Attestation-based Policy Enforcement for Remote Access

Reiner Sailer, Trent Jaeger, Xiaolan Zhang, Leendert van Doorn
IBM T.J. Watson Research Center
Hawthorne, New York, 10532
{sailer, jaegert, cxzhang, leendert}@us.ibm.com

Abstract

Intranet access has become an essential function for corporate users. At the same time, corporation's security administrators have little ability to control access to corporate data once it is released to remote clients. At present, no confidentiality or integrity guarantees about the remote access clients are made, so it is possible that an attacker may have compromised a client process and is now downloading or modifying corporate data. Even though we have corporate-wide access control over remote users, the access control approach is currently insufficient to stop these malicious processes. We have designed and implemented a novel system that empowers corporations to verify client integrity properties and establish trust upon the client policy enforcement before allowing clients (remote) access to corporate Intranet services. Client integrity is measured using a Trusted Platform Module (TPM), a new security technology that is becoming broadly available on client systems, and our system uses these measurements for access policy decisions enforced upon the client's processes. We have implemented a Linux 2.6 prototype system that utilizes the TPM measurement and attestation, existing Linux network control (Netfilter), and existing corporate policy management tools in the Tivoli Access Manager to control remote client access to corporate data. This prototype illustrates that our solution integrates seamlessly into scalable corporate policy management and introduces only a minor performance overhead.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Security, Measurement, Management

Keywords

Security Management, Trusted Computing, Remote Access

Acknowledgments: The authors thank the anonymous reviewers and Elisa Bertino for their valuable comments.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'04, October 25-29, 2004, Washington, DC, USA.
Copyright 2004 ACM 1-58113-961-6/04/0010 ...$5.00.

1. INTRODUCTION

Remote access to corporate Intranets is now an essential aspect of the corporate work environment. Client systems are often used by employees both inside the corporate Intranet and remotely from home. Since the remote client systems that access the corporate Intranet are largely administered by their users, this presents corporations with the problem of controlling access to their information. Such remote client systems can download confidential corporate information, so leakage of this information is a problem. Further, remote client systems can modify sensitive corporate data, so the integrity of the modifications are also an issue. In this paper, we present a novel system that enables corporate data servers to manage confidentiality and integrity requirements on accesses by remote client systems.

Figure 1 shows the generic systems model: AliceCorp is the corporation that offers remote access to its employees. Bob is one such employee who accesses the Intranet with his client system. The remote access client program on Bob's client enables access to the corporate Intranet. AliceCorp requires that the corporate data remains protected when offering the remote access service, i.e., that remote requests originating from the client do not compromise the integrity of the corporate Intranet and that confidential contents of responses to the client do not leak through Bob's client. However, a client's ability to ensure enforcement of the corporate policy is rudimentary at best today.

Today, most clients are mainly under control of their users and the client configuration and software can differ considerably from what the corporation would prefer. A client machine that was originally configured by the corporate administrators may no longer meet the original security requirements, such that several security vulnerabilities may be present. Consequently, clients have been conveniently considered insecure and untrusted because protecting them was either too expensive or too restrictive. As a result, companies used appearance factors, such as operating system type and patch level, the version of the anti-virus data base, or active/non-active system passwords to reason about the client's trustworthiness.

Specifically, this paper examines two kinds of attacks (illustrated in Figure 1) that cannot be countered by existing remote access control mechanisms because of their lack of reliable client security guarantees:

Firewall-Bypassing: External attackers can gain unauthorized access to the corporate Intranet. Most corporations today, having invested into security for firewall and intrusion detection systems, have raised the bar for external attackers

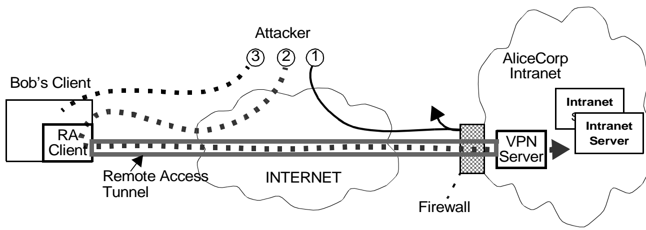

Figure 1: External Threats Based on Remote Access

(Figure 1, attack 1) considerably. Therefore, attackers find it easier to compromise the weaker client systems to hop indirectly into the Intranet. By using the external interfaces of Bob's client, attackers can gain access to Bob's remote access tunnel and thus bypass the corporate firewall (see Figure 1, attack 2). This is a real-time attack that is very difficult to discover from within the corporation because the attacker's actions are covered by the user's actions. With many operating systems offering remote sessions, it cannot be determined whether a service request originating from a client is actually initiated by the client's local user or an external attacker that has gained unauthorized access to the client remotely or locally through a Trojan Horse.

Information-Leaking: Confidential data can also leak indirectly through the client (Figure 1, attack 3). When data is sent to a remote client, control over these data items is transferred to the user's client and to the user. Users may allow other users, knowingly or by mistake, to download such data in peer-to-peer sessions or to copy data onto removable storage devices and distribute them afterwards; manipulated programs (Trojan Horses) on the client can gather and leak information as well.

Current research addresses some of the problems inherent in preventing these attacks, but significant issues remain. First, efforts are underway to leverage trusted hardware, such as the TCG trusted platform module (TPM) [1], to measure system integrity, but determining practical integrity levels and using them to make access control decisions remain open questions. Second, classical integrity policies, such as Biba and Low Water Mark (LOMAC) [2], exist, but their application to remote clients may be impractical. For example, some degree of dependence on lower integrity data may be permissible. Lastly, while the basic notions of reference monitors are well-understood [3] and distributed policy enforcement systems exist [4], preventing these attacks requires greater dependence on the abilities of the remote client than usual. For example, control of information leakage requires that the remote client be able to control itself network interfaces according to a corporate policy. Issues include determining which components can be trusted to enforce policies, determining the kinds of policies that can be enforced, and identifying practical limitations that can be made to enable more effective enforcement.

This paper presents an approach that can be used in the construction of VPN connections to control access to corporate data access and use by remote clients. First, we leverage the TCG/TPM to measure the integrity of remote client systems. We build this on an existing *integrity measurement system* that runs on Linux and measures all executable code, including libraries and kernel modules, that are loaded onto a Linux system [5]. Next, we define an *integrity policy model* that associates semantics with the integrity measurements of remote clients and enables expression of policies that leverage subject identity and integrity in access control decisions. We utilize the Tivoli Access Manager (TAM) system [6] to represent and distribute our integrity policies to remote corporate clients. Lastly, we implement a *personal firewall* on remote clients running Linux that enforces confidentiality policies. Because we can measure the integrity of the remote system reliably using the TPM, the integrity of the remote client can be accurately determined and unauthorized access via the firewall can be prevented. Further, we can verify the integrity of our enforcement mechanism and manage possible leakage paths, so enforcement of confidentiality requirements on corporate information is possible. We have implemented a working prototype on Linux 2.6 and our measurements show that the performance impact of this approach is minor and accounts to about 4% on TCP traffic.

In Section 2, we detail the individual problems that underlie enforcement of access control on remote clients and outline the goals of this research. After describing related work in Section 3, we outline our approach to measure and control remote client's access to corporate data in Section 4. Section 5 details the implementation. Section 6 contains the evaluation of the approach and its implementation. We conclude the paper and discuss future directions in Section 7.

2. BACKGROUND

We aim to provide a remote access policy enforcement architecture that can protect against *firewall-bypassing* to gain unauthorized access (whose prevention we refer to as Security Goal SG1) and *information-leaking* of confidential data (Security Goal SG2) defined in the Introduction.

2.1 Measurement

The first problem is to determine whether the remote client is running authorized code when it is connected to the corporate Intranet. Since any memory of a process can become executable, this is a difficult problem in general. However, integrity measurement approaches are emerging that can be leveraged to narrow the problem.

Recent advances in hardware enable better integrity measurement on client systems. Many clients are now sold with TCG Trusted Platform Module (TPM) [1] chips included, and the low cost of such chips indicates that broad application is possible. The TPM has a set of registers that it protects from the client and it provides two operations on each register content: *extend* and *quote*. The *extend* operation takes a value as input and computes the SHA-1 hash [7] of the current register content and that value. This enables the user of the TPM to build a hash chain. The envisioned use of such a hash chain is to measure a predefined sequence of code loads, such as for *authenticated boot* of an operating system from the system's BIOS and boot-loader. The *quote* operation results in the TPM generating a signed message, using a key protected by the TPM, of the target register's contents. This message can be used to send an authenticated hash chain to a remote party which in turn may validate the integrity of the code contained in the hash chain that belongs to the client. The TPM has other functions, such as random number generation and sealed storage, but these are not relevant to our discussion.

Recent work uses the TPM to measure the integrity of

the application level on Linux systems [5]. Building on the integrity verification of the Linux operating system using the technique described above, we enable Linux to use the TPM to store measurements of the code it loads, including kernel modules, applications, and their shared libraries. The TPM stores a value representing a hash chain of loaded code while the extended Linux kernel stores the actual log of hashes. Since the code is measured and the TPM is extended prior to execution, remote parties can verify the integrity of a client system by retrieving the Linux hash log and the quoted (i.e., signed) hash chain value from the TPM.

With respect to our case, the corporate VPN server can validate the log for a client system before permitting the connection to be opened or permitting the client to download confidential corporate information. Further, the corporate VPN server can even use integrity measurement to determine if a client system is capable of enforcing policies that can control information leakage. The research question is whether such actions can be made practical for a remote access environment. We discuss the issues of using integrity measurement to estimate system integrity and ensure effective policy enforcement below.

2.2 Security Policy

Given the integrity measurements of the previous section, we want to be able to determine the likelihood that the employee, rather than an attacker, is accessing the corporate Intranet. Measurements identify the code that has been loaded by the client system at the time of measurement. Typically, this is used to identify vulnerable code. Since vulnerable code can be compromised by an attacker, the likelihood that an attacker will compromise such programs to masquerade as the employee is high (assumed to be 1).

However, the situation is not always so clear-cut, as code may be protected by limiting the interfaces to it and the inputs it receives. First, some code may have known vulnerabilities that may be leveraged only through network interfaces. If access to these interfaces is limited to trusted parties (i.e., trusted as much as the corporate Intranet), then the use of such code may be permissible. Second, some programs, such as Microsoft Word, may execute macros that can permit a document writer to masquerade as the user. In such cases, the source of the inputs may determine whether an attacker may be in control of the client's accesses to the corporate Intranet.

In traditional integrity models, such as Biba [8] and LOMAC [2], integrity of a subject is based on its dependency on other subjects. For example, a LOMAC policy requires that the integrity level of the subject be equal to the minimum integrity level of the objects that it has read or executed (where integrity level is inversely proportional to likelihood of compromise or vulnerability). Unfortunately, integrity measurement does not provide a complete picture of dependency. We measure the code that is loaded, not the information flows. However, a combination of knowledge about the code and the possible information flows allowed for this code may provide a sufficient model for reasoning about security decisions. We discuss the combination of policy enforcement with code measurement in the next section, but briefly discuss policy modeling below.

In order to reason about the likelihood that the subject making a request is the employee for whom the VPN connection was made, we must not only estimate this likelihood, but the policy model must also be able to express it. Most systems either associate policy with a user or a sensitivity level (e.g., for confidentiality or integrity). In this case, we have a combination of both the employee and an estimate of the integrity level of the client system. A research question is whether current policy models are capable of expressing such policies and can enforce them effectively.

2.3 Enforcement

Prevention of particular information flows may enable us to use code that has some potential for misuse in a high integrity manner. The problem is to determine what forms of control are useful and identify the necessary enforcement mechanisms. E.g., we find that it is appropriate to use confidential data in processes that may not write the data to the client's file system (i.e., all persistent writes go to the corporate Intranet). The challenge is to identify such system restrictions that enable useful processing and determine how these can be measured, so that the corporate servers can verify that their policies can be correctly enforced.

The types of enforcement options that we consider include access to network connections, file systems, and other objects managed by the operation system. Thus, it is necessary for the client operating system to provide the necessary controls over these objects. E.g., Linux provides the NetFilter interface that enables fine-grained control of network communication within the operating system. The Linux Security Modules framework can be used to control access to file systems and other system objects. E.g., an instance of a word processor could be started that can only communicate with the corporate Intranet or a RAM file system.

Measurement is not only the basis for integrity, but also the basis for enforcement. If we load a kernel on the client system that implements the enforcement properties that we desire, then the corporate server can use the measurement of this kernel to verify that the expected enforcement will be performed.

2.4 Experiment

The research problem is to find whether an acceptable solution exists for the problems outlined above. To recap, we envision that secure client access to a corporate Intranet requires that the following problems be solved: (1) determine the integrity level of the client system based on the code it is running; (2) determine whether to trust this client to enforce information flow controls necessary to make such integrity assumptions about client; (3) determine whether additional security properties, such as confidentiality, need to be enforced by the client and whether the kernel supports these; (4) integrate the integrity of the client into the remote access control policies governing the client's access to the corporate servers; (5) enforce this policy on remote access clients and VPN server; and (6) track changes in the remote client's security properties (i.e., sense relevant changes in the client's software stack) and implement resultant policy changes.

3. RELATED WORK

We examine related work in the areas of integrity measurement, security policies for integrity management, and policy enforcement on remote clients.

System Integrity Measurement: Verifying the integrity of a client's software stack isn't a new problem; practical solutions have only appeared recently though. Arbaugh et.

al. [9] describe an architecture to securely boot operating systems in such a way that only trusted systems will be booted in all cases. *Outgoing authentication* [10] enables attesting the software stacks of cryptographic co-processors [11, 12]. Both approaches are too restrictive for client environments because they require only completely trusted configurations boot in the former case or are ultimately implemented as single application systems in the latter case.

Some subsequent research focused on using additional hardware to assess the software stack integrity. *Independent auditors* are explored by Hollingworth et. al. [13], Dyer et. al. [14] and Molina et. al. [15]. Hollingworth suggests using the second CPU on a dual-processor board to provide autonomous monitoring and control of the operating system. Dyer et. al. apply secure co-processors from which the client system and its network access can be both protected and monitored. Zhang et. al. [16] extended these ideas to monitor the integrity of a client kernel by examining kernel data structures from a secure co-processor. Molina et. al. explore a co-processor as an independent auditor that supervises the integrity of the host operating system.

More recently, research has focused on measuring the integrity of systems in a secure manner and enabling verification by remote parties, called *authenticated boot*. All these architectures envision leveraging the TCG Trusted Platform Module (TPM) [1] for securely storing measurements. The NGSCB approach [17, 18] also depends on special hardware to separate a trusted system partition from the standard operating system. Terra [19] is a trusted computing architecture that is built upon a trusted virtual machine monitor that –among other things– authenticates the software running in a VM for remote parties. However, the VMM must be trusted and deriving security properties from the footprint of VM partitions appears difficult. All of these solutions would be quite expensive and at the same time very restrictive if they were to be applied to remote access clients.

Integrity Policies: Most access control policies aim to provide system integrity guarantees, although integrity is typically implicit. Access matrix style policies, such as role-based access control (RBAC) [20, 21], associate policies with subjects or roles that stand for a set of subjects. The integrity of a individual subject or object is not explicitly specified, but the permission assignments are intended to control access to provide sufficient integrity. Integrity is also implicit in secrecy policies, such as Bell-LaPadula [22].

Policies that explicitly reason about integrity include Biba, LOMAC, and Clark-Wilson [8, 2, 23]. In Biba and LOMAC, subjects and objects are given integrity levels, and subjects cannot retain their integrity level and depend upon lower integrity subjects or objects. Biba prohibits such information flows, whereas LOMAC permits such flows by lowering the integrity of subjects dynamically based on the dependencies they use. For Clark-Wilson, low integrity dependencies are permitted, but only if the high integrity subject either discards or upgrades the integrity of the data. In this paper we measure code integrity and their enforcement abilities to achieve coarse-grained integrity guarantees. Biba and Lomac are finer-grained models, which are often too restricted in practical environments. Clark-Wilson requires assurance that is impractical for our applications.

Client Policy Enforcement: Steve Bellovin proposed in 1999 [24] that firewalls should be considered for client systems as well as their historic place at network boundaries to improve filtering effectiveness. Ioannidis et. al. [25] proposed to distribute IPSEC credentials through trust management to such distributed firewalls. On the one hand, filtering as close as possible to the client allows fine-grained access control and ensures that all packets received by and sent from the client are intercepted by the firewall. On the other hand, moving firewalls onto the client makes them more difficult to manage and exposes them to client vulnerabilities. None of the existing approaches considers validated client security properties in the access control policy.

4. REMOTE ACCESS ARCHITECTURE

We propose a remote client access control enforcement architecture (c.f. Subsection 4.2) that implements access control using the following steps:

(i) We use non-intrusive software-stack attestation (Integrity Measurement Architecture IMA [5]) and apply it to client systems to classify the integrity of the client. This measurement approach is described in Section 2.1. The corporate VPN server receives and verifies a list consisting of measurements of all executable content that has been loaded for execution (annotated SHA-1 values of files) into the remote client's run-time since reboot. We use this information to determine in Subsection 4.1 the integrity level of the client system (i.e., the likelihood that the client is acting properly on behalf of the employee).

(ii) The integrity measurements are also used to determine if the client can enforce desired integrity and confidentiality goals. In this case, it is the presence of trusted enforcement programs rather than the absence of low integrity programs that is the issue. We describe this verification in Section 4.2.

(iii) Given the client's integrity class and the presence of the necessary enforcement software, the VPN server can delegate enforcement of the access control policy to the client system. The access control policy and decisions are described in Section 4.3.

In Section 5, we demonstrate the architecture's use.

4.1 Determine Client Integrity

In this step, the VPN server uses the validated integrity measurements of the client system [5] to classify its integrity level and enforcement abilities. The integrity level determines the identity for client accesses to corporate data and the enforcement abilities determine the ability of the client to control data once it is received.

Figure 2 illustrates this process. One measurement represents the hash chain of loads for the BIOS, boot-loader, and operating system. This should match a known value for this sequence of code loads. Subsequent measurements cover the software loaded on the operating system, including programs loaded via *exec*, kernel modules, and shared libraries. The method ensures that a remote party can cryptographically verify the source, integrity, and freshness of the measurements. The properties of the measurements ensure that no measurements may be removed or reordered once made.

Using this mechanism, the VPN server obtains and validates a list of measurements comprising all executable content that was loaded into the client's run-time since reboot. All executables corresponding to a measurement can potentially have been compromised and thus compromise the client system [1]. Consequently, compromised software might

[1]As we measure software when it is loaded, we would not see

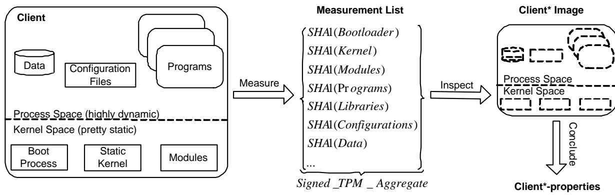

Figure 2: Attestation by Measurement Projection

$$client \in Distrusted \leftarrow \exists e \in E(client) : \neg(e \in Known) \vee (e \in (Malicious \cup Uncontrolled \cup Remote)) \quad (1)$$

$$client \in IntHigh \leftarrow \forall e \in E(client) : (e \in Acceptable) \quad (2)$$

$$client \in IntMedium \leftarrow \neg(client \in IntHigh) \wedge \forall e \in E(client) : e \in (Acceptable \cup Local) \quad (3)$$

Figure 3: Determining the Client Integrity Level

prevent the client from correctly measuring events that occur after the compromise.

Thus, we determine the integrity of the client by evaluating each measurement against a known set of measurements. Each known measurement is associated with its integrity class. Following classes partition the set of known software:

- **Malicious**: Such programs are known to be malicious (e.g., syslogd of the root toolkit, Trojan versions of libraries). The presence of one of these programs in the measurements indicates a compromise.
- **Remote (Vulnerabilities)**: Such programs use network connections and have known vulnerabilities to remote attackers. Web and mail clients are potentially included in this category. Remote attackers from the Internet or Intranet could compromise these programs, which could not be detected by the measurements taken at the time of loading those programs.
- **Local (Vulnerabilities)**: Such programs have vulnerabilities, but these vulnerabilities are limited to local input data, such as files with embedded executable content. Although local exploits are possible, most practical systems today will need to run programs that have local vulnerabilities. Additional client tools, (e.g., virus scanners) can mitigate this risk.
- **Uncontrolled**: Certain programs change the software stack, but do not yet perform integrity measurements, such as kernels not implementing the measurements or remote access programs not measuring their sensitive configuration files. The execution of these programs could result in the load of unmeasured malicious or vulnerable code. These programs should not be run at present, but could be modified to work with the system later.
- **Acceptable**: These programs contain no known vulnerabilities or malicious code and do not enable circumvention of the measurement system.

The known set includes fingerprints (SHA-1 hashes) of all known executables and other files that are expected to be found in measurement lists of remote access clients. In our example, it includes SHA-1 hash values of all Redhat 9.0 programs and libraries including updates, the fingerprints of our own extensions for client policy control, acceptable kernels, and boot configurations.

We use these sets of fingerprints to evaluate a client according to the rules 1- 3 shown in Figure 3. The above evaluation is valid until the client loads new executables that were not previously measured. To keep track of the integrity of clients connected to the Intranet, the client must update the VPN server with new integrity measurements. It is preferable that such updates be done at measurement time and prior to the actual load.

if this software becomes compromised after loading. However, we can and will use the known vulnerabilities of measured software and decide whether we assume this case or not.

4.2 Policy Enforcement Architecture

The policy enforcement architecture is shown in Figure 4. To the client, we have added a *personal firewall* that intercepts all IP packets that enter or leave the client. It makes access decisions based on a policy obtained from a *policy agent* residing in user space. In addition, we may use kernel mechanisms on the client, such as the Linux Security Modules (LSM) interface, to control access.

Further, the VPN server provides enforcement of client accesses to corporate data. The server uses its determination of the client's integrity and knowledge of the client's enforcement capabilities to control the client's access to corporate data. From this, the server determines whether the client can protect the integrity of computations (SG1) and prevent leakage of confidential corporate data (SG2).

The personal firewall extracts the service type (e.g., SSH, Telnet, HTTPS, HTTP) and the service direction (incoming or outgoing service; this can be different from the packet flow direction) from the packet and submits this information to the policy agent.

The policy (see Section 4.3) is obtained from a remote corporate policy server based on the information in a configuration file *local.conf*. The policy agent uses a local replica of the corporate access policy, *authzn_persfw.db*, to retrieve the policy for this object (e.g., outgoing SSH) and returns it to the firewall, where it is used for authorizing the packet and stored for future reference in a kernel policy cache.

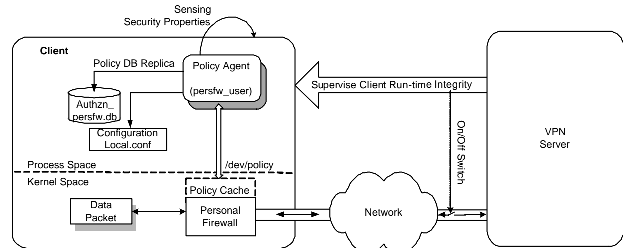

Figure 4: Policy Enforcement Architecture

Verification of the presence of a sufficient policy enforcement architecture is necessary before releasing corporate data to the client system. First, the measured kernel indicates whether it contains the personal firewall. Second,

when the policy agent is loaded, it is measured. Third, the policy agent measures the configuration and policy database files that it actually uses. Lastly, the kernel must indicate which process it is using as the policy agent.

In addition to running enforcement code, a kernel capable of enforcement will require other functions to implement the necessary controls. For confidentiality, we may require that no data is saved in persistent storage on the client. Thus, access controls may prevent file system access or a RAM file system may be used locally. Also, emerging technologies, such as sealed storage, may be used to protect the confidentiality of data unless the proper code is run on the client. The kernel and its capabilities, such as the Linux Security Modules framework, RAM file system only, etc., are verifiable from the code measurement. We do not specifically define these mechanisms.

Thus, if the kernel and policy agent are not compromised, then the measurement of their input code and configurations should be sufficient to identify their integrity and prove to the VPN server that the client system is capable to enforce corporate policies.

4.3 Security Policy

Given the client's integrity and the identity of the corporate employee, the personal firewall determines the access rights of the client to corporate data. Corporations might not be able to deliver access control decisions and enforcement on this fine-grained, per-packet-level for thousands of remote clients. However, our experiments with the prototype show that this architecture is sufficiently scalable if the policy enforcement is done locally on the client and only the general supervision of the client's computing integrity is performed remotely by the VPN server. The actual decision function implemented by the personal firewall is shown in Figure 5.

```
Bool AccessDecision(packet(service,direction), cap(userid)){

      [decision,client-constraints,packet-constraints]
                := azn(cap(userid),service,direction);

      return (decision AND  hold(client-constraints))
                         AND (hold(packet-constraints));
}
```

Figure 5: Access Control Decision Function

First, the personal firewall extracts service type and service direction from an access controlled packet. Then, the policy agent adds the user's credentials and retrieves the authorization decision from the policy data base. If the result is "permission denied", then this decision is communicated to the firewall, and this packet is discarded. If the result is "granted", then the policy agent evaluates extended attributes that refer to *client constraints*. Client constraints describe the integrity and confidentiality levels required of the client system (e.g., the client has mechanisms to implement high secrecy control). If they are fulfilled, then it communicates the *packet constraints* and a preliminary "granted" permission back to the personal firewall. The packet constraints describe the enforcement required by the client on the packet (e.g., protect the confidentiality of the data in a 'high' manner). The personal firewall will evaluate the constraints and determine whether they meet the *hold* predicate. This predicate implies that the client system's integrity and confidentiality enforcement meet or exceed the constraint defined by SG1 (firewall bypass prevention) or SG2 (information leakage prevention). We describe scenarios that aim to achieve this in Section 5.1.

5. IMPLEMENTATION

This section describes the implementation of the policy enforcement prototype. It is based on an example set of measurements from which we determine client integrity and enforce remote access based on the integrity of the client.

We implemented the remote access client system on Redhat Linux 9.0 with a 2.6.5-bk2 kernel. The policy agent agent comprises about 1100 lines of code (Loc) not including the authorization library that is part of the Tivoli Access Manager. The kernel part comprises about 1800 Loc including cache handling, connection tracking, packet classification, and authorization retrieval from user space.

5.1 Example

Figure 6 shows selected measurements that we will refer to in the course of this section.

```
#000: BC...AB (bios and grub stages aggregate)
#001: A8...5B grub.conf (boot configuration)
#002: 1238A...A22D1 vmlinuz-2.6.5-bk2-lsmtcg
#003: 84ABD...BDA4F init (first process)
#004: 9ECF0...1BE3D ld-2.3.2.so (dynamic linker)
...
#439: 2300D...47882 persfw_user (client policy agent)
#440: BB18C...BDD12 libpdauthzn.so (policy client libraries)
#441: D12D9...829EE libpdcore.so
...
#453: DF541A...BE160 local.conf (policy agent)
#454: 6AC585...DC781 authzn_persfw.db (policy db)
...
```

Figure 6: Sample of client measurements *E(client)*

There are four key measurements for determining client integrity and its enforcement abilities in Figure 6: (i) the personal firewall that is integrated into the Linux kernel (*vmlinuz-2.6.5-bk2-lsmtcg*, c.f. #002 in Fig 6) and that controls the network traffic of the client, (ii) the policy agent (*persfw_user*, c.f. #439 in Fig 6) that retrieves authorization decisions from the policy database, (iii) the configuration file of the policy agent (*local.conf*, c.f. #453 in Fig 6), and (iv) the local copy of the policy database (*authzn_persfw.db*, c.f. #454 in Fig 6) that is replicated with the corporate master data base when the policy agent starts up on the client.

The kernel and all preceding boot states are measured by the BIOS and bootloader. Succeeding measurements are induced by the kernel and applications using the Linux Integrity Measurement Architecture (IMA) [5]. We instrumented the policy agent to induce measurements on the configuration file *local.conf* and the policy data base file *authzn_persfw.bd* before loading and using them.

5.2 Server Enforcement

The VPN server will first determine the client's integrity and then it will determine the client's enforcement abilities. Based on these, it will determine whether to grant any access to the client (SG1) and whether it will entrust any confidential information to the client (SG2).

Every slight variation of these files (program version differences, policy changes in the data base, changes in the configuration file) will be reflected by differing measurement values. The VPN server compares now measurements #002,439,453,454 to a set of known and trusted fingerprints on the VPN server these programs.

At this point, we know whether these programs are authentic and configured according to the corporate policy. What we do not know is whether they are actually running or not (i.e., they could have run earlier and exited). However, we know that the kernel is running and from the kernel measurement, we can conclude whether the personal firewall is actually installed and thus filters all network traffic. The personal firewall by default (hard-coded) allows only traffic between the client and the VPN policy server that supplies replication of the policy database file.

For example, we use the kernel image measurement and compare it with a known set of SHA-1 values of kernels exhibiting images with different (meaningful) configurations. Table 1 shows kernels with different configurations. If our client kernel measurement equals one of the hashes in the table, then we confirm its integrity matches a known kernel and assign it the associated enforcement abilities.

Kernels #2 and #3 are capable of protecting system integrity (SG1). They ensure that all external network communication is completely controlled by the IP Netfilter-based personal firewall. This is necessary to close the means for an attacker to compromise the system. They do not support drivers for the serial line, such as a wireless keyboard, which could be abused by nearby attackers to issue unauthorized keyboard commands on behalf of the user. Kernel #1 supports modem connections, which can use non-IP communication and thus bypass the IP Netfilter-based firewall.

We approach confidentiality by confining data that is retrieved from the Intranet to the client system's current boot cycle. Within this boot-cycle, we ensure that this data cannot leak through client interfaces (e.g., USB, WIFI, Serial Port) or be carried over to other (less restrictive) boot cycles via file systems or other persistent storage, c.f., Figure 7.

Kernel #3 supports local persistent file systems, which means that confidential information could be carried over to other less protected boot cycles. Kernel #2 is the only one that meets the requirements ensuring that no confidential data is leaked (SG2). We configure the client to mount file systems over the protected remote access link from the Intranet. Additionally, we configure the client to use a RAM disk for its root file system, which supports just enough functionality to run a Gnome Desktop, a web browser, and the remote access policy architecture. This web browser is then used as the client interface to access confidential data inside the Intranet. Some client interfaces however, such as the display and keyboard, must be allowed in order to render the client usable. Whether sound is to be allowed or not is an issue of corporate policy. Attacks through these interfaces are excluded from our attacker model (c.f. Section 2).

At the time the remote access client disconnects from the Intranet, it must reboot in order to clean the confidential information from the memory. The personal firewall is configured not to re-open the network interface after a client disconnects from the corporate VPN until after reboot if it has accessed services requiring confidentiality properties.

The VPN server combines the integrity level of the remote client with the enforcement abilities derived from its kernel

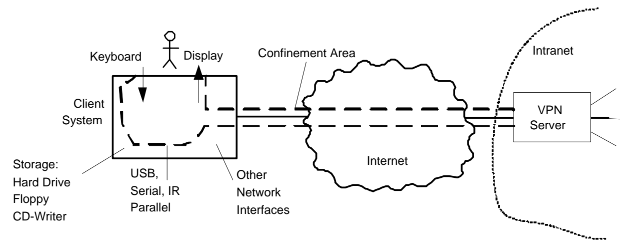

Figure 7: Confining Data on the Remote Client

image to conclude that the client system running exemplary kernel #2 with an integrity validation of high (no unknown, no malicious, no vulnerable software) satisfies security goals SG1 and SG2. Thus, requests from this client can be trusted to have actually been initiated by the user logged into the remote client and authenticated against the VPN server.

5.3 Policy Integration

We retrieve the corporate access control policy for the personal firewall using the Tivoli Access Manager (TAM) [6]. TAM is a centralized server where employee identities and their access rights are stored. Access rights are specified as permissions for subjects to perform operations on objects.

We use the identity of the user logged into the remote client to find subject identity information (i.e., TAM user id, member group ids). In our prototype, remote users are allowed to access a service if they have execute permission ('x') on the object. Objects are identified by a type of service (HTTP, SSH, etc.) combined with the service direction.

Permissions are stored in Access Control Lists (ACLs) attached to the service object and are extended by additional security property requirements using TAM extended attributes of the ACL (e.g., required client security properties, transport security on the remote access tunnel, or restrictions to specified server IP addresses). Access of a user id from a remote client (subject) to an Intranet Service (object) is permitted, if the ACL attached to the Intranet Service object allows the subject to execute this object and if additional security properties as specified in the extended attributes of the object hold (c.f. Figure 8 and Table 2).

Using this approach, we define policies to (i) allow service-specific policies that take into account the service direction (incoming/outgoing from the remote access client's perspective) and (ii) attach additional security requirements (e.g. client and packet constraints) to this service's specific policy. For this purpose, we define a new objectspace "remoteAccessPolicy" in Tivoli AM and populate it with the supported services. An objectspace is basically a directory structure, where each node and leaf represents a virtual resource to which access control lists can be attached. Figure 8 shows the object space tree for our remote access policy. Table 2 shows attribute names and values, as well as in the rightmost column the enforcement entity: packet-related constraints are enforced by the firewall, client enforcement constraints are enforced by the client policy agent, and general client-integrity constraints are enforced by the VPN server.

The authorization request for a user Mycroft would look as follows: aznDecision(cap, "/remoteAccess Policy/app-services/http/out", "x"). *Cap* comprises the capabilities of Mycroft (IDs of groups of user Mycroft), the next parameter

#	Kernel(SHA-1)	Configuration	Property
1	123AD...A22D1	IMA, Modem, Wireless, Onboard-Ethernet, Persfw Netfilter, IPSEC, no other communication, modules support	none
2	E1F98...34AA1	IMA, Onboard-Ethernet, Persfw, Netfilter, IPSEC, no other communications no modules support, no serial port support, no persistent file systems	SG1 & SG2
3	AA424...4131B	like Kernel #2 but supports ext2 file systems	SG1

Table 1: Properties Based on Kernel Configurations

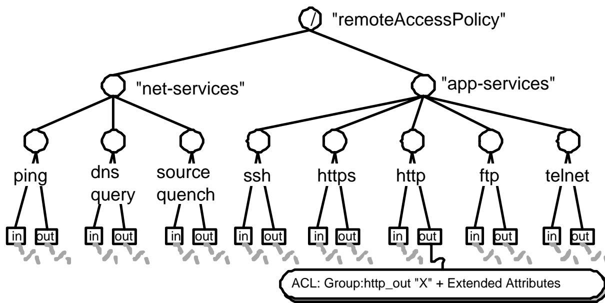

Figure 8: Remote Access Policy Object Space

Extended Attribute Name	Value	Constraint
TransportProtocol and Port	TCP/80	packet (FW)
ServerIP	10.9.*.*	packet (FW)
TransmissionSecurity	SSL/IPSEC	packet (FW)
MinClientIntegrity	Medium	client (VPN)
Meet SG1 (fw bypass protection)	Required	client (PA)
Meet SG2 (data confinement)	Not Required	client (PA)

Table 2: Extended Attributes Example: http_out

denotes the service and type (virtual resource object), the third string specifically asks for *eXecute* permission. The capabilities of the user are acquired by the authorization agent once throughout its initial binding to the authorization replica and re-used as long as the client's user does not change. The aznDecision call returns either *access denied* or *access permitted*. If access is permitted, then it returns also the extended attributes connected to the ACL that was used to determine the authorization result.

In our example, if the user at the remote access client is not member of the group *http_out*, then the policy client receives an *access denied* response. Otherwise, it receives back *access permitted* including the additional constraint attributes, implemented as a list of string-pairs. Additionally, to allow a user with ID Mycroft outgoing PING, DNSLookup, and SSH, we merely add user Mycroft to the groups *ping_out*, *dnslookup_out*, and *ssh_out*.

These extended attributes are shown in Table 2, and they refine the access constraints for the outgoing HTTP service: we allow outgoing HTTP connections (outgoing from a client perspective) for members of the 'http_out' group to all servers in the subnet 10.9.*.* if the remote access tunnel is either SSL or IPSEC protected. These constraints are enforced by the personal firewall (FW). Additionally, the client must exhibit at least medium Integrity level, which is checked by the VPN server, and must enforce SG1, which is checked by the policy agent (PA) locally on the client. This example shows, how the corporate remote access policy manages firewall rules and constraints for remote clients that are then enforced by the personal firewall as described in the next section.

5.4 Personal Firewall Implementation

We integrated the *personal firewall* into the Linux kernel. It first registers a queue handler for INET packets with the kernel Netfilter [26]. It then redirects all packets traversing the *NF_IP_LOCAL_IN* hook (incoming IP packets) and the *NF_IP_POST_ROUTING* hook (outgoing IP packets) to our queue handler. Using a queue handler, we ensure that we don't block the kernel or network traffic if we have to delay some packets to resolve access control decisions that involve policy lookups in user space. The queue handler then enforces the remote access policy on each data packet before re-inserting it with DROP or ACCEPT verdict according to the access control decision.

We implemented a stateful packet filter that keeps session information on TCP packets and thus requires access control only on the the TCP connection setup packets. For packets belonging to an existing TCP session, we only check whether the initial client security properties changed. For packets requiring access control decisions, the personal firewall will create a policy query request and send it to the policy agent through the /dev/policy interface (returning an earlier read request by a polling policy agent thread). Transaction numbers ensure that returned access control decisions can be safely linked to the requests. The personal firewall caches earlier access control decisions to resolve future queries locally. The cache is very effective because there are only a few services used by typical clients. It be used until the policy changes, which is indicated by the policy agent.

We have implemented some global rules into the personal firewall, such as to drop all packets that are not initiated by the client or the VPN server ends of the remote access tunnel. Consequently, the client can communicate over IP only through the tunnel. We also restrict traffic between the remote access client and the corporate policy server to the protocol and port used by the policy agent to replicate the policy data base. To avoid service interactions, we switched off any IPTables kernel support that could compete with our registration of the personal firewall with Netfilter hooks.

5.5 Policy Agent Implementation

The *policy agent* uses the authorization interface [27] of Tivoli Access Manager to resolve authorization requests. First it binds to the TAM using the userid and password of the remote access user. In return, the policy agent receives this user's credentials, which are used in subsequent authorization requests as described in Section 5.3. We use the local mode of the authorization interface that creates a local replica (about 600 KBytes for a tree as outlaid in Figure 8) of the AM master policy data base on the client. Authorization requests are resolved locally on the client by

the authorization library code, which yields about 11,000 authorizations/second. Tivoli Access Manager offers to either periodically poll the master policy data base for updates or to receive notifications of master policy data base changes. We implement the latter because we do not want thousands of remote access clients to poll the corporate policy server unnecessarily. This configuration is subject to notification deletion attacks; however, as the policy data base is measured and validated, the VPN server can detect out-of-date policy data bases on clients.

The policy agent polls the "/dev/policy" character device file for authorization queries from the personal firewall. Any received request is then translated into a format that is understood by the Policy Access Manager authorization service (see Section 5.3). The translated request is fulfilled by the local replica of the master data base, which returns the authorization decision. If the decision yields *access permitted*, the policy agent validates additional constraints specified in the extended attributes *Meet SG1/SG2* (c.f., Table 2).

Therefore, the policy agent determines the client security properties by reading the kernel measurement from the measurement list and comparing it to known SHA-1 values with known properties stored in the *local.conf* configuration file. Measuring *local.conf* on the client and validating it at the VPN server ensures that the policy agent and the VPN server agree about the corresponding assignment of kernel SHA-1 value and client security properties. The policy agent assumes that the measurement list is correct and expects the VPN server to disconnect the client if the client integrity becomes compromised. If it applies, the policy agent then validates the required client security properties with the ones derived from the current kernel measurement. If the constraints of the authorization decision are satisfied (given properties dominate the required properties), then the general verdict (accept) and remaining packet-related constraints (e.g., server IP constraints) are returned for enforcement to the personal firewall through the "/dev/policy" device file. Unique transaction numbers ensure that the firewall relates requests and responses correctly to each other. Waiting queues ensure smooth operation of the kernel.

6. ANALYSIS

Policy decisions and enforcement overhead. The remote access client runs Redhat Linux 9.0 on a 2GHz IBM Thinkpad T30 that includes a TPM security chip. We used a 2 GHz Netvista Linux workstation as the VPN server and a separate 2 GHz Windows 2000 Server for the Tivoli Access Manager run-time suite including LDAP and Authorization server. The initial binding of the policy client to the authorization service consists of the local replication of the master authorization data base (600 KBytes in our example) on the Access Manager server as well as acquiring the remote user's capability set and takes about 2 to 3 seconds. Table 3 shows the network round-trip delay through the personal firewall performing policy decisions on each data packet as averages of 100,000 round-trip samples.

The first two rows show the UDP and TCP round-trip reference values without the firewall in our experiment testbed. Line 3 shows the overhead introduced by resolving the policy for each round-trip (2 packets) through the user space policy agent (not using caching). Comparing this number to line 1 determines the policy overhead per packet to be about $(1087 - 162)/2 = 463\mu s$, which translates to about 2000 authorizations per second. Line 4 measures only the TCP connection tracking overhead added by the policy agent because authorization decisions are made only once at TCP connection setup time, which is not included in the measurement. The last two lines show the generally assumed overhead using kernel policy caching (15 cache lines) and assuming cache hits for all authorization decisions. The UDP overhead is about 11% because every single packet must be checked against the policy. The TCP overhead (line 6) is only 4%, representing the TCP connection tracking and no additional policy decisions for packets belonging to existing TCP connections (assuming the policy did not change).

#	Configuration	RT	Overhead
1	Reference UDP	162 μs	0%
2	Reference TCP	200 μs	0%
3	No Policy Cache UDP	1087 μs	570%
4	No Policy Cache TCP	209 μs	5%
5	Policy Caching UDP	180 μs	11%
6	Policy Caching TCP	208 μs	4%

Table 3: Prototype Performance (PL 100 bytes)

The kernel policy cache stores authorization decisions so that successive policy lookups are resolved in the kernel and don't involve interaction with the user space policy agent. As soon as the policy changes or the client security properties change, the kernel cache and existing TCP connection tracking entries are marked dirty and no longer used. Most of the performance-critical traffic will be TPC traffic, which yields about 4% overhead. The maximum policy-induced delay for UDP or TCP connection setup packets (e.g., delay when starting SSH) is about $463\mu s$; the average overhead afterwards is about 4%. Pre-loading the kernel policy cache with the most likely needed authorization decisions can further reduce the delay.

To decide if SG1 or SG2 hold, the policy agent on the client compares the local kernel measurement to a list of known kernel measurements and reads the properties of this kernel from the configuration file. This operation does not add visible overhead as our set of known kernels is small.

Overhead of measuring clients The general overhead of the integrity measuring architecture is very low [5]. Full initial measurement validations incur about 3 seconds round-trip delay, including the request from the VPN server to the client, receiving back the signed aggregate and a list of 400-500 measurements, validating the measurement list against the signed aggregate, evaluating individual measurements against the database, and inferring the client integrity properties as described in Figure 3. The related processes are not optimized. Subsequent evaluations (integrity heartbeat) only involves retrieving a new signed aggregate (1/5 second) and newly added measurements from the client.

Practically, our approach applies to remote access to sensitive services, e.g., data-entry points at banks (teller terminals), or remote access to classified data (government, corporations). We don't envision such a client-approach to connect to an ISP or to connect to Amazon.com. Thus, the remote client is governed by the security policy of the access server and it is pretty common that this restricts the client environment wherever this is possible to ensure that the client conforms to the corporate security policy and corporate software support. The fingerprint data base is accordingly maintained and includes fingerprints of currently

acceptable software (not future, nor old). E.g., a corporation will support one or two Linux distributions (e.g., Redhat or SuSE) and related software. The database for each supported distribution will be roughly 25000 entries, reflecting the distribution software package size. It grows as new updates are added to a supported distribution (approximately once a week) and is independent of the number of client systems running such a distribution. The data base is maintained by flagging existing entries as distrusted (or deleting those entries) once the represented software or configuration is known to include unacceptable vulnerabilities and patched versions are available. This goes usually hand-in-hand with adding new known fingerprints for software updates. Client's running unsupported distributions or software will be recognized by unknown (hence untrusted) fingerprints and do not add to the fingerprint data base.

Scalability is relevant in two ways – client measurement and verification. Client measurements grow linearly with the number of new software modules executed. Rerunning existing software does not result in a new measurement and has negligible performance impact as shown in (perf). Verification time per measurement is constant (based on hash table retrieval), so the verification time is also linear in the number of measurements of the client. The verification space is linear with the size of the distribution. Multiple distributions on the same platform may reuse much of the same code. The use of different platforms will result in different binaries, but the number of platforms is a small constant number, so we expect that the space considerations would still be linear in distribution size.

7. FUTURE WORK AND CONCLUSION

We have designed and implemented a novel access control architecture that enables corporations to verify client integrity properties and establish trust into the client's policy enforcement before allowing them remote access to corporate Intranet services. To this end, we have shown how to (1) determine the integrity level of a client system based on the code running on the client; (2) determine whether to trust this client to enforce information flow controls necessary to make such integrity assumptions about client; (3) determine whether additional security properties, such as SG1 or SG2, must be enforced by the client and whether the kernel supports these; (4) integrate the integrity of the client into the remote access control policies governing the client's access to the corporate servers; (5) enforce this policy on remote access clients and the VPN server. We have introduced an integrity heart-beat enabling the VPN server to track changes in the remote client's security properties (i.e., sense relevant changes in the client's software stack) and implement resulting policy changes. We have implemented a Linux 2.6 prototype system that utilizes the TPM measurement and attestation framework, existing Linux Netfilter, and Tivoli Access Manager to control remote client access to corporate data. This prototype illustrates that our solution integrates seamlessly into scalable corporate policy management and introduces only minor performance overhead.

Future work includes measuring isolation properties of client systems in order to handle unknown or distrusted fingerprints. To this end, we are experimenting with measuring SELinux policy and policy enforcement and with extending the measurement architecture onto virtual machine monitors allowing to attest to a single isolated virtual machine.

8. REFERENCES

[1] Trusted Computing Group. *Trusted Platform Module Main Specification*, October 2003. Version 1.2, Revision 62, http://www.trustedcomputinggroup.org.

[2] T. Frazer. LOMAC: Low water-mark integrity protection for cots environments. In *IEEE Symposium on Security and Privacy*, May 2000.

[3] J. P. Anderson. Computer Security Technology Planning Study, 1972.

[4] G. Karjoth. Access Control with IBM Tivoli Access Manager. *ACM Transactions on Information and System Security*, 6(2):232–257, 2003.

[5] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Thirteenth Usenix Security Symposium*, pages 223–238, August 2004.

[6] IBM Tivoli. IBM Tivoli Access Manager for e-business. http://www-3.ibm.com/software/tivoli/products/access-mgr-e-bus/.

[7] D. Eastlake and P. Jones. Secure Hash Algorithm 1 (SHA1), September 2001. Request for Comment 3174.

[8] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, Mitre Corporation, Mitre Corp, Bedford MA, June 1975.

[9] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A Secure and Reliable Bootstrap Architecture. *IEEE Computer Society Conference on Security and Privacy*, pages 65–71, 1997.

[10] S. W. Smith. Outgoing authentication for programmable secure coprocessors. In *ESORICS*, pages 72–89, 2002.

[11] J. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart. Building the IBM 4758 Secure Coprocessor. *IEEE Computer*, 34(10):57–66, 2001.

[12] IBM PCI-X Cryptographic Coprocessor, 2004. http://www-3.ibm.com/security/cryptocards/html/pcixcc.shtml.

[13] D. Hollingworth and T. Redmond. Enhancing operating system resistance to information warfare. *MILCOM 2000. 21st Century Military Communications Conference Proceedings*, pages 1037–1041, 2000.

[14] J. Dyer, R. Perez, R. Sailer, and L. van Doorn. Personal Firewalls and Intrusion Detection Systems. In *2nd Australian Information Warfare & Security Conference (IWAR)*, November 2001.

[15] J. Molina A. Mishra and W. Arbaugh. The co-processor as an independent auditor. http:// www.missl.cs.umd.edu/komoku/documents/coauditor.ps.

[16] Xiaolan Zhang, Leendert van Doorn, Trent Jaeger, Ronald Perez, and Reiner Sailer. Secure Coprocessor-based Intrusion Detection. In *Tenth ACM SIGOPS European Workshop*, September 2002.

[17] Paul England and Marcus Peinado. Authenticated operation of open computing devices. In *ACISP 2002*, LNCS, pages 346–361. Springer-Verlag, July 2002.

[18] B. A. LaMacchia. Next-generation secure computing base (NGSCB), April 2003. RSA Conference 2003, San Francisco.

[19] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proc. 9th ACM Symposium on Operating Systems Principles*, pages 193–206, 2003.

[20] D.F. Ferraiolo and D.R. Kuhn. Role based access control. In *15th National Computer Security Conference*, 1992.

[21] R. S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. In *IEEE Computer*, volume 29(2), pages 38–47. IEEE Press, 1996.

[22] D. E. Bell and L. J. LaPadula. Securecomputer system: Unified exposition and multics interpretation. Technical Report MTR-2997, MITRE Corporation, Bedford, MA, July 1975.

[23] D. R. Wilson D. D. Clark. A comparison of commercial and military computer security policies. In *IEEE Symposium on Security and Privacy*, 1987.

[24] S. M. Bellovin. Distributed Firewalls. login, November 1999.

[25] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a distributed firewall. In *Proceedings of the ACM Computer and Communications Security (CCS) 2000*, pages 190–199, November 2000.

[26] The netfilter/iptables project, 2004. http://www.netfilter.org.

[27] The Open Group. Authorization (AZN) API – Technical Standard. http://www.opengroup.org/products/publications/catalog/c908.htm.