Augmenting Storage with an Intrusion Response Primitive to Ensure the Security of Critical Data

Ashish Gehani Surendar Chandra
Department of Computer Science and Engineering, University of Notre Dame
384 Fitzpatrick Hall, Notre Dame, IN 46556
{ashish.gehani,surendar}@nd.edu [*]

Gershon Kedem
Department of Computer Science, Duke University
Box 90129, Durham, NC 27708
kedem@cs.duke.edu

ABSTRACT

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. A primary goal of such action is to assure the security of the data stored in the system. If this operation is effected manually, the delay between the alarm and the response may be enough for an intruder to cause significant damage.

The alternative proposed in this paper is to provide a response primitive for intrusion detectors to utilize in automating the response. We describe RICE, a modification to the Java file subsystem that provides such functionality for data that is deemed to be threatened by an attack. If it is activated when an intrusion appears likely to succeed, it guarantees the confidentiality, integrity and availability of the protected data even after a system is compromised.

In particular, RICE allows cryptographic encapsulation of data to be reduced to simple key deletion so that it can be effected rapidly. Further, it uses digitally signed hashes of file deltas to allow untainted data to be distinguished from the rest. Finally, file deltas are replicated at a remote node to ensure that changes made by an attacker can be undone using the remote replicas.

1. INTRODUCTION

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.

[*]This work is supported by the U.S. National Science Foundation (IIS-0515674 and CNS-0447671)

Efforts have been made to utilize information about the attack to take precautionary measures automatically, such as terminating processes or network connections. These 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. This is the focus of this paper. By allowing an intrusion detector to interface directly with the storage system through an exposed programming interface, we show how the cryptographic and replication responses needed to ensure the security of the data can be automated.

Implementing runtime protection of data imposes an overhead, however. If the data's security is not critical, the impact on performance may not warrant the changes, while in certain applications it clearly will matter. For example, in the context of financial systems, a breach of confidentiality, integrity or continued availability of the data after an attack can be catastrophic. In such cases the tradeoff is weighted in favor of instituting additional protective measures.

RICE is implemented as a modification of the Java Runtime Environment. It provides intrusion detection applications with a simple programming interface to cryptographically disable (and re-enable with manual authentication) read access to subsets of the data stored in the filesystem. Using authenticated hashes of changes to files, it allows unauthorized writes to be cryptographically detected. Finally, it utilizes authenticated, encrypted replication of deltas. This allows the changes made by an attacker to be undone.

RICE's efficacy is demonstrated through its use with a simple intrusion detector (modeled after Stat [11]) which leverages its capabilities to automatically limit the consequences of imminent attacks. The overhead imposed by the cryptographic operations is also examined.

2. MOTIVATION

Current intrusion detection applications have limited response options. They typically raise an alarm when an intrusion has been detected. The more advanced ones can tear down network connections and kill processes. They do not have a means of protecting the data on the system. In particular, the response is invoked after an intrusion is detected, at which point it may be too late to make any guarantees about the security of the system.

We instead consider the case where the intrusion detector can take proactive protective action before the intrusion completes. This is when a partial match to an intrusion signature has been detected. In this context, tearing down network connections and killing processes on a host are not viable options since they are not reversible and because there will be high false positive rates even if consservative thresholds are used for what is considered to be a partial match. However, protecting the data (using cryptography) at the expense of its accessibility is a reversible operation. While this does irreversibly alter the execution path of an application (which is unable to access the data that it needs), this can be compensated for by programmatically exposing this information and allowing applications to respond to this exceptional condition.

3. GOALS

We outline below the envisioned goals of a storage system augmentation built for intrusion response. We also describe the approach we take to achieving the goals.

Guarantee Security

Data security has three aspects - confidentiality, integrity and availability. If the intrusion response primitives are utilized as designed, they should be able to ensure that all three aspects of the data are maintained. Since an attacker may achieve complete control of the system, we can not rely on the operating system after an intrusion has occurred. This necessitates the use of cryptography to guarantee confidentiality and to verify integrity. In addition, to ensure the availability of data that may be deleted or otherwise modified, the data must be replicated at a remote node.

Reduce Mean Time To Response

If an intrusion detector is limited to raising an alarm, the data remains exposed to attack for a significant period until a manual response can be invoked. Instead the storage subsystem should allow an intrusion detector to directly interact with it to invoke the requisite response. We achieve this by providing a simple programming interface by which the intrusion detector can request particular groups of data objects to be considered at risk. Thereafter, it becomes the storage system's responsibility. Since this is a completely automated event, the mean time to response can be reduced from minutes or hours to seconds.

Compartmentalize the Impact

Usability of a storage system response primitive can be enhanced significantly by limiting its adverse impact on the parts of the system that are not threatened. To achieve this, data is divided into groups (orthogonal to the access control groups in use) that are likely to be affected by particular intrusions. When protective measures are instituted, they should only affect the groups containing the targets of a current attack. This allows the rest of the system to continue functioning unhindered, improving system usability and reducing its vulnerability to denials of service created by the triggering of the protective measures.

Simplify Recovery

Adding security often reduces convenience. In the current context, once a set of data has been protected, it should no longer be readable, writes to it should not be authenticated and replicas should be detected as untrusted. Once a threat is deemed to have passed, these changes must be undone. There are two aspects to this. The first is what must occur at runtime, which is the removal of the cryptographic protections.

Since the cryptographic keys needed to remove the protections would have been deleted to prevent an attacker from being able to access them, they must be recovered from a keystore. Access to the keystore must be password protected for the same reason and thus necessitates manual authentication by the system's administrator or user (as specified by policy). To minimize the inconvenience this imposes, when a protection group is no longer deemed to be under threat, instead of unprotecting it immediately, it is added to a pool of candidates for unprotection. Only when a file from one of these groups is accessed does the keystore get accessed. At this point all the elements of the pool have their protection removed. This effectively amortizes the authentication over a number of recoveries, diminishing the inconvenience it causes.

Make Undoability Usable

The other aspect of undoing changes is what must be done offline if an intrusion occurred. This is the utilization of the remote replicas to reconstruct the filesystem to a point in time where it had not been subverted. The usability of the recovered state is dependent on the semantics of write operations. If there are no transactional guarantees, files may be left with multiple pieces dating to different times. Such data is likely to be of little utility. The approach we use is to allow recovery to be guaranteed at the granularity of transactions defined by the opening and closing of a file for writing.

4. BACKGROUND

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 [1], Transparent Cryptographic File System [2], and Cryptfs [13] 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 [9] and Secure File System - Read Only [4] use asymmetric ciphers to provide authentication but not confidentiality. Encrypting File System [10] 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 [6] and Cepheus [3] 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 read and write access to files by flooding the server port used to listen for requests for keys.

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.

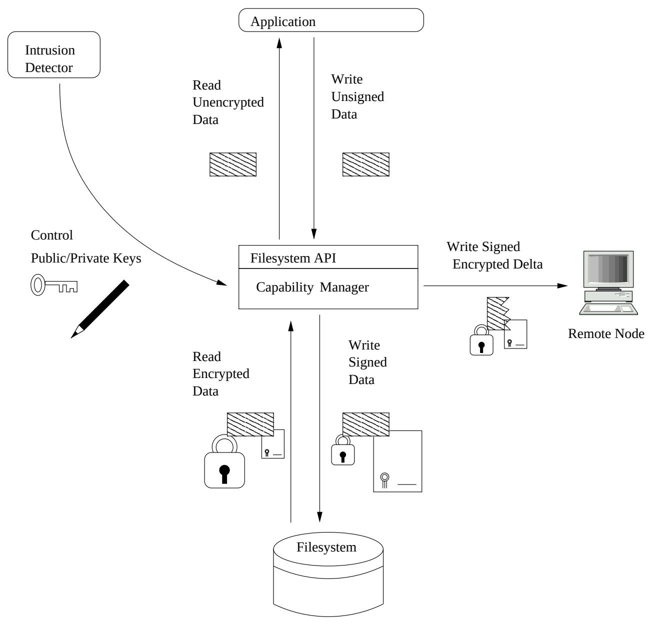

Figure 1: RICE's augmentation of the storage subsystem allows transparent data encryption, integrity hash checking and replication for threatened data. It exposes a programming interface to allow an intrusion detector to activate the protections only when needed to minimize its impact on system usability.

5. OVERVIEW

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 minimize the computation needed in response to a likely intrusion. 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 as described in Section 7.4. When it is no longer used by any application, it is transparently protected once again, as outlined in Section 7.5. An overview of the scheme can be seen in Figure 1.

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, as explained in Section 7.6. Access to the data can be re-enabled by manual authentication, as outlined in Section 7.7. The scheme uses mechanisms similar to a cryptographic filesystem. The differences are highlighted in Section 4.

We describe how RICE is utilized in conjunction with a prototype intrusion response engine in Section 8.1. The overhead it introduces during the normal operation of the system is evaluated in Section 8.2.

6. DESIGN

6.1 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 6.2.

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 allows files that are unlikely to be affected by a threat not be safeguarded. The property also 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.

6.2 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.

6.2.1 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.

6.2.2 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.

6.2.3 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 prior 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.

6.3 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 inter-operable 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.

6.4 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, 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.

6.4.1 Transaction Contract

Once an intrusion has been detected, it is necessary to use the sequence of replicated deltas to undo the changes made by the attacker so as to return the system to an untainted state. If the cryptographic operations (and implicitly the contract of the transaction between the application and the storage subsystem) were at block granularity, then the semantics of the recovered state would be unclear. To see why this is true, consider the following case. Assume that when a block is no longer being written to, the system will re-encrypt it and commit the changes to a remote node. Now consider the implications when an application does a write which spans multiple blocks, some of which have been re-encrypted and replicated at the point in time that a likely intrusion is detected. The response subsystem will delete the relevant keys, making the writes to the remaining blocks unauthenticated. After recovery, the file will contain some blocks containing part of the write operation and some blocks reflecting the earlier state of the file. This leaves the file in an unusable state. It is preferable to be able to ensure that either the entire write can be authentically committed or the file can be reverted to the prior state.

6.4.2 Common Case

In addition to the issue of the semantics, we consider the implication for performance. Reading and writing are far more common operations in a typical workload. As a result, it is reasonable to optimize this case at the expense of the case of opening and closing a file. We therefore opt to encrypt, decrypt and compute hashes at 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).

7. IMPLEMENTATION

We now 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.

7.1 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.

7.2 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.

7.2.1 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.

7.2.2 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.

7.2.3 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.

7.3 Capability Manager

7.3.1 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 manipulation of files using 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.

7.3.2 Initialization and Committal

The **CapabilityManager** takes two parameters. The first is the capabilities file referred to in Section 7.2. 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.

7.4 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).

7.5 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.

7.6 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).

7.7 Re-enabling Access

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 manually at the console. 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 minimizes the negative impact on usability.

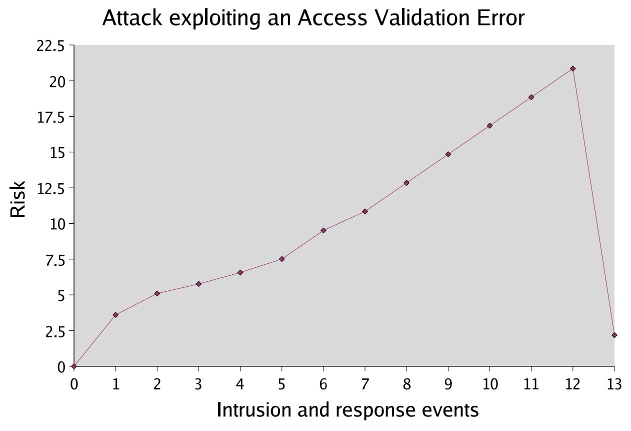

Figure 2: Attack exploiting an access validation error.

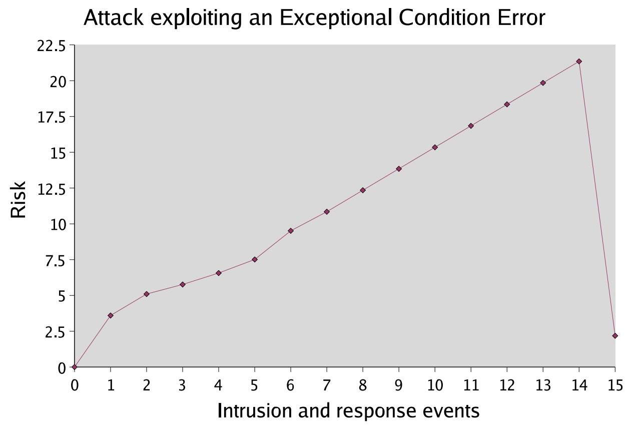

Figure 3: Attack exploiting an exceptional condition handling error.

8. EVALUATION

8.1 Security Benefits

RheoStat [5] is a prototype detection and response engine. It uses a formal risk framework implemented in the module **RiskManager** to effect automated response on a host. Its model calculates the risk based on the threats, exposure to the threats and consequences of the threats. Threat levels are estimated using information about the extent to which intrusion signatures have been matched. System vulnerability is calculated based on the exposure allowed by the system's current access control configuration.

RheoStat can manage the risk by reconfiguring the access control configuration. In the following experiments, it is modified to manage the risk by invoking RICE's protective measures. When it deems a group of files likely to be affected by an intrusion, it invokes RICE's *disable()* command for the relevant group. As can be seen from the following experiments, the response is rapid, needing only a few system events for the data protection to be effected.

The NIST ICAT database [7] 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 three attacks, with each one 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) [8]. 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.

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). 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 2, 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 is invoke RICE's *disable()* operation on 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. The confidentiality of the files in the 'Documents' group is maintained.

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

In Figure 3, 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 use RICE to *disable()* 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. Although the attack itself will succeed and the intruder will gain access to the system, all the data in the 'Documents' group will now be secured by RICE. It will not be possible to decrypt any of the files and any changes will not be authenticated.

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 receives 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 4, 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

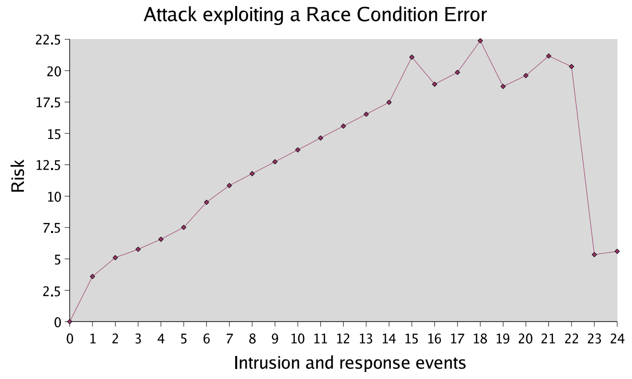

Figure 4: Attack exploiting a race condition error.

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 invoking RICE's *disable()* operation on 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. Using RICE, the integrity of the password file is thus maintained.

8.2 Runtime Overhead

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 microbenchmarks 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

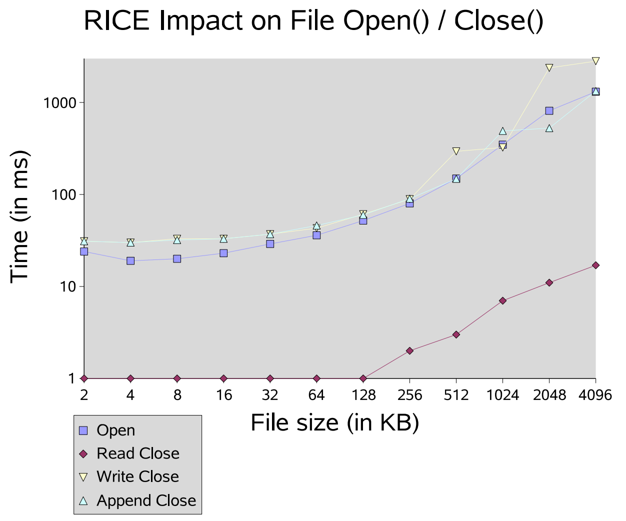

Figure 5: Cost of opening and closing a RICE protected file, measured as a function of file size. The cost of closing depends on whether the file was opened for reading, writing or appending.

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 5.

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 [12] 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 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 6.

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* performs 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

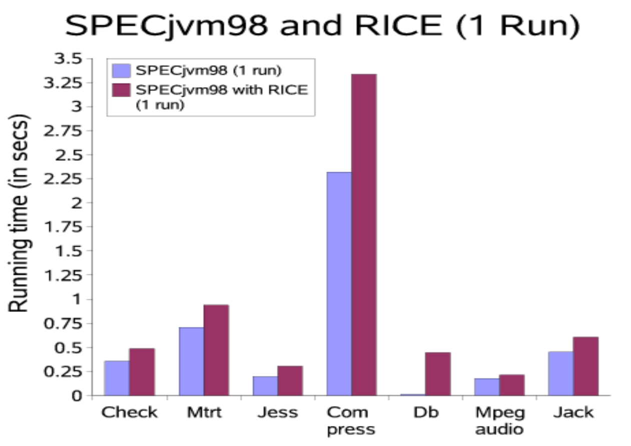

Figure 6: RICE imposes a noticeable impact on applications in SPECjvm98 that rely heavily on the filesystem if limited to a single run.

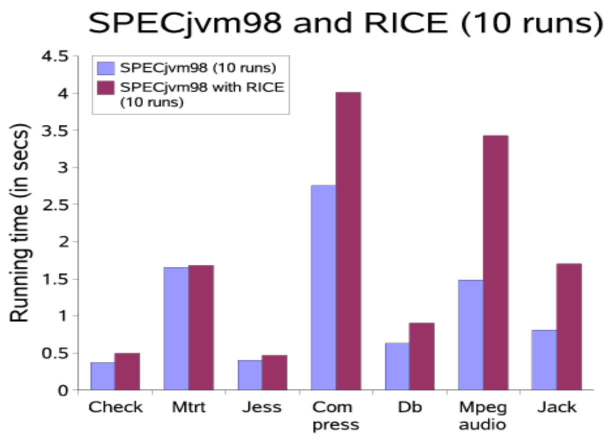

Figure 7: RICE's overhead is noticeably diminished when SPECjvm98 runs 10 times, due to the benefits of caching.

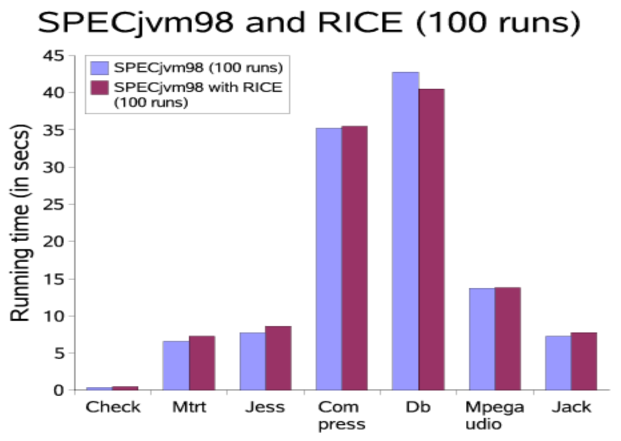

Figure 8: When SPECjvm98 runs a 100 times, the caching benefits compensate for the initial and final cryptographic operations to the point where the impact of using RICE is no longer significant.

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 7. 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 8. 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 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 will address the issue.

8.3 Caveat

As with all automated systems, it is possible for an attacker to utilize knowledge of the response behavior of the system to bypass the protective measures. For example, in the case of intrusion response engines which rely on anomaly detection, the attacker could perform their invasive steps stealthily enough that they fall below the threshold of what is considered intrusive. In such cases, RICE would not be invoked and its protections would not be active when the system's security is subverted.

Of greater concern is the possibility that an attacker may utilize the protection measures to launch denial-of-service attacks against legitimate users of the system. The attacker could effect this by attempting to learn the contents of the intrusion signature database, then launching partial attacks that cause the system to invoke the protective measures of RICE. This would cause the data to be made inaccessible, an effective denial-of-service to running applications. One method to guard against this is to use timers on the detection signatures, so that attacker is forced to be relatively aggressive in launching the attack, in which case an administrator is likely to be alerted in time to intercede. Such protections, however, are beyond the scope of this paper.

9. CONCLUSION

RICE provides a means to augment the Java Runtime Environment to provide data security guarantees when invoked by an intrusion detecor. Using RICE, precautionary measures can be added to mediate file access 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 limited rapidly by deletion of cryptographic keys. File modifications result in deltas that are replicated to a safe node, thereby guaranteeing availability even after a penetration occurs. When the security of the data being protected with RICE is critical, the benefit of the assurances will outweigh the performance impact (which can be ameliorated using cryptographic hardware acceleration).

10. REFERENCES

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

[2] 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.

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

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

[5] A. Gehani and G. Kedem, RheoStat : Real-time Risk Management, Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection, 2004.

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

[7] http://icat.nist.gov

[8] http://www.w3.org/Jigsaw

[9] D. Mazieres et al, Separating Key Management from Filesystem Security, 17th Symposium on Operating Systems Principles, 1999.

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

[11] P.A. Porras, STAT - A state transition analysis tool for intrusion detection, Master's Theisis, University of California Santa Barbara, June 1992.

[12] http://www.specbench.org/osg/jvm98/

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