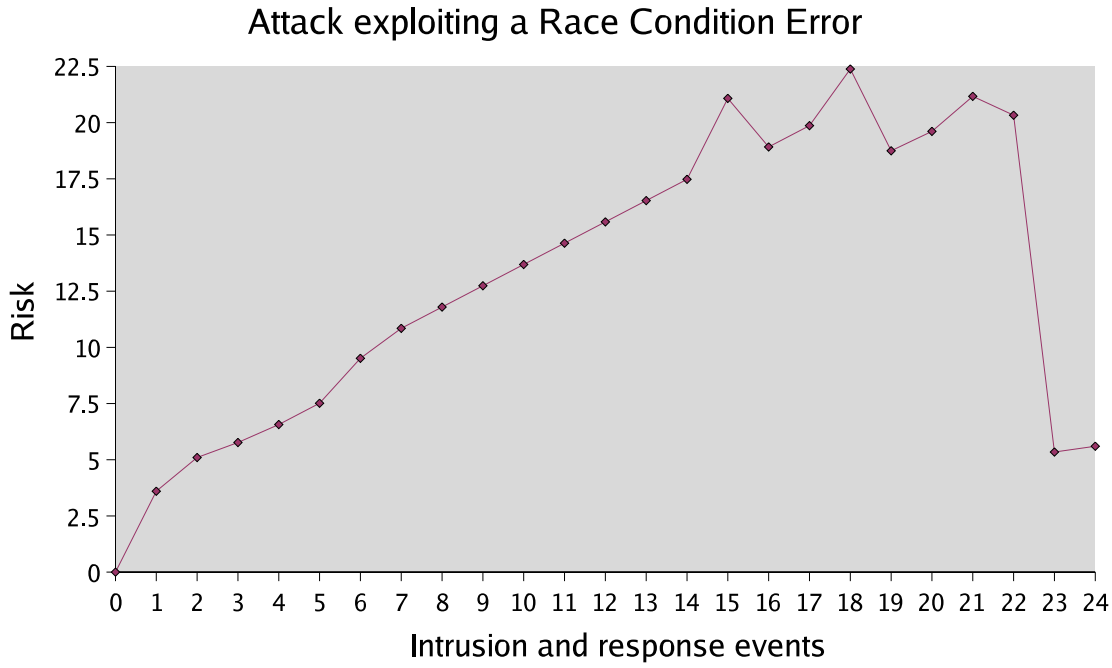

Figure 6.11: Attack exploiting a race condition error.

ceives a connection. Ninth, it serves the HTML document which includes a form for uploading files. Tenth, it receives another connection. Eleventh, it executes the servlet for uploading a file. Twelfth, the temporary password file is overwritten by the upload. Thirteenth, another connection is accepted by the server. Fourteenth, the servlet for creating a new account is executed. Fifteenth, it appends the new account to the temporary password file (which has been subverted at this point if the attack has not been interfered with).

In the Figure 6.11, event 5, events $8-14$, event 17, and event 20, pertain to this signature. Events $1-4$ and $6-7$ relate to other signatures. Event 15 also pertains to another signature but causes the risk to exceed the threshold of tolerance. The system responds in event 16 by activating a predicate to deny the write permission for the password file in the uploads

directory. Event 18 pertains to another signature but causes the risk to exceed the threshold of tolerance. The system responds in event 19 by activating a predicate for the permission which controls whether the file upload servlet can execute. The predicate activated is a Chinese Wall check which will subsequently allow access only to other files in the same group as the servlet. Event 21 pertains to another signature but causes the risk to exceed the threshold of tolerance. The system responds in event 22 by activating a predicate for the write permission of the temporary version of the password file. The predicate activated is a Chinese Wall check which will subsequently allow access only to other files in the same group. Since another group has already been accessed in event 21, the write permission for the temporary version of the password file will subsequently be denied. Since the risk has not reduced below the threshold of tolerance, another risk reduction measure is taken in event 23 in the form of limiting access to the 'Documents' object group. By event 24, where the attack attempts to complete, the response measures in place prevent it from succeeding. In particular, write access has been disabled for the temporary version of the password file in the temporary directory where uploads are allowed, due to event 22.

6.7 Related Work

The field of intrusion detection was created based on the premise that an intruder's behaviour would differ from that of a legitimate user [Anderson80], [Denning87]. Since the systems and their users have complex behaviour, characterizing the difference between intrusive and non-intrusive activity has been a formidable challenge and numerous approaches have been

tried.

Most intrusion detection systems have been designed to focus on recognizing an attack and then raising an alarm, without taking any further action. Such projects are IDES [Lunt88], MIDAS [Sebring88], Haystack [Smaha88], Wisdom and Sense [Vaccaro89], NSM [Heberlein90], DIDS [Snapp92], USTAT [Ilgun95], NIDES [Anderson95], Execution Monitoring [Ko94], IDIOT [Kumar94], JiNao [Jou97], and BRO [Paxon98].

Intrusions can be limited statically through the use of sandboxing techniques which specify constraints that are verified at runtime. Examples of such systems are Java [Java] and Janus [Goldberg96].

Alternatively, the response can be dynamic. Early systems developed limited ad-hoc responses, such as limiting access to a user's home directory or logging the user out [Bauer88], or terminating network connections [Pooch96]. This has also been the approach of recent commercial systems. For example, BlackICE [BlackICE] allows a network connection to be traced, Intruder Alert [Symantec] allows an account to be locked, Net Prowler [Symantec] can update firewall rules, NetRanger [Cisco] can reset TCP connections and RealSecure [ISS] can terminate user processes.

Frameworks have been proposed for adding response capabilities. DCA [Fisch96] introduced a taxonomy for response and a tool to demonstrate the utility of the taxonomy. EMERALD's [Porras97] design allows customized responses to be invoked automatically, but does not define them by default. AAIR [Carver01] describes an expert system for response based on an extended taxonomy.

Risk analysis has been utilized to manage the security of systems for several decades [FIPS31]. However, its use has been limited to offline

risk computation and manual response. [SooHoo02] proposes a general model using decision analysis to estimate computer security risk and automatically update input estimates. [Bilar03] uses reliability modeling to analyze the risk of a distributed system. Risk is calculated as a function of the probability of faults being present in the system's constituent components. Risk management is framed as an integer linear programming problem, aiming to find an alternate system configuration, subject to constraints such as acceptable risk level and maximum cost for reconfiguration.

In contrast to previous approaches, we use the risk computation to drive changes in the operating system's security mechanisms. This allows risk management to occur in real-time and reduces the window of exposure.

6.8 Summary

We have described a formal framework for managing the risk posed to a host. The model calculates the risk based on the threats, exposure to the threats and consequences of the threats. These are estimated using augmentations to an intrusion detector, the reference monitor and the filesystem of a host, respectively. We then described of the design and implementation of a prototype intrusion response engine, RheoStat. Its utility for automated response is illustrated with a set of attack scenarios in which it manages the risk in real-time by dynamically altering the configurations of the ARM reference monitor, the RICE data protection system and the NOSCAM auditing subsystem, and thereby limits the efficacy of

the intrusions.

Chapter 7:
Conclusion

7.1 Lessons Learned

An underlying thread in the subsystems developed was the need for asynchronous primitives in the runtime environment that could be utilized by security applications. Current systems' synchronous approaches suffice when they were used in manual operations but do not scale to automated security reconfiguration. If limited to synchronous primitives, automated solutions must poll or periodically alter values that are of interest, rather than being able to rely on a callback occurring when a value changes. This introduces an artificial tradeoff between accuracy and overhead. Asynchronous primitives address this.

The other issue that repeatedly surfaced was the need for finer grain protection. The choice of granularity is subject to a tradeoff between efficacy and reduced performance. If the grain is too fine, the overhead imposed is severe and makes the protection primitive impractical. If the grain is too coarse, the protection mechanism is liable to allow breaches due to unnecessary exposures. Protection primitives in use owe their design to a period when tolerance for overhead was lower and the need for finer grain security primitives was not acute. The balance has changed on both fronts and finer grain primitives are now needed. The ideal choice would be a method to allow the granularity of protection to be variable, making it possible to completely do away with the tradeoff.

7.2 Future Directions

We constructed a prototype as a means of exploring the abstract model to determine what runtime support would be needed. However, each subsystem constructed would benefit from the extension and tuning that would be needed when empirically derived databases of intrusion detection signatures, permission predicates, lists of compromised files, and sets of auditing commands, are used.

SADDLE could be extended to allow applications to submit entire predicates to the Audit Registry. This would allow the number of callbacks to be further decreased since another level of event filtration would occur before the Audit Consumers are notified.

ARM predicates are activated when the exposure they guard against is relevant to the threat at hand. Each predicate has a relative weight based on the extent to which it reduces exposure. Defining specific predicates and determining their weights remains an open research area.

RICE necessarily operates at file granularity when computing integrity checks and signing deltas and closing files. This does not scale when file size increases. Alternate schemes that provide weaker guarantees but improved performance can be explored. One scheme would be to protect at object granularity in object systems (such as Java). The advantage would be that smaller pieces of data are protected and hence the latency may be more managable. The disadvantage would be that in the presence of many small objects, the overhead for manipulating the associated metadata would become dominant.

In the case of NOSCAM, the frequency with which forensic data ought

to be gathered, the audit level at which a particular type of data ought to be recorded and what classes of data ought to be stored, can all be optimized. NOSCAM uses a scalar to determine threat level. This could be replaced by a vector which separates the threat level into several components so auditing of different classes of activity can be altered independently.

While the underlying model of RheoStat generalizes to other intrusion detection schemes, it may be worth re-examining the models for computing risk that have been used by the information assurance community. Temporal context can be used to dynamically modify weights of permission predicates and protected data. Non-linear predictors for modeling signature matching may improve the correlation between observed values and computed estimates of risk.

7.3 Final Remarks

This thesis leverages the information available in the partial matches of the signatures that form the database of an intrusion detector. It couples each signature with the set of resources that need to be exposed for the attack to succeed and the set of files that will be impacted by the attack. This information is combined to produce a composite threat level. When this exceeds a predefined threshold, the system deduces the permissions that must be constrained and the files that need to be protected to manage the risk. It also pro-actively gathers forensic data to compensate for the risk that is not modeled.

SADDLE's Audit Registry allows application to register for callbacks from the runtime when events of interest occur. RheoStat utilizes this

service to maintain subscriptions for the event horizon of the signature database. When the horizon changes, the change in risk level is computed. If it crosses the acceptable threshold, ARM predicates are activated as needed to reduce exposure. If risk needs to be mitigated further, the RICE attributes for confidentiality, integrity and/or availability of the threatened file groups are changed to limit potential damage. Based on the new risk level, the set of forensic data gathered from the runtime by NOSCAM is enlarged as needed. Similarly, when RheoStat partial signature matches time out and the risk level decreases, the ARM permission constraints, RICE attributes and NOSCAM auditing level, are all made less restrictive. In this manner, the system is able to continuously respond to threats and manage the risk that they pose while minimizing the impact the security measures have on performance and usability.

Appendix A:
Configuration

A.1 NOSCAM Audit Configuration

```
# noscam_audit.cfg
#
# NOSCAM configuration file. Comments must begin with #. Blanks lines are
# ignored. One line should specify the MySQL password. Another line can
# optionally specify the frequency of mirroring. All other lines
# should specify events to be audited.
# ------------------------------------------------------------------------
# Format: password <MySQL NOSCAM Password>
#
# <MySQL NOSCAM Password> should be a string. It will be used to connect
#                         to the database for inserting and selecting
#                         audit data.
#
# Example:
#
# password noscam
# ------------------------------------------------------------------------
# Format: update <Mirror Frequency>
#
# <Mirror Frequency> should be an integer. This is the number of seconds
#                    that pass before new database records are replicated
#                    for mirroring to a safe location/immutable medium.
#
# Example:
#
# update 600
# ------------------------------------------------------------------------
# Format: <Frequency> <Priority> <Immutable> <Category> <Delta> <Command> <Options>
#
# <Frequency> should be a positive integer. This is the number of seconds
#             between invocations of the command.
#
# <Priority> should be an integer. If noscam_audit is running at a level above
#            and including this one, the command will be invoked.
#
# <Immutable> should be either 0 or 1. If it is 1, the records will be
#             periodically committed to an immutable medium.
#
# <Category> should be an alphabetical string. It serves as a classifier.
#            Typical values are "file", "net", "cpu".
#
# <Delta> should be either 0 or 1. If it is 1, the output from an invocation
#         will be compared to that of the initial invocation, and only the
#         difference is stored. This is useful when the output is large but
#         does not vary much.
#
# <Command> should be a string. This is an executable (binary or script) in
#           the runtime path from where noscam_audit is invoked. Its output
#           will be recorded by noscam_audit.
#
# <Options> should be a string. These are the parameters to be passed to
#           to the command.
#
# Example:
#
```

```
# 300 5 1 net 0 netstat --protocol=inet
#
# This causes noscam_audit to run ‘netstat --protocol=inet’ every
# 5 minutes (300 seconds) if the current priority level is 5 or lower.
# It labels the output as being in the class ‘net’ so queries for ‘net’
# activity by noscam_run will show its output. Since <Delta> is 0, the
# output will be stored (as opposed to storing the difference from the
# previous invocation). Periodically the output is committed to the
# immutable format.
# ------------------------------------------------------------------------------
```

Example

```
# ------------------------------------------------------------------------------
# Format: password <MySQL NOSCAM Password>
# Format: update <Mirror Frequency>
# Format: <Frequency> <Priority> <Immutable> <Category> <Delta> <Command> <Options>
# ------------------------------------------------------------------------------

password noscam
update 300

60      5       0       cpu     1       lastcomm
60      7       0       cpu     0       ps      auxw
600     9       0       net     1       last
14400   9       0       file    0       stat    /etc/passwd
600     3       0       file    0       ls      -l /tmp
14400   9       0       file    1       strings /bin/login
120     7       0       net     0       netstat -a -A inet
14400   1       0       file    0       md5sum  /root/.ssh2/authorization
300     7       0       net     0       route
600     5       0       net     0       arp
3600    5       0       cpu     0       dmesg
3600    5       0       cpu     1       ksyms
3600    5       0       cpu     1       sysctl  -a
300     5       0       cpu     1       lsof    -U
600     6       0       file    0       lsof
14400   4       0       hw      1       lspci
14400   4       0       hw      1       lsusb
14400   6       0       hw      1       lsdev
14400   6       0       cpu     1       lsmod
14400   4       0       cpu     1       procinfo
14400   4       0       cpu     1       rpm     -qa
14400   6       0       hw      1       cdrecord        -scanbus
300     6       0       net     1       findsmb
300     6       0       file    1       mount
14400   3       0       net     1       ifconfig
3600    4       0       net     1       iwconfig
300     4       0       file    1       df
3600    6       0       file    1       du      -hs /tmp
900     6       0       net     1       ipchains        -L
14400   4       0       file    0       find    / -name core
```

A.2 ARM Predicates

```
# predicates.cfg
#
# Predicate configuration file. Comments must begin with #. Blanks lines
# are ignored. Each permission which can utilize a predicate to reduce
# exposure must be listed here.
#
#-------------------------------------------------------------------------------
# Format:
#
# Permission: <Permission>
# Target: <Target>
# Action: <Action>
# Predicate: <Predicate>
# Timeout: <Timeout>
# Exposure: <Exposure>
# Frequency: <Frequency>
#
# <Permission> must be the class name of the permission to which the predicate
#              is being associated.
#
# <Target> must be the object for whose access permission is being requested.
#
# <Action> must be the type of access which is being requested.
#
# <Predicate> must be the class name of the predicate which will be evaluated
#             to reduce the exposure if needed when granting this permission.
#
# <Timeout> must be a positive integer. This is the number of milliseconds
#           that the predicate is allowed to evaluate for. After this time,
#           if the predicate has not completed, it is terminated and the
#           permission is denied.
#
# <Exposure> must be a floating point number in the range 0 to 1. If the
#            predicate is evaluated, this is the residual exposure after its
#            evaluation.
#
# <Frequency> is a positive floating point number indicating the relative
#             frequency of the permission in the workload.
#
# Example:
#
# Permission: java.io.FilePermission
# Target: /data/employees/salaries.doc
# Action: read
# Predicate: edu.duke.cs.saphir.examples.OperationalHours
# Timeout: 1000
# Exposure: 0.5
# Frequency: 10
#
# When a request for 'read' access to the file '/data/employees/salaries.doc'
# occurs, the predicate 'edu.duke.cs.saphir.examples.OperationalHours' is
# executed in designated active. If it evaluates 'true', then the static
# permission check can proceed. If it evaluates 'false' or does not
# complete in 1000 milliseconds, then the permission is denied. When the
# Risk Manager chooses a response to a threat, it weighs the fact the
# permission's relative frequency in the workload is 10 and that it reduces
# exposure by 0.5 when deciding whether to activate the use of the predicate.
#-------------------------------------------------------------------------------
```

A.3 RICE Group Manager

```
Usage:
  To add a file to or remove a file from a group:

    java -Dmode=<Mode> -Dpassword=<Password> -Dfile=<Group File Location>
         -Dgroup=<Group Name> edu.duke.cs.saphir.GroupManager
         <File to be added to or removed from group>

         <Mode>=add : Add file to group, <Mode>=remove : Remove file from group

  To list all files in a group:

    java -Dmode=<Mode> -Dpassword=<Password> -Dfile=<Group File Location>
         -Dgroup=<Group Name> edu.duke.cs.saphir.GroupManager

         <Mode>=list : List all files in group <Group Name>

  To list all groups:

    java -Dmode=<Mode> -Dpassword=<Password> -Dfile=<Group File Location>
         edu.duke.cs.saphir.GroupManager

         <Mode>=list : List all groups

  To generate a group file without password protection (for runtime use):

    java -Dmode=<Mode> -Dpassword=<Password> -Dfile=<Group File Location>
         -Doutput=<Runtime File Location> edu.duke.cs.saphir.GroupManager

         <Mode>=output : Write group file without password protection to
                         <Runtime File Location>

  To generate a password protected group file from an unprotected one:

    java -Dmode=<Mode> -Dpassword=<Password> -Dfile=<Group File Location>
         -Dinput=<Unprotected File Location> edu.duke.cs.saphir.GroupManager

         <Mode>=input : Read group file without password protection from
                        <Unprotected File Location>
```

A.4 RICE Group Costs

```
# groups.cfg
#
# The consequence of compromising the security of each ObjectGroup is listed
# here. The cost associated with each group's confidentiality, integrity and
# availability are listed, followed by the relative frequency with which
# members of the group are accessed by a typical workload.
#
#-------------------------------------------------------------------------------
# Format:
#
# <ObjectGroup name>
# <Confidentiality cost> <Integrity cost> <Availability cost> <Workload frequency>
#
# <ObjectGroup name> is the name of the RICE Object Group.
#
# <Confidentiality cost> must be a positive floating point that represents the
#                        cost of a compromise of the confidentiality of the group.
#
# <Integrity cost> must be a positive floating point that represents the cost of
#                  a compromise of the integrity of the group.
#
# <Availability cost> must be a positive floating point that represents the cost
#                     of a compromise of the availability of the group.
#
# <Workload frequency> must be a positive floating point number that specifies
#                      the relative frequency of the object group's members
#                      occurrence in a typical workload.
#
# Example:
#
# system
# 10 100 10 1000
#
# The cost of loss of confidentiality of 'system' files is 10. This is lower than
# the cost of loss of integrity which could allow the security policy to be
# subverted. Since the 'system' files typically do not change much, they are
# likely available in backups and the significance of their availability after
# an attack is therefore lower. Finally, 'system' files are used frequently in
# a typical workload and hence have a high frequency.
#-------------------------------------------------------------------------------
```

A.5 RheoStat Signatures

```
# signatures.cfg
#
# Signature configuration file. Comments must begin with #. Blanks lines
# are ignored. Each signature either corresponds to an intrusion or a privileged
# predicate. The sequence of RheoStat states must be immediately listed after
# the signature name. Each state event is composed of the subject, event type,
# and object.
#
#-------------------------------------------------------------------------------
# Format:
#
# <Signature name>
# <Subject> <Event> <Object>
#
# <Subject> should be the class name that is causing the event to occur. The
#           wildcard * can be used to match all subjects.
#
# <Event> should be an auditable event, that is one for which appropriate
#         instrumentation is present in the runtime such that it is reported to
#         to the Audit Registry.
#
# <Object> should be the target of the event. Its specific form will depend on the
#          type of event. The wildcard * can be used to match all objects.
#
# Example:
#
# A check is done to ensure upload rights for the directory but no check
# is done to ensure that the file being uploaded is not the authentication
# 'Passwords.cfg' file, allowing the attacker to add themselves to the
# users that can read the upload directory.
#
# Input Validation Error
# java.lang.Thread ACCEPT_LOCAL_PORT 8001
# org.w3c.util.CachedThread OPEN_READ /WWW/saphir/LimitedUploadLocations.html
# java.lang.Thread ACCEPT_LOCAL_PORT 8001
# org.w3c.util.CachedThread LOAD LimitedDirectoriesUpload.class
# org.w3c.util.CachedThread OPEN_WRITE /WWW/saphir/uploads/Passwords.cfg
#-------------------------------------------------------------------------------
```

A.6 RheoStat Threat Timeouts

```
# timeouts.cfg
#
# Threat timeout configuration file. Comments must begin with #. Blanks lines
# are ignored. Each signature from signatures.cfg that corresponds to an
# intrusion must have associated with it two time out values whose units are
# seconds.
#
#-------------------------------------------------------------------------------
# Format:
#
# <Signature name>
# <Pre-match Timeout> <Post-match timeout>
#
# <Pre-match timeout> should be a positive integer. This is the time in which a
#                     signature must match completely. If it does not match in
#                     the specified time, it will be reset to a state that is the
#                     equivalent of no matching event having occurred. This value
#                     is typically small.
#
# <Post-match timeout> should be a positive integer. This is the time that will
#                      be allowed to pass after a signature has been matched
#                      completely before the threat is deemed likely to no longer
#                      be present. When this amount of time passes after an
#                      intrusion occurs, the signature will be reset to its
#                      original state, on the assumption that corrective action
#                      would have been taken in the interim. This value is
#                      typically large.
#
# Example:
#
# Access Validation Error
# 60 3600
#
# If the signature 'Access Validation Error' is not completely matched within
# 60 seconds from the occurrence of its first event, it is reset. If it is
# matched, then an hour (3600 seconds) is allowed to pass before its threat
# is reset to 0.
#-------------------------------------------------------------------------------
```

A.7 Risk Manager Threats

```
# threats.cfg
#
# Threat list. Each intrusion signature listed in signatures.cfg can either be
# an intrusion detection signature, a privileged predicate or both. If a
# signature is a corresponds to an intrusion, it must be listed here.
#
#-------------------------------------------------------------------------------
# Format:
#
# <Signature name>
#
# <Signature name> should be one of the signatures listed in signatures.cfg.
#
# Example:
#
# Access Validation Error
#
# The 'Access Validation Error' signature will be monitored as a threat and
# the risk level will be dynamically adjusted according to the extent to which
# it is matched.
#-------------------------------------------------------------------------------
```

A.8 Risk Manager Exposures

```
# exposures.cfg
#
# Exposures configuration file. Comments must begin with #. Blanks lines
# are ignored. Each exposure consists of the threat which it pertains to
# followed by the permission whose granting creates the exposure.
#
#-------------------------------------------------------------------------------
# Format:
#
# Threat: <Threat>
# Permission: <Permission>
# Target: <Target>
# Action: <Action>
#
# <Threat> must be one of the threats listed in threats.cfg.
#
# <Permission> must be the class name of the permission which the threat
#              utilizes in the course of its completion.
#
# <Target> must be the object for whose access permission is requested.
#
# <Action> must be the type of access which is being requested.
#
# Example:
#
# Threat: Access Validation Error
# Permission: java.io.FilePermission
# Target: /data/employees/salaries.doc
# Action: read
#
# The intrusion signature 'Access Validation Error' requires 'read'
# access to the file '/data/employees/salaries.doc' in order for the
# attack to successfully complete.
#-------------------------------------------------------------------------------
```

A.9 Risk Manager Consequences

```
# consequences.cfg
#
# The consequences of successful intrusions is specified by the data that may
# be affected. Each intrusion signature is followed by a sequence of RICE
# Object Group's whose members may be impacted by the attack.
#
#-------------------------------------------------------------------------------
# Format:
#
# <Signature name>
# <ObjectGroup name>
#
# <Signature name> is the name of the intrusion and must be listed in
#                  signatures.cfg.
#
# <ObjectGroup name> is the name of the RICE Object Group that is affected.
#
# Example:
#
# Access Control Validation
# system
#
# If the 'Access Control Validation' intrusion occurs, it can compromise
# the security of the data in the RICE Object Group 'system'.
#-------------------------------------------------------------------------------
```

A.10 Risk Manager Threshold

```
# threshold.cfg
#
# The risk threshold below which the system must remain. If the system’s
# risk rises above it, then the Risk Manager will act to reduce the risk.
#
#-------------------------------------------------------------------------------
# Format:
#
# <Threshold>
#
# <Threshold> must be a floating point number which acts as the risk threshold.
#
# Example:
#
# 1000
#
# If the risk rises over 1000, the system will tighten permissions and reduce
# data access. When the risk drops below this, the previous steps are reversed.
#-------------------------------------------------------------------------------
```

Bibliography

[Anderson80] James P. Anderson, Computer Security Threat Monitoring and Surveillance, Technical Report, Fort Washington, PA (1980).

[Anderson95] D. Anderson, T. Frivold, and A. Valdes, Next-generation intrusion detection expert system (NIDES): A summary. Technical Report SRI-CSL-95-07, Computer Science Laboratory, SRI International, May 1995.

[Anon02] Anonymous, Defeating Forensic Analysis on Unix, Phrack Vol. 59, Nr. 6, July 2002.

[Axelsson99] Stefan Axelsson, The Base-Rate Fallacy and its Implications for the Difficulty of Intrusion Detection, 6th ACM Conference on Computer and Communications Security, 1999.

[Axelsson00] Stefan Axelsson, Intrusion Detection Systems: A Survey and Taxonomy, Technical Report, Chalmers University of Technology Department of Computer Engineering, Number 99-15, March 2000.

[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.

[Barkley94] John Barkley et al, Security in Open Systems, National Institute of Standards Technology Publication 800-7, July 1994.

[Bauer88] D. S. Bauer and M. E. Koblentz, NIDX - A Real-Time Intrusion Detection Expert System, Proc. of USENIX Technical Conference, 1988, pp. 261-273.

[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.

[Bilar03] Daniel Bilar, Quantitative Risk Analysis of Computer Networks, PhD thesis, Dartmouth College, 2003.

[Bishop96] M. Bishop, C. Wee and J. Frank, Goal Oriented Auditing and Logging, IEEE Transactions on Computing Systems, 1996.

[BlackICE] blackice.iss.net

[Blaze93] M. Blaze, A cryptographic file system for UNIX, Proceedings of 1st ACM Conference on Communications and Computing Security, 1993.

[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.

[Bruschi01] Danilo Bruschi and Emilia Rosti, Angel : A tool to disarm computer systems, New Security Paradigms Workshop, 2001.

[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.

[Carver01] Curtis Carver, Adaptive, Agent-based Intrusion Response, PhD thesis, Texas A and M University, 2001.

[Cattaneo01] G. Cattaneo and L. Catuogno and A. Del Sorbo and P. Persiano, The Design and Implementation of a Transparent Cryptographic Filesystem for UNIX, FREENIX, 2001.

[Chapman92] D. Brent Chapman, Network (In)Security Through IP Packet Filtering, Third Usenix Security Symposium, September 14-17, 1992, Baltimore, MD, pp. 63-76.

[Chapman00] Chapman Flack and Mikhail J. Atallah, Better Logging Through Formality: Applying Formal Specification Techniques to Improve Audit Logs and Log Consumers, Recent

Advances in Intrusion Detection, Lecture Notes in Computer Science, Number 1907, pp. 1-16, Springer-Verlag, October 2000.

[Cisco] www.cisco.com

[Clark88] David D. Clark, The design philosophy of the DARPA internet protocols, SIGCOMM, pp. 106-114, 1988.

[Cohen85] Fred Cohen, Computer Viruses, PhD thesis, University of Southern California, 1985.

[Cowan98] C. Cowan, et al, StackGuard : Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, USENIX, 1998.

[CSRMMBW88] Proceedings of the 1st Computer Security Risk Management Model Builders Workshop, Martin Marietta, Denver, Colorado, National Bureau of Standards, May 1988.

[CSRMMBW89] Proceedings of the 2nd Computer Security Risk Management Model Builders Workshop, AIT Corporation, Ottawa, Canada, National Institute of Standards and Technology, June 1989.

[CSRMMBW90] Proceedings of the 3rd International Computer Security Risk Management Model Builders Workshop, Los Alamos National Laboratory, Santa Fe, New Mexico, National Institute of Standards and Technology, August 1990.

[CSRMMBW91] Proceedings of the 4th International Computer Security Risk Management Model Builders Workshop, University of Maryland, College Park, Maryland, National Institute of Standards and Technology, August 1991.

[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.

[Denning87] D. Denning, An Intrusion-Detection Model, IEEE Trans. on Software Eng., February 1987.

[Farmer00] Dan Farmer and Wietse Venema, Forensic Computer Analysis, Dr Dobb's Journal, September 2000.

[FIPS31] Guidelines for Automatic Data Processing Physical Security and Risk Management, National Bureau of Standards, 1974.

[FIPS65] Guidelines for Automatic Data Processing Risk Analysis, National Bureau of Standards, 1979.

[Fisch96] Eric Fisch, Intrusive Damage Control and Assessment Techniques, PhD thesis, Texas A and M University, 1996.

[Fletcher95] Sharon Fletcher et al, Software System Risk Management and Assurance, Proceedings of the New Security Paradigms Workshop, August 1995.

[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.

[Fu99] K. Fu, Group Sharing and Random Access in Cryptographic Storage Filesystems, MIT Master's Thesis, 1999.

[Fu00] K. Fu, M. F. Kaashoek and D. Mazires, Fast and Secure Distributed Read-only Filesystem, Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation, 2000.

[Garey79] M.R. Garey and D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, San Francisco, 1979.

[Gehani02a] Ashish Gehani and Gershon Kedem, SADDLE : An Adaptive Auditing Architecture, 14th Incident Response Conference, 2002.

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

[Goldberg96] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer, A secure environment for untrusted helper applications, Proceedings of the 6th Usenix Security Symposium, San Jose, CA, July 1996.

[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.

[Heberlein90] L.T. Heberlein, G. Dias, K. Levitt, B. Mukherjee, J. Wood, and D. Wolber, A Network Security Monitor, Proceedings of the Symposium on Research in Security and Privacy, p296-304, May 1990.

[Honeynet] project.honeynet.org/faq.html

[Howard97] John Howard, An Analysis Of Security Incidents On The Internet, Carnegie Mellon University, 1997.

[Hughes00] J. Hughes et al, A Universal Access, Smart-Card-Based, Secure File System, 9th USENIX Security Symposium, 2000.

[ICAT] icat.nist.gov

[Ilgun95] Koral Ilgun, Richard A. Kemmerer and Phillip A. Porras, State Transition Analysis: A Rule-Based Intrusion Detection Approach, IEEE Transactions on Software Engineering, 21(3), pp. 181-199, March 1995.

[ISS] www.iss.net

[Java] java.sun.com

[Jigsaw] www.w3.org/Jigsaw

[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.

[Jou97] J. F. Jou, S. F. Wu, F. Gong, W. R. Cleaveland, and C. Sargor, Architecture design of a scalable intrusion detection system for the emerging network infrastructure, Technical report, MCNC, Dep. of Computer Science, North Carolina State University, April 1997.

[Kim94] Gene H. Kim and Eugene H. Spafford, Experiences with Tripwire: Using integrity checkers for intrusion detection, Systems Administration, Networking and Security Conference III, Usenix, April 1994.

[Ko94] Calvin Ko, Karl Levitt, and George Fink, Automated detection of vulnerabilities in privileged programs by execution monitoring. Proceedings of the Tenth Annual Computer Security Applications Conference, December 1994.

[Kumar94] S. Kumar and E. Spafford, A Pattern Matching Model for Misuse Intrusion Detection, Proceedings of the Seventeenth National Computer Security Conference, p11-21, Oct 1994.

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

[Lonvick01] Lonvick, C., The BSD Syslog Protocol, RFC 3164, August 2001.

[Lunt88] T. F. Lunt, R. Jagannathan, R. Lee, S. Listgarten, D. L. Edwards, P. G. Neumann, H. S. Javitz, and A. Valdes, Development and Application of IDES: A Real-Time Intrusion Detection Expert System, Technical Report, SRI International, 1988.

[Manber94] U. Manber and S. Wu, Glimpse: A tool to search through entire file systems, Proceedings of the Winter USENIX Conference, January 1994.

[Mazieres99] D. Mazieres et al, Separating Key Management from Filesystem Security, SOSP, 1999.

[MS99] Encrypting File System for Windows 2000, Microsoft, 1999.

[MS96] Logging with Routing and Remote Access, Microsoft Knowledge Base Article Q161426, Microsoft, December 1996.

[Necula97] George Necula and Peter Lee, Research on proof-carrying code for untrusted-code security, IEEE Proceedings of the Symposium on Security and Privacy, Oakland, CA, 1997.

[NIST800-12] Guidelines for Automatic Data Processing Physical Security and Risk Management, National Institute of Standards and Technology, 1996.

[NIST91] Description of Automated Risk Management Packages That NIST/NCSC Risk Management Research Laboratory Has Examined, National Institute of Standards and Technology, 1991.

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

[Paxon98] Vern Paxon, Bro: A system for detecting network intruders in real-time, Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, 1998.

[Peeters03] Rene Peeters, The maximum edge biclique problem is NP-complete, Discrete Applied Mathematics, Volume 131, Issue 3, p651-654, 2003.

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

[Pooch96] U. Pooch and G. B. White, Cooperating Security Managers: Distributed Intrusion Detection System, Computer and Security, (15)-5, p441-450, September/October 1996.

[Porras92] P.A. Porras, STAT - A state transition analysis tool for intrusion detection, Master's Theisis, University of California

Santa Barbara, June 1992.

[Porras97] P.A. Porras and P.G. Neumann, EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances, Proceedings of the Nineteenth National Computer Security Conference, p353-365, Baltimore, Maryland, 22-25 October 1997.

[Portscan] www.securityfocus.com/templates/article.html?id=126

[RFC2246] T. Dierks et al, TLS Protocol, www.ietf.org/rfc/rfc2246.txt, 1999.

[Roger01] Muriel Roger and Jean Goubault-Larrecq, Log Auditing through Model Checking, Proc. 14th IEEE Computer Security Foundations Workshop, Cape Breton, Nova Scotia, Canada June 2001, pp. 220-236, IEEE Comp. Soc. Press.

[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.

[Schneier99] Bruce Schneier and John Kelsey, Secure Audit Logs to Support Computer Forensics, ACM Transactions on Information and System Security, 2(2), pp. 159-176, May 1999.

[Schwartau99] Winn Schwartau, Time Based Security, Interpact Press, 1999.

[Sebring88] M.M. Sebring, E. Shellhouse, M.E. Hanna, and R.A. Whitehurst, Expert system in intrusion detection: A case study. Eleventh National Computer Security Conference, Baltimore, Maryland, October 1988.

[Sibert88] W. Olin Sibert, Auditing in a Distributed System: Secure SunOS Audit Trails, 11th National Computer Security Conference pp. 81-91 (1988).

[Smaha88] S. E. Smaha, Haystack: An intrusion detection system, Proceedings of the Fourth Aerospace Computer Security Applications Conference, p3744, 1988.

[Snapp92] S. R. Snapp, S. E. Smaha, D. M. Teal, and T. Grance, The DIDS (distributed intrusion detection system) prototype, Proceedings of the Summer 1992 USENIX Conference, p227-233, Berkeley, CA, USA, June 1992.

[SooHoo02] Kevin Soo Hoo, Guidelines for Automatic Data Processing Physical Security and Risk Management, PhD Thesis, Stanford University, 2002.

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

[SPECweb99] www.spec.org/osg/web99/

[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.

[Symantec] www.symantec.com

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

[Vaccaro89] H.S. Vaccaro and G.E. Liepins, Detection of anomalous computer session activity, Proceedings of the IEEE Symposium on Research in Security and Privacy, p280-289, Oakland, CA, May 1989.

[Zadok98] E. Zadok, I. Badulescu and A. Shender, Cryptfs: A Stackable Vnode Level Encryption Filesystem, Columbia University Technical Report CUCS-012-98, 1998.

Biography

Ashish Gehani was born in Bombay, India on 29th August, 1971. He received the Bachelor of Science (Honors) in Mathematics from the University of Chicago. His thesis for the Master of Science in Computer Science from Duke University focused on Video Resolution Enhancement. Other publications include Micro-Flow Bio-Molecular Computation in the *Journal of Biological and Informational Processing Sciences*, DNA Cryptography in Springer's *Natural Computing* book series, Transcoding Characteristics of Web Images in *Multimedia Computing and Networking*, SADDLE : An Adaptive Auditing Architecture in the *14th Incident Response Conference*, and NOSCAM : Sequential System Snapshot Service in the Center for Secure and Distributed Systems' *Computer Forensics Workshop*. He was the recipient of the North Carolina Networking Initiative Fellowship and a Usenix Research Grant.

Index