11-29-2004

Secure Hardware Enhanced MyProxy: A Ph.D. Thesis Proposal

John Marchesini
Dartmouth College

David Kotz
Dartmouth College

Secure Hardware Enhanced MyProxy

A Ph.D. Thesis Proposal*
Dartmouth College Technical Report TR2004-525
John Marchesini and Sean Smith
Department of Computer Science
Dartmouth College

November 29, 2004

1 Introduction

Because public-key cryptography can enable secure information exchange between parties that do not share secrets a priori, PKI has long promised the vision of enabling secure information services in large, distributed populations.

A number of useful applications become possible with PKI. While the applications differ in how they use keys (e.g., S/MIME uses the key for message encryption and signing, while client-side SSL uses the key for authentication), all applications share one assumption: users have keypairs. Where these user keypairs are stored and used is the primary focus of this research.

Traditionally, users either put their key on some sort of hardware device such as a smart card or USB token, or they place it directly on the hard disk such as in a browser or system keystore. Most modern operating systems (such as Windows and Mac OSX) include a keystore and a set of *Cryptographic Service Providers* (CSPs) which use the key. In fact, many cross-platform software systems, such as the Java Runtime and the Netscape/Mozilla Web browser include their own keystore so that they may use a user's keypair without having to rely on the underlying OS (thus enhancing portability).

Most keystores fall into one of four basic categories:

- A software token which stores the key on disk (most likely in some sort of encrypted format). Examples of this approach include the default CSP for Windows and the Mozilla/Netscape Web browser.
- A hardware token which stores the key and performs key operations. The interaction between an application and the key is typically mediated by the OS (although in some cases, the application may interact with the device directly, e.g., Mozilla). In order for the OS or application to be able to speak to the token, the token vendor must provide a driver for the device which adheres to one of the two common standards for communicating with cryptographic devices: CAPI for Microsoft [37], and

*This thesis is based in part on earlier research, and incorporates material from earlier papers [33, 34]. This Technical Report describes a project which is currently underway. The thesis proposal occurred on June 15, 2004.

RSA's PKCS#11 [60] for the rest of the world. Examples of hardware tokens include the Aladdin eToken and Spyrus Rosetta USB tokens, as well as more powerful devices (sometimes referred to as *cryptographic accelerators* or *Hardware Security Modules* (HSM)) such as nCipher's nShield [41].

- A *secure coprocessor* which stores the key, can perform key operations internally using cryptographic hardware, and can even house the applications directly, such as the IBM 4758 [7, 68]. These devices can also be used as cryptographic accelerators or HSMs by not placing the applications inside of the device.

- A *credential repository* which is a dedicated machine that stores private keys for a number of users. When a user Alice wishes to perform key operations, she must first authenticate to the repository. The repository then certifies a temporary key with the Alice's permanent key via a digital signature, or actively participates in the requested key operation. Examples of credential repositories include MyProxy [42], hardened MyProxy [30], and SEM [4]. These systems will all be discussed in some detail throughout this proposal.

Problems with the Status Quo In previous work, we examined the security aspects of some of the standard keystores and the their interaction with the OS [34]. We concluded that software tokens are not safe places to store private keys, and we demonstrated the permeability of keystores such as the Microsoft default CSP and the Mozilla keystore. Our experiments showed that it is possible for an attacker to either steal the private key or use it at will.

In addition to being unsafe, software keystores have the disadvantage of being immobile. Once a key is installed on a desktop, the only way to transport it to another machine is to export it and re-import it on the new machine. As user populations become more mobile and acquire multiple devices (e.g., it's not uncommon for someone to have a computer, PDA, and a cell phone), this immobility becomes more problematic.

Hardware tokens (e.g., the Aladdin and Spyrus USB tokens) claim to solve both of these problems—they get the key off of the desktop and give users mobility. We experimented with these devices as well, and found that an attacker is still able to use the key at will. With respect to mobility, devices such as USB tokens can add some benefit, provided that the appropriate software is installed on each machine, and that users use supported OSes (the tokens we experimented with did not have Apple or Linux support).

A more detailed analysis of these problems and their impact on the PKI vision will be discussed in Section 2 (see [34] for details of our experiments). Can secure coprocessors and credential repositories do any better than hardware and software tokens?

Secure Coprocessors In other previous work, we examined secure coprocessors (e.g., [63, 79, 80]): careful interweaving of physical armor and software protections can create a device that, with high assurance, possesses a different security domain from its host machine, and even from a party with direct physical access. Such devices have been shown to be feasible as commercial products [7, 68] and can even run Linux and modern build tools [22]. In our lab, we have explored using secure coprocessors for *trusted computing*—both as general designs (e.g., [46]) as well as real prototypes (e.g., [23])—but repeatedly were hampered by their relatively weak computational power. Their relatively high cost also inhibits widespread adoption, particularly at clients.

In some sense, secure coprocessors offer high-assurance security at the price of low performance (and high

cost). However, in industry, two new trusted computing initiatives have emerged: the *Trusted Computing Platform Alliance (TCPA)* (now renamed the *Trusted Computing Group (TCG)* [45, 71, 72, 73]) and Microsoft's *Palladium* (now renamed the *Next Generation Secure Computing Base (NGSCB)* [11, 12, 13, 14]). These also seem to be tied up with Intel's *LaGrande* initiative [69].

Many in the field ([1, 59] are notable examples) have criticized these initiatives for their potential negative social effects; others (e.g, [20, 54, 55]) have seen positive potential. (Felten [17] and Schneider [57] give more balanced high-level overviews.)

These new initiatives target a different tradeoff: lower-assurance security that protects an entire desktop platform (thus greatly increasing the power of the trusted platform) and is cheap enough to be commercially feasible. Indeed, the TCG technology has been available on various IBM platforms, and other vendors have discussed availability. Some academic efforts [29, 36, 70] have also explored alternative approaches in this "use a small amount of hardware security" space, but no silicon is available for experiments yet.

Last year, we built a trusted computing platform based on the TCG specifications and hardware, and we called this platform "Bear". Section 4 will give an overview of Bear and details can be found in previous work [31, 33].

This Thesis The picture painted by these previous projects suggests that common desktops are not secure enough for use as PKI clients, and adding USB hardware tokens does not provide a solution either. The picture also suggests that trusted computing can improve the security of client machines.

In practice, the Grid computing community has embraced the credential repository approach in order to provide security and mobility to clients. Their repository is called *MyProxy* [42], and there have even been efforts to harden a MyProxy repository with an IBM 4758 [30].

The question that I propose to investigate is: *Can I build a system which applies trusted computing hardware in a reasonable manner in order to make desktops usable for PKI?* This question will be further defined throughout the course of this proposal, but in essence, I'm asking if it is possible to build a system which uses secure hardware to store private keys in a credential repository, prevents private key disclosure, and allows relying parties to reason about the system. It is worth noting that the application of trusted computing hardware in a *reasonable* manner implies that clients should not have to be equipped with high-end HSMs to participate in the system. The system should be flexible enough to accommodate clients with high-end HSMs, but should not require clients to have such a device.

Concretely, I plan to start with the hardened MyProxy design, and extend it in the following ways:

- I will place the entire repository application on a secure platform (as opposed to just the keys in the hardened MyProxy approach),
- I will take advantage of secure hardware on the clients, if available,
- I will incorporate a policy framework into the system so that users can clearly express their wishes with respect to delegation, and
- I will formally reason about the design to ensure that relying parties have a sound reason to trust certificates generated by the system.

The idea is to extend the MyProxy system to take full advantage of secure hardware—on the servers, clients, and in the policy framework. The resulting design is called *Hardware Enhanced MyProxy* (SHEMP).

Since SHEMP is based on the MyProxy design, it can be used for the same purpose: to serve as an online credential repository for Grid PKIs. However, the incorporation of secure hardware and a policy framework allow SHEMP to be used for a number of additional applications such as general PKIs and mobile PKI clients (discussed in Section 6). Additionally, some of the tools used to build the SHEMP system have interesting applications of their own, outside the context of SHEMP (discussed in Section 4).

This Paper Section 2 examines the problem in detail. Section 3 discusses the criteria that a solution to the problem should meet. Section 4 discusses the SHEMP toolkit in detail and Section 5 applies those tools to build a system. Section 6 explores a number of applications which could benefit from the proposed system. Section 7 discusses other approaches and finally, Section 8 concludes and presents a proposed timeline.

2 The Problem

Put simply, the problem that I propose to solve is that modern desktops are unsuitable for use as PKI clients. They allow a user's private key to be stolen or used at an attacker's will, they make it difficult for users (and application authors) to do the "right thing", they are inherently immobile, and they do not allow relying parties to make good trust judgments about the system (i.e., they allow the key to be used for transactions which the user was not aware of or did not intend). A more detailed description of the experiments used to draw this conclusion can be found in previous work (see [34]); this section presents a brief analysis of some of those results.

Software One cause of the problem is the software which runs on the client's desktop and its interaction with the underlying hardware. Given the complexity of modern software, it has become almost impossible to know exactly what is happening during a given computation. Is the machine executing the right code? Has critical data been altered or stolen?

One unfortunate consequence of this increase in complexity is a reduction in the level of usability of the system. Clearly, it becomes difficult for users to make reasonable trust judgments about the system if the system is difficult to use. In a security setting, this inability to reason about the system can thwart the security efforts that the system's designers have implemented.

A second unfortunate result of modern software's complexity is an expansion in the set of software that must be trusted in order for the system to operate correctly. This set of software is often referred to as the *Trusted Computing Base* (TCB). A good discussion of the TCB can be found in the "Orange Book" [43], and the motivations to keep the TCB small are clear: minimize the attacker's target and maximize the chance for developers to build secure systems.

Placing a private key on such a complex system is problematic. By exploiting the complexity, it is possible for an attacker to trick users into giving away their key directly, or use it for purposes which they are unaware of or did not intend. By exploiting the fact that so much of a complex system needs to be trusted in order for it to behave correctly, it is possible for an attacker to either get the key directly, or be able to use it at will without alerting the key's owner. We found that getting one user-level executable to run on the client is enough to accomplish a successful attack.

Hardware Many in the field suggest getting the private key off of the desktop altogether and placing it in a separate secure device of some sort. Taking the key to a specialty device (such as an inexpensive USB token) would seem to reduce the likelihood of key theft as well as shrink the amount of software which has to be trusted in order for the system to be secure. Specifically, at first glance, it would appear that just the device and the software which provides access to the device (i.e., its CSP) need to be trusted.

However, relying on such a device is also problematic. Just putting the private key on a token is not enough. The token's CSP is still interacting with the whole system (the OS and CAPI), and the entire system still has to be trusted. Putting the private key on a token gives some physical security and makes it harder to steal the key (physical violence notwithstanding), but it does not protect against malicious use, and it does not increase usability.

Secure coprocessing is an improvement from a security standpoint, but it is not a magic bullet either. From a practical standpoint, high end devices such as the 4758 are far too expensive to deploy at every client. On the other end of the spectrum, lower priced devices (e.g., the TPM) probably cannot withstand many common attacks (such as hardware attacks, or attacks from root) without additional measures (e.g., aid from the processor, such as what is being considered in the literature [29, 36, 69, 70]).

Immobility In addition to the security and cost considerations mentioned above, the desktop PKI client paradigm suffers another problem: immobility. Modern computing environments are becoming increasingly distributed and user populations are becoming increasingly mobile. To further the problem, the number of computing devices that a typical user owns is growing. It is not uncommon for someone to own a desktop, a laptop, a cell phone, and a PDA. Which device(s) should house the private key?

One proposal is to use inexpensive tokens (such as USB tokens) and allow users to carry their token with them across devices and computing environments. This approach has a number of drawbacks in addition to the security problems mentioned above. First, some devices may not have the proper hardware or software installed, or may not have support altogether. Second, a particular machine may not be trustworthy, or may have malware installed which abuses the private key. Again, getting the private key in a token does not shrink the TCB.

Another proposal is to move the key around on some removable media (e.g., a floppy) and export the key to some intermediate format (e.g., PKCS#12 [61]) and then import the key at the destination. This approach suffers a number of drawbacks as well. First, some devices may not support the media—e.g., I'm unaware of cell phones with floppy drives. Second, the intermediate format may be insecure. Peter Gutmann revealed a vulnerability in the way that the key is stored on disk once it has been exported (in a .pwl or .pfx file). There is a tool named breakms, available from Gutmann's Web site [21], which performs a dictionary attack to discover the password used to protect the file and outputs the private key.

3 Criteria for a Solution

In order for *any* proposed solution to succeed in making desktops usable for PKI, it must address a range of issues including security, usability, and mobility. For the solution to be of any practical interest, it must safely store and use the private key, give application developers flexibility while maintaining security, match the model of real world user populations, and allow relying parties to make reasonable judgments about the system.

Security The notion of security is difficult (or impossible) to measure in a practical system. Within a formal framework, one can prove that a system is secure, but once the formal frameworks give way to implementations, problems often arise. As a result, the operating definition of security in this proposal involves minimizing the risk, impact, and window of opportunity for misuse of a user's key.

One design force at work in the SHEMP system is the notion of using very secure hardware in some places and less or no secure hardware in others. In SHEMP, machines which house users' private keys are called *key repositories*. Machines which actually use the key on a user's behalf are called *clients*. In general, both repositories and clients can have very secure hardware such as a secure coprocessor, less secure hardware such as a TPM, or no secure hardware at all.

In order to make any real claims of security, key repositories should be able to withstand a wide range of attacks. In designing the repository, I assume that an attacker can get root privileges on the repository's host machine. This implies that the attacker can watch any process's memory, and run any code of his choice on the host. Furthermore, since secure hardware could be involved, I assume that an attacker has physical access to that hardware and can attempt to perform local hardware attacks. As a result, repositories should be able to resist local physical and software attacks, and should refuse to disclose any user's private key, even if the attack is running with root privileges.

In practice, this may involve using a device such as an IBM 4758 to house the repository, thus giving the repository a different security domain than its host. Although using a device such as the 4758 is necessary to achieve maximum security, the design I am proposing is flexible enough to deal with any type of hardware on the repository. No matter what type of hardware is used, the system must be able to give all parties enough information about the repository so that it can make informed decisions.

The model for clients is different—clients cannot be trusted with users' private keys. The system I am proposing is designed to be flexible enough to accommodate clients with a range of security levels, as well as provide a means for expressing those security levels. The mechanisms used to achieve this extensibility will be discussed in Section 5, but for the purposes of defining a threat model, it is safe to assume that clients should not be trusted to house the user's private key directly.

Concretely, clients obtain an authorization to use the private key which lives in the repository. The strength of this authorization will depend on the security level of the client and repository. For instance, clients with attestation capabilities will be able to use those in order to enhance authorization. In defining a threat model, it is only fair to assume that this authorization process can be compromised as well, allowing an attacker to use the key without actually obtaining it (the difficulty of the attacker's task depends on the security level of the client). In the event that the authorization is compromised, the system must minimize the time that the authorization is valid. In previous work, we found that the status quo allows an attacker to use the key indefinitely [34].

Usability The second feature that a proposed solution should provide is usability. Developers must be able to use this platform to build and deploy real applications. This requires that the platform must be easily programmable with modern tools. The platform must also allow easy maintenance and upgrade of its software. Finally, the platform must permit installation and upgrade of software to happen at the end user's site since forcing software developers to ship hardware stifles innovation, limits user choice, and complicates trust [67].

Mobility The third feature that a proposed solution must provide is mobility. Modern user populations increasingly use multiple computing platforms from multiple locations. A solution should allow for mobility, and should not risk key disclosure any time the user moves geographically or uses different devices.

Reasoning About the System All of the above features are meaningless unless relying parties can reason about the system. To this end, the last feature that any proposed solution should provide is the ability to reason about it. Specifically, if Alice is given a pile of certificates from Bob, and Bob was issued these certificates from the system, what can Alice deduce about Bob—does she have any real reason to trust him?

In the context of this proposal, the *relying party* Alice makes decisions about Bob based on the certificate(s) that the *target* Bob gives her. Specifically, it is not enough for these certificates to express some transitive trust relationship which begins at one of Alice's trust anchors and ends with Bob. Those certificates must express something about the environment Bob is operating in. How secure is his client machine? How secure is the key repository? Was the software Bob used to authorize himself the "right" software? How likely is it that his authorization was compromised?

In addition to providing a mechanism for relying parties to answer such questions, the system should be flexible and expressive enough to allow Alice to answer domain-specific security questions about Bob's environment (e.g., is Bob's machine inside the firewall?).

4 My Toolkit

As a researcher in the Dartmouth PKI Lab, I have had a chance to explore and develop a number of tools which will serve as the building blocks for constructing a solution. Essentially, my toolkit can be divided into four major categories: secure hardware which is used as the basic keystores (both at repositories and clients, when available), a delegation infrastructure which is used to establish secrets across multiple platforms, a policy language which is used to express private keys' usage and delegation policies at the repository as well as express attributes of repositories and clients, and a formal framework which is used to reason about the system.

Secure Hardware Over the years, the Dartmouth PKI Lab has built a number of systems which involve and/or enhance secure coprocessors. Most of our initial systems were constructed around the IBM 4758, as Sean Smith brought it to the PKI Lab from IBM [7, 63, 67, 68]. Members of our group have used these devices to enhance privacy [23], harden PKI [25, 32, 64], and enhance S/MIME [46].

The 4758 is a secure coprocessor which provides secure storage facilities, cryptographic acceleration, and a platform on which to run third-party applications. The 4758 is a very secure device, having been validated to FIPS 140-1 Level 4. It can withstand both software and hardware attacks, and effectively provides a different security domain from its host machine.

An extremely useful feature of the 4758 is what it calls *Outbound Authentication*, which enables applications running inside of the 4758 to authenticate themselves to remote parties [65]. A good overview of the 4758 and its capabilities can be found in the literature (e.g., [7, 63, 67, 68]).

More recent projects have involved constructing a "virtual" coprocessor out of commodity hardware. Our initial design and prototype was based on the TCG specification (see [45, 71, 72, 73]) and was called

"Bear" [33].

The Bear platform is less secure than the 4758. It does provide a means to ensure file integrity for files which a *Security Admin* decides are necessary. However, since the design is based on the TCG specification and hardware, it is susceptible to local hardware attacks, as well as attacks from root [31, 33].

Bear has a mechanism which allows it to "attest" to the integrity of the platform when challenged. The TCG specifications refer to this mechanism as *attestation*. More information about Bear can be found in previous work (e.g., [31, 33]), and a summary of the attestation mechanism can be found in earlier work [31] as well as the literature (e.g., [45, 56, 71, 72, 73]).

Delegation A detailed description of the design of my solution will be given in Section 5, but the main idea is to store the user's private key in a key repository, and issue short-lived keypairs (contained in *Proxy Certificates* [74, 77]) to the client. As noted earlier, repositories and clients may vary with respect to the level of secure hardware they possess, and thus may vary in their methods of generating and storing the short-lived keypairs. For example, if a client is a Bear platform, then the authorization to use the user's key in the repository will be delegated to the temporary key stored in the TPM on the client.

The delegation framework used to create the short-lived keys and certificates should be simple, widely accepted in practice, and should not require extra infrastructure (such as CRLs, etc). There are a number of frameworks and certificate formats to choose from, such as X.509 [16], X.509 Proxy Certificates [74, 77], Permis [6, 47], Keynote Credential Assertions [3, 27], and SDSI/SPKI [8, 9, 10]. Nazareth provides an overview of a number of these delegation systems, as well as comparisons between them [39].

I chose X.509 Proxy Certificates for a number of reasons. First of all, they are standardized by the IETF, and are awaiting an RFC number assignment. Second, because they are X.509-based, they can be used in many places in the existing infrastructure that are already outfitted to deal with X.509 certificates. Third, they are widely used in the Grid community and are used in the dominant middleware for Grid deployments: the Globus Toolkit [18]. Fourth, they allow dynamic delegation without the help of a third party, allowing clients to obtain a Proxy Certificate without having to endure the cumbersome vetting process at the Certificate Authority. Fifth, the Proxy Certificate standard defines a Proxy Certificate Information (PCI) X.509 extension which can be used to carry a wide variety of (possibly domain-specific) policy statements (e.g., XACML statements). Last, the PCI extensions can be used to limit the privileges granted by the delegator (e.g., perhaps according to the security level of the client).

In all fairness, SDSI/SPKI certificates could probably be used as well. Our lab has a produced a number of projects which use the SDSI/SPKI format [39, 66], and using Proxy Certificates gives me an opportunity to explore something new. Further, some of our lab's projects have encountered difficulties when passing SDSI/SPKI certificates to relying parties [66].

Policy In order to enhance the expressiveness and usability of the system, users must be able to relay their wishes regarding key usage to others. Further, the system must also be able to convey attributes of both key repositories and clients to relying parties. The "Extended Key Usage" field of X.509 certificates allows users to restrict key usage, but it does not consider attributes of the key's environment (i.e., the repository and client). Since I could not find a tool which fit my purposes, I developed my own.

The goal is for the relying party Bob, upon receiving a Proxy Certificate for Alice, to be able to examine the certificate, Alice's *Key Usage Policy* (KUP), and the attributes about Alice's client and the repository which

houses Alice's private key, and decide whether to trust the certificate. Additionally, the repository should enforce Alice's KUP, and only issue a Proxy Certificate if the client and repository meet the conditions in the KUP. The Proxy Certificate could possibly be limited to some subset of Alice's privileges, as expressed in her KUP.

There are a number of policy frameworks which can be used to accomplish these tasks, and the system design is agnostic to which framework is used. However, for the sake of implementation, I chose to use the *eXtensible Access Control Markup Language* (XACML) [78]. XACML is an XML-based language for expressing generic policies and attributes. A *Policy Decision Point* (PDP) takes a policy and a set of attributes about the requester, and makes an access control decision. From a practical standpoint, XACML is generic enough to perform the tasks at hand, and has an open-source implementation (thanks to Seth Proctor at Sun Microsystems) [49] which is implemented in the language of my prototype: Java.

Protocol Analysis Tool The last tool of interest is one which can be used to reason about trust in the formal sense. If the system is sound, then it should generate valid certificates if and only if Alice authorized the generation of those certificates.

Again, the details of the architecture will be described in Section 5, but briefly, the overhead incurred by the system is that users now have chains of certificates as opposed to just one. The first certificate is from the user's *Certificate Authority* (CA) to the user's keypair, and the second is the Proxy Certificate from the user's keypair to the short-lived delegation key. The implication is that relying parties now have to reason about a chain of certificates instead of just one. Additionally, relying parties also have to consider the attributes of the repository and server along with Alice's KUP in making decisions.

A number of formal frameworks exist to reason about bindings between entities and keys and the protocols used to establish these bindings [5, 28, 35]. My intention is to apply the Maurer-style notation [35] to the SHEMP design, as it is simple and expressive enough to capture most of the design concepts. It is quite likely that I will have to extend the notation in order to capture all of the concepts in SHEMP.

5 My System

Using the toolkit presented in Section 4, how can I build a system which meets the criteria of Section 3 and solves the problems of Section 2, thus answering the question: "Can I build a system which applies trusted computing hardware in a reasonable manner in order to make desktops usable for PKI?"

Concretely, the goal of the SHEMP system is to allow the relying party Bob to be able to make valid trust judgments about Alice upon receiving a Proxy Certificate from her. Bob should have some reason to believe that Alice authorized the issuance of her Proxy Certificate for the intended purpose(s), and that the private key described in the Proxy Certificate is authentic.

Furthermore, a successful solution must meet the criteria of Section 3, as well as the formal criteria which will be developed in the thesis (Section 8.1 discusses this issue in more detail). The next section describes one candidate architecture based on X.509 certificates. Section 5.2 considers how well the architecture meets the criteria in Section 3, and the formal analysis will be done as part of the thesis.

5.1 The SHEMP Architecture

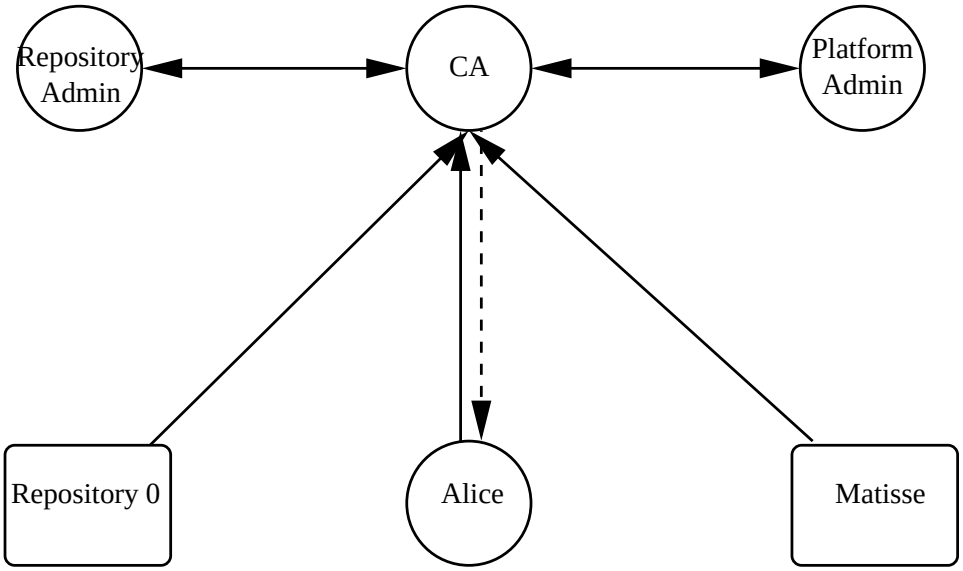

Figure 1: The parties in the SHEMP system. The circles represent individuals or organizations and the boxes represent machines. The arrows indicate trust relationships between the parties; an arrow from A to B means "A trusts B".

The Players The first step in developing a system which relies on and enables trust is defining the entities involved and showing the trust relationships between them.

Initially, there are three familiar parties involved: a CA, a user (Alice), and a user's machine (Matisse). As in any typical PKI, Alice trusts her CA to certify members of her population (including herself). This relationship is depicted as a solid arrow from Alice to the CA in Figure 1.

In order for the CA to trust Alice, it must believe her identity and that she has the private key matching the public key in her certificate request (typically a *Registration Authority* (RA) verifies Alice's identity on the CA's behalf). Once the CA/RA believe Alice's identity is authentic and that she owns the private key, the CA will express its trust in Alice in the form of a CA-signed identity certificate. This relationship is depicted as a dashed edge from the CA to Alice in Figure 1.

For an application running on Alice's machine (Matisse) to trust certificates signed by the CA (such as Alice's), it needs to have the CA certificate installed. This relationship is represented by the edge from Matisse to the CA in Figure 1. To illustrate a concrete example of the necessity of this relationship, assume that some organization uses S/MIME mail. If Alice and Bob both have identity certificates signed by the CA and Bob sends Alice a signed message, then Alice's mail program needs to know Bob's certificate and it needs to trust the entity which vouched for Bob's identity (the CA/RA).

In addition to the three familiar parties described above, the SHEMP system introduces three more: a *Repository Administrator* who runs the key repository(s), a *Platform Administrator* who is in charge of the platforms in the domain (such as Matisse), and at least one key repository (depicted as *Repository 0* in Figure 1).

The Repository Administrator is in charge of operating the key repository. Since the repository contains the entire population's private keys (and is thus a target for attacks), it must be maintained with care. Concretely, the Repository Administrator is in charge of loading private keys into the repository and vouching for the repository's identity and security level (these will be discussed below). Thus, it is necessary for the CA to trust the Repository Administrator. Since the Repository Administrator is a member of the CA's domain

(in fact, probably part of the same organizational unit—such as Dartmouth College Computing Services), it trusts the CA as well. This relationship is depicted by the edge connecting the Repository Administrator to the CA in Figure 1.

The Platform Administrator is in charge of the platforms that end users (e.g., Alice) will use. At the base level, the Platform Administrator has the same responsibilities as a typical system administrator: configuring machines, installing and upgrading software, applying patches, etc. Additionally, the Platform Administrator is in charge of creating and vouching for platform identities and security properties (discussed below). Since the Platform Administrator is in charge of the nodes that will be using the keys stored in the repository, the CA must trust the Platform Administrator. Since the Platform Administrator is a part of the CA's domain (again, possibly part of the same organizational unit), it trusts the CA. The relationship is shown in Figure 1 as the edge connecting the Platform Administrator to the CA.

The last entity involved is the actual key repository which holds the users' private keys. As with individual platforms (e.g., Matisse), the repository trusts the CA. This relationship makes it possible for entities with CA-signed certificates to establish SSL connections to the repository. Since the repository trusts the CA, it believes the identity of an entity with a CA-signed certificate. This relationship is represented by the edge between Repository 0 and the CA in Figure 1

It is worth noting that there could be more entities involved in the system. For example, there will most certainly be multiple users (e.g., Alices) and platforms (e.g., Matisses). Further there could be any number of CAs in virtually any valid architecture (hierarchy, mesh, etc.). There could also be multiple repositories with different Repository Administrators, as well as multiple Platform Administrators. The only constraint that must be enforced is that the multiple parties form a valid chain of certificates.

For example, assume that Dartmouth College has one root CA for the college, and each department runs CAs for their own department. In this case, the Computer Science Department runs a CA, and its CA certificate is signed by the Dartmouth College root CA, thus forming a chain. In this example, the department may also run its own repository, and the department's Repository Administrator is certified by some college-wide repository administrator (again forming a chain of certificates). A college-wide Platform Administrator could certify some departmental Platform Administrator to have the department's machines under her jurisdiction (again, forming a chain). The details of the certificates used by SHEMP will be discussed below, but it is important to note that the system generalizes beyond the entities in Figure 1. The set of entities in Figure 1 is the smallest set which is necessary and sufficient to describe the system.

Identity Certificates Setup The way SHEMP (and PKI in general) represents trust is via certificates. From the initial trust relationships between the entities in Figure 1, a number of certificates can be immediately issued. Figure 2 illustrates these initial certificates; they are contained in the dashed box which could possibly represent an LDAP directory where users go to locate certificates.

The certificates are issued from the CA to entities which have a mutual trust relationship with the CA. Since the administrators and Alice all have such a relationship with the CA, they are all issued identity certificates. The certificates not shown in Figure 2 are the CA certificates which are installed at the key repository and at the platform. As previously discussed, these certificates are necessary to allow things like client-side SSL connections, and are represented by the one-directional edges in Figure 2.

The first phase of setup begins when machines are added to the domain. As a repository is added, the Repository Administrator must take a number of steps to set it up. First, he must generate a keypair for the repository. This keypair can be generated in a number of ways depending on what type of platform the

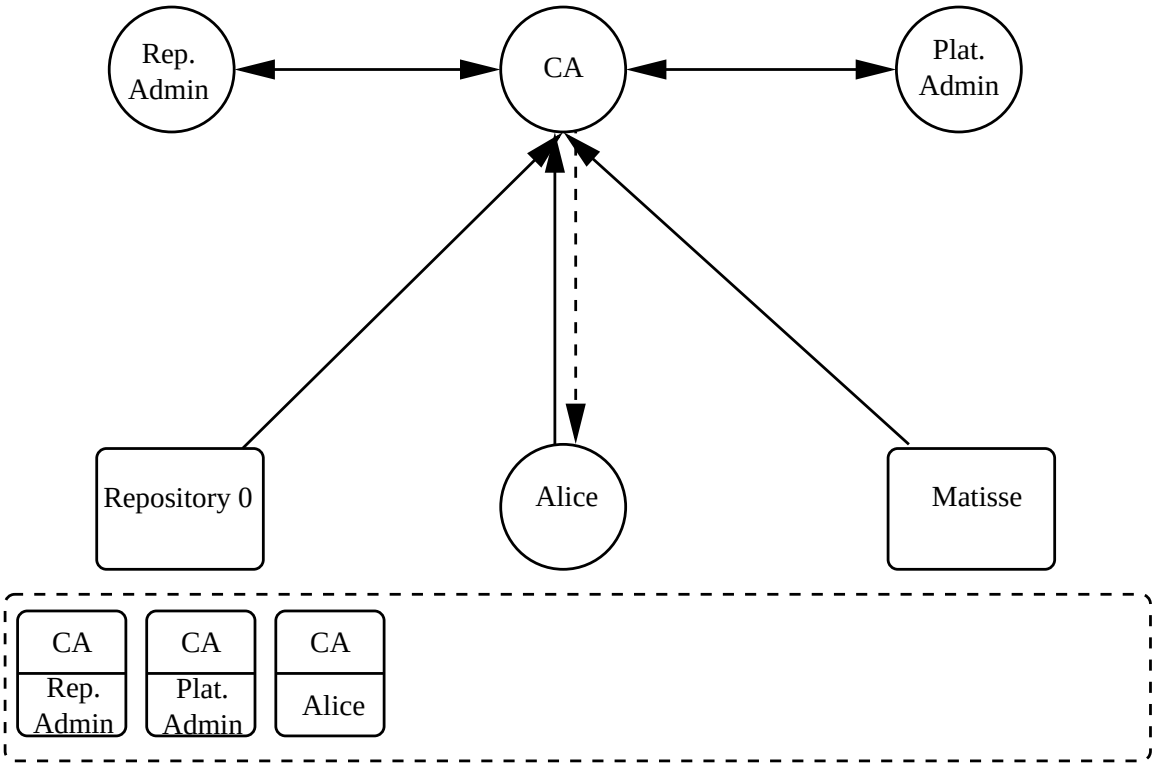

Figure 2: The entities, trust relationships, and initial certificates in SHEMP. The boxes inside of the dashed area represent certificates. In this figure, all three certificates are signed by the CA, and are issued to the Repository Administrator, Platform Administrator, and Alice respectively. In practice, the certificates shown may differ slightly from one another as they represent different sorts of trust relationships. For example, the Platform and Repository Administrator certificates may have the `basicConstraints` X.509 extension set, indicating that they are able to act as CAs themselves. In contrast, Alice's certificate may have the extension set to false, indicating that Alice's certificate is an end entity certificate.

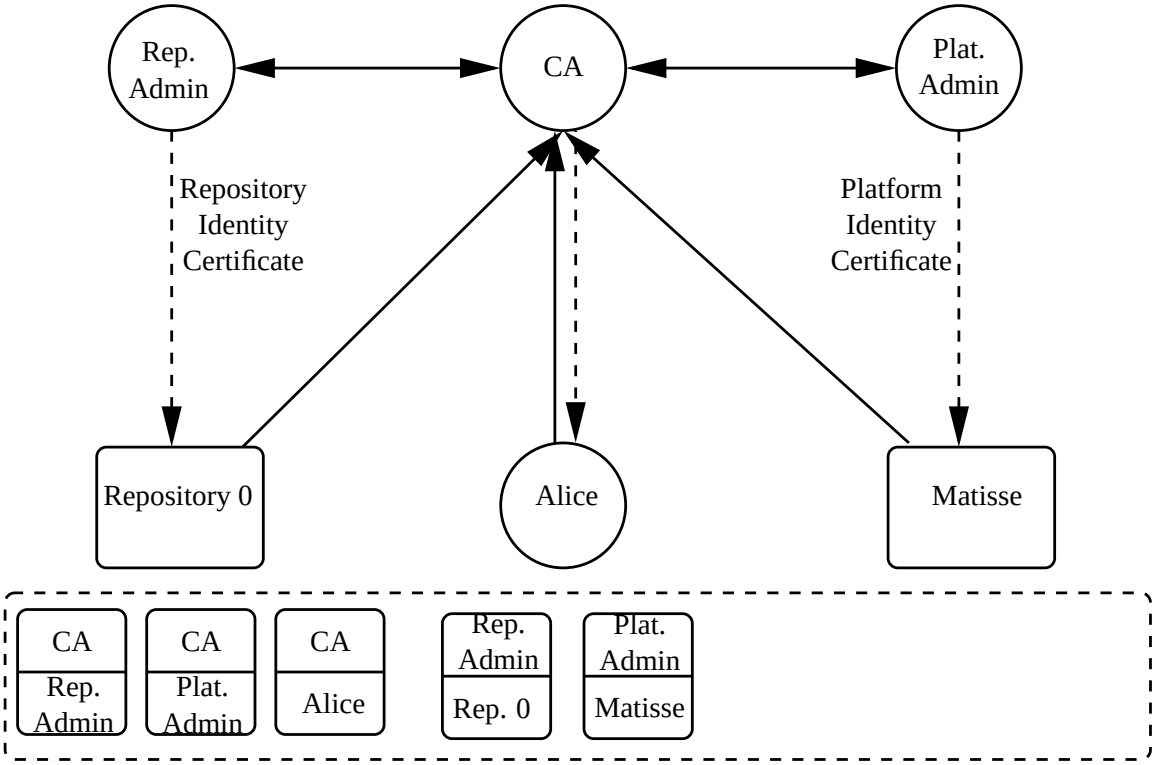

Figure 3: The administrators issue identity certificates to the repository and Matisse. The dashed edges indicate the issuing of a certificate, and the resulting certificates are added to the certificate store.

repository runs on. For instance, if the repository runs in a 4758, then the keypair ought to be generated there, so as not to be compromised. If the repository runs on a Bear platform, then the keypair should be generated inside of the TPM. The idea is to use secure hardware, if available.

Second, the Repository Administrator binds the public portion of that keypair to an identifier for the repository. SHEMP is agnostic about these identifiers. A repository could be identified by a name, a hardware MAC address, the hash of the newly-generated public key, etc. The only restriction that SHEMP imposes is that this identifier uniquely identify the repository. The binding of the public key to the identifier is accomplished via the Repository Identity Certificate issued by the Repository Administrator (depicted as the certificate issued from the Repository Administrator to Repository 0 in Figure 3).

A similar procedure is performed by the Platform Administrator each time a new machine is added to the domain. First, the Platform Administrator generates a new keypair on the platform, using the most secure method available (e.g., a 4758 or a TPM).

Second, the Platform Administrator binds the public portion of the keypair to a unique identifier for the platform. This binding is represented as the Platform Identity Certificate (depicted as the certificate issued from the Platform Administrator to Matisse in Figure 3). As with the repository, SHEMP is agnostic to the specific mechanism used to identify the platform, but administrators should use the "least spoofable" identifier possible. For example, if a TPM is present, the TPM's Endorsement Key could be used, providing a more secure identifier than a hardware MAC address (which is easily spoofed).

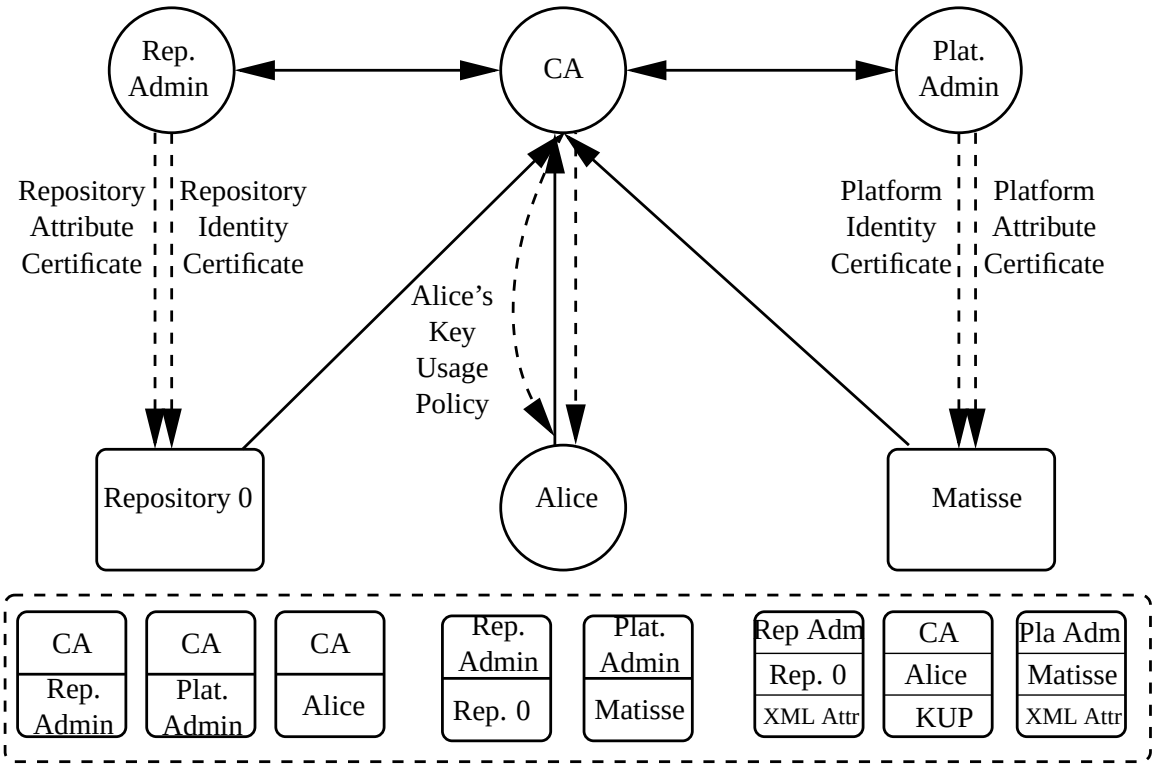

Figure 4: The administrators issue attribute certificates to the repository and Matisse which contains their security level expressed in XML. The CA issues an attribute certificate to Alice which contains her Key Usage Policy (KUP). The KUP is an XACML policy which specifies Alice's delegation policy.

Attribute Certificates Setup The final phase of setting up the system involves issuing attribute certificates to the appropriate entities. These attribute certificates are used to bind the security level of the machines (i.e., the repository and client platform) to the machine's identifier, and to bind a user's delegation policy to the user's identity.

As the Repository Administrator configures the repository, he must also assign some domain-specific security level to the repository. Concretely, the security level is expressed by the Repository Administrator generating and signing some XML attributes for the repository. The idea is for the administrator to make some signed XML statements such as "This repository runs on a Bear platform", "This repository is in a secure location and guarded by armed guards.", etc. These attributes can be arbitrarily complex, and are stuffed into a *Repository Attribute Certificate* (RAC). The RAC is identified by the same identifier that the Repository Administrator used in the Repository Identity Certificate, and thus binds the repository to its XML attributes. The RAC is then signed by the Repository Administrator and placed in a well-known location, such as an LDAP directory. This procedure and the resulting certificate is shown in Figure 4.

The story continues when client platforms are added to the network. As the Platform Administrator configures new machines, she constructs some XML attributes for the platform and signs them. These attributes are expressed in XML, and can state any domain-specific properties that the Platform Administrator feels are important in determining the security level of the machine. Examples may include statements such as "This machine is inside the firewall", "This machine is a Bear platform", "This machine was patched on April 21, 2004", etc.

Like the RAC, these attributes can be arbitrarily complex and are stuffed into a *Platform Attribute Certificate* (PAC). The PAC is identified by the same unique identifier that the Platform Administrator used to identify the platform in the Platform Identity Certificate. Again, machines with no secure hardware may be identified by a hardware MAC address, whereas a Bear platform may be identified by the TPM's endorsement key. In any case, the PAC binds the client platform's identity to its XML attributes. The PAC is signed by the Platform Administrator and is placed in a well-known location such as Dartmouth's LDAP directory. This procedure and the resulting certificate is shown in Figure 4.

The last part of the setup occurs when a user Alice visits the CA for the first time in order to get her identity certificate issued. Alice goes through the standard identity vetting process, eventually proving her identity to the CA/RA.

At the CA, Alice also gets a chance to express her Key Usage Policy (KUP), which governs how her key is to be used. For example, Alice may specify "If my key lives in a 4758 repository, and I request a Proxy Certificate from a Bear platform, grant the Proxy Certificate full privileges. If my key lives in a Bear repository, and I request a Proxy Certificate from any machine outside the firewall, allow my key to be used for encryption only. etc." This KUP is expressed as an XACML policy, and is signed by the CA. The signed KUP is identified by Alice's name (or public key or possibly stuffed into Alice's X.509 identity certificate as an extension—yet to be determined) and is placed into the LDAP along with her identity certificate. Alice's private key is then loaded into the repository (actually, it is most likely generated there and the CA receives a *Certificate Request Message Format* (CRMF) request), and setup is complete (see Figure 4).

The System in Motion Once setup is completed, Alice is free to wander throughout the domain and use her key. For example, assume that she needs to register for classes via an SSL client-side authenticated Web site. Alice begins by finding a computer which is acting as a client (i.e., has the client software installed, and hence has an Platform Identity Certificate and PAC in the directory). For illustration, assume Alice walks up to the client named Matisse.

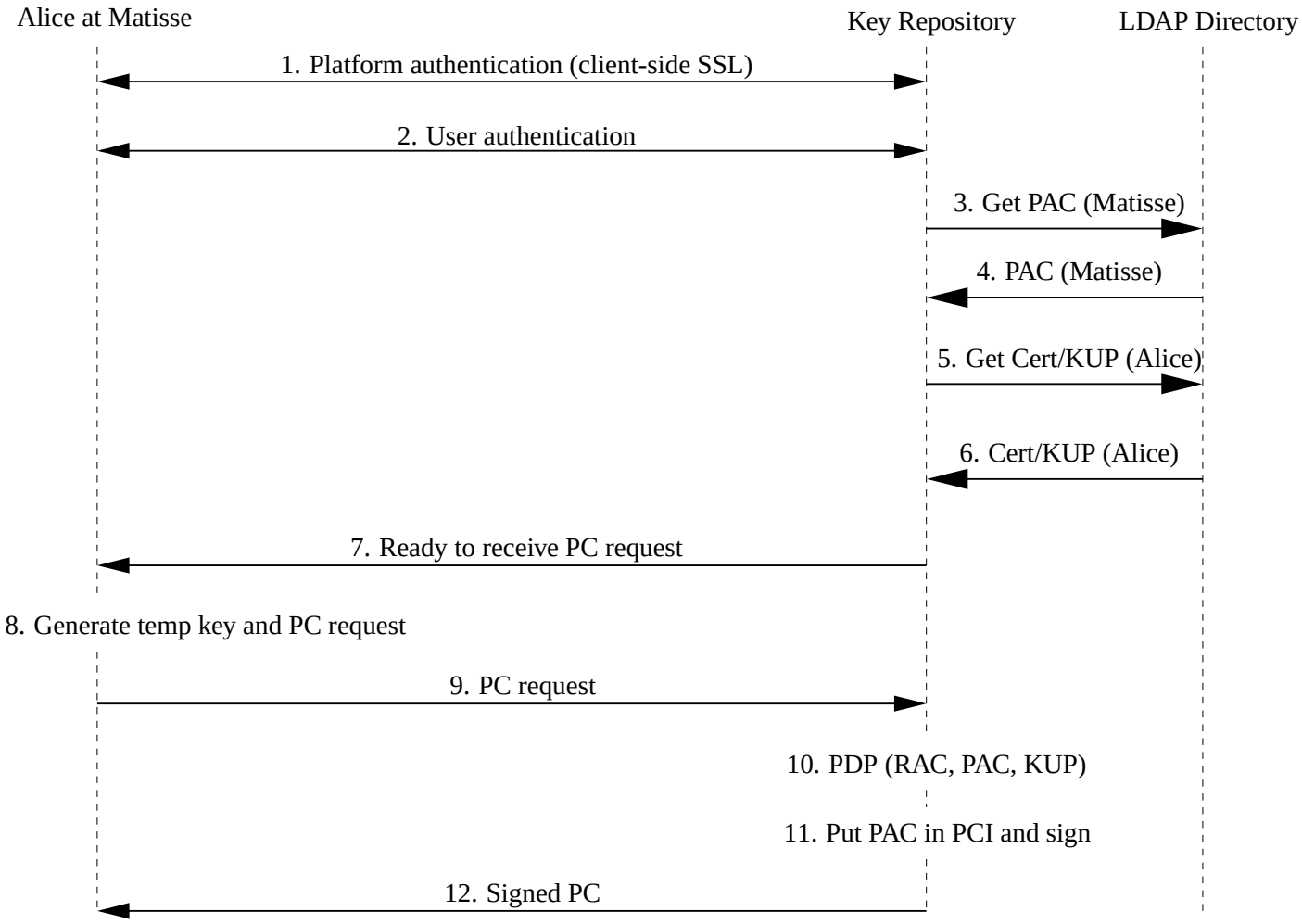

Figure 5: The basic protocol for generating a Proxy Certificate (PC) under the SHEMP system. Steps 3, 4, 5, 6, 10, and 11 represent my enhancements to the basic MyProxy approach. Step 8 also differs somewhat in that under SHEMP, the keypair can be generated in the most secure manner that a client has at its disposal (e.g., a TPM).

The protocol for establishing a Proxy Certificate is shown in Figure 5. Matisse first connects to the repository and establishes a client-side SSL connection. The repository and platform identity certificates are used to negotiate this connection. Recall that the Repository and Platform Identity Certificates are signed by the appropriate administrators (Repository and Platform, respectively), and that the administrators have CA-signed certificates (or a valid chain of certificates back to the CA). The implication is that there is a valid certificate chain from each of the platforms back to the CA. Since both the repository and platform trust the CA, they have good reason to believe the client-side SSL authentication.

The second step is for Alice to authenticate herself to the repository. SHEMP is agnostic with respect to how authentication is accomplished. For prototyping purposes, Alice will use a username/password, but for real security, Alice needs to use a authentication technique which cannot be intercepted by rogue processes on the client. For instance, Alice could use some other keypair (possibly stored on a token) for authentication purposes or she could use biometrics, etc.

Once both Matisse and Alice have authenticated, the repository software uses Matisse's identifier to look up Matisse's PAC. As shown in Figure 5, the repository may also fetch Alice's identity certificate and KUP if it is not locally stored on the repository (possibly to save space on the repository). Once the repository has gathered all of the policy information about the Matisse and Alice (e.g., the PAC, KUP, and Alice's identity certificate), it will acknowledge Alice's and Matisse's authentication, and waits for a Proxy Certificate request from Matisse.

Matisse will then generate a temporary keypair for Alice to use. Again, this may be generated a number of ways depending on the resources available to the client. For example, if the client is a Bear platform, it could generate a keypair in the TPM so that the key will never leave the TPM. If the client is a standard unarmed desktop, it may generate a keypair with OpenSSL [44]. In any event, the client (Matisse) generates an unsigned Proxy Certificate containing the public portion of the temporary key, and sends it to the repository to be signed by Alice's private key.

The repository must then decide if it should sign the request with Alice's private key. The repository takes the security levels of itself and Matisse (contained in the RAC and PAC, respectively) and generates an XACML request containing the attributes. This XACML request and Alice's KUP are then evaluated to determine whether the operation is allowed. Concretely, an XACML PDP running on the repository (as part of the repository software) will make this decision.

If the operation is allowed, the repository will stick the PAC into the Proxy Certificate's PCI extension, and then sign the Proxy Certificate with Alice's private key. Placing the PAC into the PCI allows the Proxy Certificate's relying party to see attributes of the client platform without having to search for them in a directory. (It would also be possible to include the RAC and Alice's KUP into the Proxy Certificate, thus giving all of the information necessary to recreate the delegation decision to the resource.) The signed Proxy Certificate is then returned to Alice.

Instead of presenting her actual certificate to services which require it, Alice now presents her Proxy Certificate which, along with her identity certificate, form a chain: one which includes her real public key which is signed by the CA, and an X.509 Proxy Certificate which contains a short-lived temporary public key, signed by her real private key.

5.2 Analysis

In order to claim victory, I need to show that the proposed system can meet the criteria put forth in Section 3.

Security The precise definition of security in the context of this proposal includes the ability to prevent private key disclosure, as well as minimizing the risk, impact, and window of opportunity for misuse of the delegation key.

The system minimizes the risk of private key disclosure by placing all of the private keys under the control of a trusted entity: the Repository Administrator. A specialist is more likely to protect the private keys than each individual user is. The Repository Administrator can be an organization, and will probably be closely related to the organization which issues certificates (i.e., the CA/RA). Consolidating the private keys to a key repository (or set of repositories) allows an organization to focus its attention and resources on one spot. For instance, an organization can increase security of the private keys by putting the repository inside of a very secure coprocessor (as is often done with the CA's private key), protecting the repositories with armed guards and security cameras, etc.

The system can prevent private key disclosure entirely by storing the keys inside the 4758 or other similar device on the repository. The 4758 provides a tamper-resistant environment to store the keys. Security researchers have subverted applications inside of the device [2], but to date, no one has compromised the device itself. Placing the repository inside of a 4758 keeps the private keys safe even in the event that the 4758's host is compromised.

Further, the consolidation of keys into a repository consolidates the audit trail. Having the repository keep good logs simplifies the amount of data necessary for forensics in the event of a compromise. If the repository is on a machine with secure hardware capabilities, then the log can be cryptographically protected to prevent tampering by intruders.

On the client side, the system provides an improvement over current technologies such as software keystores or USB tokens. The only keys used by clients are short-lived ones contained in the Proxy Certificates. Each Proxy Certificate contains a signed PAC in the PCI extension which relays the platform's security properties to relying parties. Thus, if a specific type of platform (or configuration, or anything else that the domain has deemed relevant) is known to be susceptible to attacks, the relying parties can adjust their level of trust in the Proxy Certificate.

Furthermore, the worst-case scenario (i.e., the authorization to the repository is compromised, perhaps by guessing Alice's password) results in the attacker being able to use a Proxy Certificate issued to Alice for a short period of time (i.e., the validity period of the Proxy Certificate, which is on the order of hours). Additionally, depending on Alice's KUP, the attacker may only have access to a subset of Alice's privileges for that amount of time.

In cases where the client is equipped with secure hardware and the attacker got Alice's key via a software attack, the client platform will detect the attack and not use the key (this can be accomplished on a Bear platform by "wrapping" the key to a specific integrity policy). If the attacker gets the key via some undetectable means, then the window of opportunity for misuse can be rather small.

A very important security feature of the system is its flexibility, which allows it to accommodate a number of environments. This flexibility is realized through the policy framework (i.e., the XACML attributes and KUP), and is generic enough to allow different organizations to define security in a manner which is appropriate for their domain.

Usability Another goal of the system is to make it usable for organizations which deploy it, and developers wishing to extend it. As previously mentioned, the use of a policy framework gives organizations the

freedom to define their own notions of security and make delegation decisions (e.g., whether to issue Proxy Certificates) based on those definitions. For example, an organization could define secure machines as those within a certain range of IP addresses, inside of a firewall, with a certain patch level, with a certain piece of secure hardware, or any combinations of these.

Since the initial prototype will only involve a Bear client and repository, it is natural to assume that developers will want to extend support other platforms (such as the 4758). Accordingly, it makes sense to write the repository in a language which is portable across a number of platforms. The most appropriate tool for this is Java.

The usability of the system is further enhanced by reducing the size of the infrastructure. Concretely, this advantage comes from the significant reduction in the number of entries in the *Certificate Revocation List* (CRL). In current PKIs, when a private key is compromised, an entry is made in the CRL to let relying parties that check the CRL know that the key is no longer valid. The proposed system does not rely on revocation lists. Again, a successful attack on the client just yields access to the temporary delegation key (whose public portion is contained in the Proxy Certificate)—Alice's real private key is still intact, and thus does not need to be revoked. In the event that an attacker captures Alice's authentication information (e.g., her password), he could repeatedly get Proxy Certificates in Alice's name. Once detected, however, there is still no need to revoke, the Repository Administrator can simply change Alice's password on the repository.

Mobility In order for the system to have any practical appeal, it must enable real-world populations. As discussed earlier, these populations are increasingly mobile in terms of geography and across devices. The proposed system can work from any device or location provided that the client is able to authenticate itself to the repository. Alice simply requests a Proxy Certificate to use temporarily from whatever client she is on. Further discussion of this approach and its impact on PKIs is found in Section 6.

While this approach allows Alice to use her key from anywhere within the domain, it also allows Alice to use her key from a wide range of clients while enforcing Alice's KUP. This scheme potentially limits the rights Alice has on lower security devices.

Reasoning For the system to actually work, relying parties must be able to make valid trust decisions based on the chain of certificates they receive. At first glance, it seems that relying parties can do this—if site S receives a chain of certificates from Alice, then Alice was aware of and intended that request to S. However, in addition to prototyping the system described in this proposal, I am proposing to look at this issue in some detail. More specifically, I intend to show that the chain of certificates generated by the system—including the RAC and PAC—is necessary and sufficient for a relying party to make sound trust judgments.

6 Applications

If the proposed solution existed and performed as advertised, I can envision a number of applications which would become possible or would be enhanced. One desirable property of the system is that it serves as a platform for further research and the development of practical applications.

PKI

One interesting application of the proposed system is to make desktops usable as clients in a traditional PKI setting. As discussed throughout this proposal, trusting standard desktops with users' private keys is not a good idea, and using inexpensive hardware such as a USB dongle is no great improvement.

SHEMP takes the user's key off of the desktop entirely. By keeping the private keys centrally located, they can be given the same level of care as the CA receives. The proposed system also allows users to roam throughout the domain and access PKI-enabled applications without having to worry about transporting their private key (in a USB token, for example). Additionally, users and enterprises are able to express their intentions regarding key usage. These intentions, along with statements about the security level of the client and repository, can be used by relying parties to draw sound conclusions about the target (i.e., that they actually are aware of and intended the request).

As discussed in Section 5.2, the worst case scenario for a faulty client is that the user's delegation key can be used for a short amount of time. Damage control under the proposed system in this scenario is much cleaner than in traditional PKI systems. First, the repository contains (possibly secured) audit logs which make forensics easier. Second, the need to revoke keys and deal with Certificate Revocation Lists is eliminated. Revocation in the proposed system is achieved by changing the user's password at the key repository.

PKI Applications Traditional PKI uses of keypairs include encryption, signing, and authentication. The Proxy Certificates generated by SHEMP can by used for any of these operations, although the short lifespan of the Proxy Certificate adds some complexity.

PKI authentication can be accomplished with no modifications to the infrastructure described in this proposal. The short lifespan has no real effect on authentication applications. Proxy Certificates were developed with authentication scenarios in mind (and dynamic delegation which will be discussed below).

For signing applications, the repository (or some other service) may need to be involved in keeping bookkeeping information which can aid relying parties. For example, suppose Alice signs a message with her temporary private key (whose public portion is in the Proxy Certificate) and sends it to Bob. What if Bob attempts to verify the signature after Alice's temporary key has expired? One possible solution is to have the repository timestamp signatures. When Alice's Proxy Certificate expires, the repository generates a new Proxy Certificate (called a *closeout certificate*) which contains the hash chain timestamp on all signatures generated during the time period when Alice's Proxy Certificate was valid. When Bob attempts to verify the signature, he checks to see if the signature's hash value is contained in the closeout certificate covering the time period in which the signature was generated. If so, then Bob can use the public key in the closeout certificate to verify the signature. The details of this scheme will be fleshed out in the thesis; this is a sketch of some initial thoughts.

For encryption, the repository may again need to get involved—this time acting as an "encryption proxy". For instance, assume that Bob wants to encrypt a message for Alice. What public key should he use? One possible solution is to have Bob encrypt the message with Alice's real public key, and send the message to Alice. Upon receipt, Alice gives this message to the repository to decrypt with her real private key. The repository then consults Alice's KUP to see if it should decrypt the message given Alice's current platform. If so, the message is decrypted, re-encrypted with the temporary public key found in Alice's Proxy Certificate, and returned to Alice. She can then decrypt the message with the temporary private key on her current client platform.

Computational Grids

By far, the largest use of the X.509 Proxy Certificate approach is the Grid community [77]. In fact, the X.509 Proxy Certificate standards [74] were drafted by members of this community, and the major Grid toolkit (i.e., Globus [18]) includes Proxy Certificate support.

Furthermore, the Grid community has produced a key repository called MyProxy [42], and has even experimented with adding secure hardware (the IBM 4758) to that repository [30]. Differences between these system and the proposed system will be explored in Section 7, but it is fair to say that the MyProxy system served as a starting point for my design (as did our lab's unpublished AXIS project).

In any case, the key repository and Proxy Certificate paradigms are well accepted in the Grid context, and developing such systems are an active area of research within that community. It is my hope that added security, flexibility, and expressiveness of my solution may be useful in the Grid context.

One large motivation for the Grid community's use of Proxy Certificates is dynamic delegation. In the Grid context, dynamic delegation refers to the process of allowing a Grid user to delegate some subset of their privileges to another entity on relatively short notice and only for a brief amount of time. While dynamic delegation scenarios are seen in a number of places, such as Jini [15] and Greenpass [66], the Grid's delegation framework is needed to allow some process to run as a specific user (Alice) on a machine across the Grid somewhere. Proxy Certificates were developed as a means to give the process some or all of Alice's privileges. The SHEMP system performs such delegation in a similar manner as the MyProxy system. One difference is that SHEMP attempts to take advantage of secure hardware wherever it can find it. Another difference is that MyProxy *allows* policy decisions to be made governing the delegation of privileges, while SHEMP *enforces* Alice's delegation policy (her KUP).

Mobile Clients

As discussed in Section 5.2, the proposed system allows users to move freely between a number of machines and frees them of manually transporting their private key.

Another interesting scenario is one in which Alice has a number of different machines (a "constellation") which all have a need to use the key, but vary in computational power. It is not hard to imagine having a desktop, a PDA, and a Web enabled cell phone. Since all of these devices belong to Alice, she should be able to use her key from all of these devices. Potentially, Alice's KUP could limit the temporary key to performing only certain operations (e.g., encryption) on power-constrained devices.

The use of a policy framework (instead of a rigorous policy language), allows policies to be constructed which use the computational power (or anything else) as a determining factor in delegation decisions (as opposed to or in conjunction with security level).

In this area of power constrained devices, schemes such as SWATT [62] enable a form of attestation which can be used to identify a platform. The system's flexibility regarding authentication mechanisms allows it to be extended to new attestation schemes as they become available.

7 Other Solutions

There are a number of other systems in existence which attempt to solve the same problems as the solution outlined in this proposal. One major difference is that few of them rely on or consider secure hardware to operate. Nevertheless, this section will give an overview of such systems.

MyProxy The closest system to the proposed system is the MyProxy credential repository [42]. The concept of the MyProxy is quite similar to the proposed system: allow users' private keys to be stored in a central location and have the repository use those private keys to sign Proxy Certificates. Some researchers have even investigated placing the private keys inside of a secure device (i.e., the IBM 4758) [30].

There a few major differences between these approaches and the proposed solution. Most notably is the lack of support for secure hardware at the clients. The proposed system is flexible enough to utilize the secure hardware available on the client's machine for identification and temporary key generation and storage purposes.

Additionally, the security level of the client machine (perhaps expressed as a function of the client's secure hardware) is relayed to the relying party so that the relying party can make a more informed judgment on whether to trust Alice's Proxy Certificate (and the private key it represents). In this light, the proposed system is more expressive than the MyProxy and hardened MyProxy systems.

Last, the hardened MyProxy system only placed the users' private keys in secure hardware, as opposed to the entire system. The IBM 4758 is a general-purpose computing device, and thus could actually run the repository software itself. The hardened MyProxy project [30] only places the private keys in the coprocessor, leaving other sensitive components (such as the password file) on the host. The proposed solution places the entire repository software (and all of its components) under the protection of secure hardware. Doing this allows clients to communicate directly with the repository software inside of the secure coprocessor, thus eliminating the proverbial "armored car to a cardboard box".

Kerberos Kerberos is an authentication framework based on secret key cryptography. A good overview of the IETF standards and workings of Kerberos can be found in the literature (e.g., [26, 58]). Very briefly, Kerberos consists of a *Key Distribution Center* (KDC) and a set of libraries which applications must use to authenticate clients. The KDC holds a *master key* which is a shared secret between it and each party in the system. When parties would like to communicate, the KDC generates a short-lived shared secret (i.e., a session key) and distributes that to the principles.

The first notable difference between Kerberos and the proposed solution is that Kerberos is mainly an authentication framework, whereas the proposed solution could be used to more as a traditional PKI (i.e., to enable secrecy and signature applications such as S/MIME as well as authentication). Further, Kerberos relies on shared keys instead of public key certificates.

Second, there is a lack of dependence on secure hardware in Kerberos. Some research has looked into placing the KDC inside of a secure coprocessor [24], but there is nothing in the Kerberos standard about this. Additionally, there is no discussion of hardening clients with secure hardware. Since Kerberos stores credentials in memory, it is susceptible to the attacks listed in previous work [34].

Sacred There is a working group within the IETF which is working on protocols for "Securely Available Credentials" (Sacred). To date, they have established requirements [53], a draft describing a framework [51] and a protocol [52]. The protocol describes two different methods for a user to transfer his private key from one device to another. Which protocol to use is based on the relationship between the two devices: peer-to-peer or client/server.

While the Sacred project shares some of the same goals as this thesis (the secure use of credentials), there are some key differences. First, the philosophy of this thesis is to leave the keys in one place and use them (via delegation and certificate chains) from many different locations. Sacred suggests moving the credentials from device to device as the user moves.

The second major difference is the reliance on trusted hardware. The protocols that the proposed solution employs takes advantage of the fact that the hardware can speak about its configuration. The Sacred protocols are more general-purpose, and hence do not rely on or compensate for the presence of specialized hardware.

PubCookies/Passport Web servers can establish longer state at a browser by saving a *cookie* at the browser. The server can choose the contents, expiration date, and access policy for a specific cookie; a properly functioning browser will automatically provide the cookie along with any request to a server that satisfies the policy. Many distributed Web systems—such as "PubCookies" [50], and Microsoft's Passport [38]—use some authentication mechanism such as a password to initially authenticate the browser user, and then use a cookie to amplify this authentication to a longer session at that browser, for a wider set of servers.

There are many differences between using single-sign-on cookies for authentication and the proposed solution. First, cookies are application-specific mechanisms only usable by Web browsers. The framework described in this proposal could be used by a number of applications, including but not limited to Web browsers.

Second, flaws in the browser can subvert the security of this approach [19, 75]. In the proposed solution, the application is not able to give away any secrets as it is the underlying hardware which is providing security. Cookie-based authentication in general does not rely on such hardware for security.

Shibboleth Shibboleth is an Internet2 project which aims to develop a middleware that supports user authorization and authentication. Nazareth [39, 40] gives a good overview of Shibboleth and its components, but the general idea is to allow Alice (from institution A) to access resources at institution B in such a way that preserves her privacy. This is accomplished by institution A issuing an anonymous handle to institution B, which it in turn uses to fetch attributes about Alice from the Attribute Authority at institution A.

A number of differences exist between my proposed solution and Shibboleth. First, Shibboleth is not a general-purpose PKI solution, as it requires the server (at institution B in the above description) to be equipped with components of the Shibboleth system (e.g., the Shibboleth Indexical Reference Establisher, and the Shibboleth Attribute Requester)

Second, there is no mention of secure hardware. The protocols used to establish handles and acquire attributes are all specified by Shibboleth, and none of them take advantage of secure hardware's ability to attest to the configuration of the machines.

Greenpass The Greenpass project [66] is a delegation framework which uses SDSI/SPKI delegation on top of X.509 keypairs in order to authorize guests to Dartmouth's wireless network. The idea is for a member of the population (Alice) which holds an X.509 certificate to delegate (i.e., by singing a temporary SDSI/SPKI certificate) to her friend Bob so that he can access Dartmouth resources during his stay (even though he is not a member of the Dartmouth population).

While there is quite a bit of similarity in the delegation framework, there are a number of important differences. As with the others, there is no mention of secure hardware in Greenpass, and thus the protocols used in the delegation do not take the machine's configuration under consideration.

Second, there are no constraints on where users store their key under the Greenpass system. Theoretically, users could store them on their desktop or on a USB token of some sort, and as stated numerous times throughout this proposal, this is not a good idea.

Online Trusted Third Parties One school of thought is to not only house the users' private keys on a remote server, but to actually perform the key operations on that server. The entity responsible for performing such operations is sometimes referred to as an "Online Trusted Third Party" (TTP) [48]. Variations on this idea involve letting the user own part of their key and allowing the TTP to own the other part via some threshold cryptography scheme. These TTPs are sometimes referred to as "Semi-Trusted Mediators" (SEM) [4].

Philosophically, the proposed solution and these TTPs and SEMs share the vision that the key should be removed from the desktop. SEMs typically split the user's private key and place a share of it in the SEM. Recent work here at Dartmouth has explored adding secure hardware to SEM [76].

There are a number of differences between SEM and SHEMP. First, SHEMP places the entire key inside of a repository, as opposed to a portion. This results in the repository becoming a "fully-trusted mediator". Second, the use of Proxy Certificates allow some operations to be accomplished without the aid of the repository (such as authentication), whereas SEM requires the mediator to be involved in all private key operations. Third, SHEMP uses a policy framework to make delegation decisions, perform key operations, and enforce users' KUPs. Last, SHEMP strives to allow for a number of different client and repository platforms with varying levels of secure hardware.

8 Summary

This proposal began by describing a practical problem which we discovered, namely, that desktop computers are not usable as a PKI clients. The TCB is too large and a very small amount of malicious code running with user privileges can effectively give an attacker access to a user's private key. Furthermore, current systems generally do not allow for users to be mobile or use keys across multiple devices.

Common solutions to these problems typically involve a token which the user uses to carry their private key (such as a USB token). In previous work, we have shown experimentally that such devices do not really solve the problem either.

I believe that the Grid community has developed an approach to the problem which is worth further investigation: online credential repositories. By enhancing the online credential repository with trusted computing—at the client as well as the server—and giving relying parties more information about the client,

I argue that the desktop can finally be useful as a PKI desktop client.

8.1 Evaluation

I propose to evaluate the SHEMP architecture by executing the following steps:

1. I will begin by formally defining (using Maurer's notation [35]) properties that a successful architecture must exhibit. Most likely, this will involve defining a set of statements that the relying party Bob must be able to deduce in order to trust Alice and believe that her private key and the private key described by her Proxy Certificate is authentic. The formal statement adds to the solution criteria put forth in Section 3; it is not meant as a replacement.

2. I will then develop an architecture which, when formally modeled using Maurer's notation (possibly with my own extensions), exhibits the properties set forth in the previous step. The architecture presented in Section 5 is the first candidate for such formal analysis.

   Concretely, this step involves a formal look at what kind of trust judgments a relying party can make upon receiving Alice's 1) Proxy Certificate, 2) identity certificate, 3) PAC, and 4) RAC. I intend to answer the relying party's question: "Given Alice's Proxy Certificate, and assuming that the repository enforced Alice's KUP, do I have any reason to believe that Alice intended and is aware of this request?"

   The formal statement that the SHEMP architecture meets the formal criteria outlined in the previous step will be treated as the thesis's main theorem.

3. I will then implement the SHEMP architecture in the Java programming language on a pair of Bear platforms. The choice of the Bear platform gives some level of secure hardware and allows me to use a number of existing tools such as the Globus's toolkit, Sun's XACML implementation, etc. It also allows me to use a language (Java) which contains a wealth of libraries, thus simplifying tasks such as parsing certificates, establishing SSL connections, etc.

   Prototyping the system will serve to make many design ideas concrete and give a means for evaluating constructs which cannot be easily formally modeled. For instance, actually implementing the XACML policy framework will give me a chance to make and maintain sample policies, thus giving a means to evaluate the framework even though it cannot be formally modeled under Maurer's notation.

   Furthermore, prototyping the decryption and signature verification schemes described in Section 6 will validate the Proxy Certificate-based approach for traditional key usage scenarios (i.e., encryption and signing).

   There is also a possibility of plugging the resulting prototype into Dartmouth's emerging Grid implementation, using it as a replacement for the MyProxy credential repository. Such an effort would allow me to observe how the system operates in a real-world environment. Concretely, this could be accomplished by writing a client API which is a superset of the MyProxy Java client API found in the Grid's CoG toolkit (the API will include interfaces to decrypt and verify signatures). This way, legacy Java-based MyProxy clients can use SHEMP without any code change, and Grid application developers do not have to learn a new API to use SHEMP.

Deliverables and Road Map

- *Proposal.* I intend to give the official proposal in front of my committee by June 2004.

- *Formal Reasoning.* The formal statement of the design criteria and the application of Maurer's notation to the architecture will be completed by July 2004.
- *Development.* I estimate that the prototype will be complete by October, 2004.
- *Writing and Defense.* The thesis will be written and I intend to defend it by January, 2005.

Appendix A: Evaluation

In order to show that SHEMP is an interesting system from a research perspective, I need to show that it is feasible, practical, and can be used to build real applications. This means that I need to demonstrate that SHEMP offers security, mobility, and flexibility and that the complexity of these features do not overwhelm users (end users as well as application developers).

9 Evaluation Against Criteria

My thesis proposal sets forth a set of criteria that defines the notion of what it means for a desktop to be "usable as a PKI client." My claim is that any solution which claims to be usable as a PKI client should:

1. Minimize the risk/impacts of key disclosure
2. Allow for client mobility
3. Be usable to application developers and users
4. Enable relying parties to make reasonable trust judgments

I then make the claim that the SHEMP system meets these criteria. This section elaborates on the criteria and outlines my strategy for supporting this claim.

9.1 Minimizing risk of key disclosure

The notion of minimizing risk can be broken down into a number of subpoints. Since quantifying risk (and thus minimization of risk) is difficult, support for this claim will come from analysis backed by implementation.

- SHEMP reduces risk of key disclosure by taking private keys off of the end-user machines (or devices such as USB tokens) and taking advantage of secure hardware on the end-user machines. SHEMP offers an improvement because end-user machines are susceptible to attacks which either disclose the private key or allow the attacker to use it at will—we have shown this experimentally in the Keyjacking work (which is part of this thesis). I plan to demonstrate (with real software) how SHEMP can defend against Keyjacking-style attacks, and explain how both the status quo and the MyProxy solution cannot defend against such attacks.

- SHEMP reduces risk by getting keys out of the end-users' control and placing them in the care of a specialist. As a way to validate this claim, I intend to compile data from Dartmouth's campus-wide USB dongle rollout slated to occur this fall with the incoming freshmen class (this exemplifies keys in the end-user's control) as well as Dartmouth's Kerberos implementation (which is an example of the repository approach—I will also include analysis of Green Grid's MyProxy, if it is ready in time). Specifically, I will compare the number of dongle-resident keys revoked due to loss or theft or whatever against the number of Kerberos Master Secrets revoked during the same time period (fall term '04).

Minimizing the impact of disclosure is also a difficult claim to quantify. As with the "minimizing risk" claim outlined above, I will break this claim down into subpoints and provide analysis and demonstration of each one.

- As with MyProxy, SHEMP minimizes the impact of private key disclosure at the client by allowing the key to be used for a short time. The Keyjacking part of this thesis showed that key disclosure is disastrous within the current client-side infrastructure. In many cases, an attacker is able to obtain the private key itself, or use it for arbitrary operations for an indefinite period of time. Under SHEMP (and MyProxy), the key issued on the client's desktop is valid for a number of hours. I plan to demonstrate in the prototype that SHEMP's default behavior is to issue temporary keys with a short lifespan (on the order of hours), and compare the size of an attacker's window of opportunity under SHEMP against the window size under the status quo.
- SHEMP also minimizes the risk of key disclosure through the use of the environmental attributes (RAC and PAC) and user policy (KUP) found in each Proxy Certificate's PCI extension. While MyProxy allows any arbitrary policy statement to be placed in the PCI extension, SHEMP mandates that the policy be included, and that the policy is expressed in the XACML policy syntax. This approach gives useful information to relying parties, allowing them to adjust their trust in the client based on the environment. Relying parties are thus aware when clients generate temporary keys under conditions which are likely to result in key disclosure, and have the possibility to limit their use. The policy framework will be demonstrated in the prototype, and the usability of the policy statements in the context of building a real application will be examined in case studies (described in Section 9.3).
- As with MyProxy, SHEMP minimizes the impact to the organization in the case of a key compromise. In the status quo, compromised keys are revoked by placing their certificate into a CRL or OCSP server. Keeping CRLs up to date and distributing them are non-trivial problems in the PKI space. Assuming that the key compromise occurs in the same place (i.e., the desktop) SHEMP (and MyProxy) revokes keys by changing the authentication information at the repository, thus reducing the amount of work for IT staff. I plan to gather evidence for this from interviewing Dartmouth College's CA, and comparing the current CRL approach to SHEMP's approach. I am particularly interested in the usability argument from the CA's perspective and the amount of time that a revocation takes to propagate in both systems (as this defines a potential window of opportunity for an attacker).
- As with MyProxy, SHEMP minimizes the impact of a key compromise by consolidating the audit trail used for forensics in the event of a key compromise. Additionally, since the SHEMP repository software can run inside of secure hardware (as will be demonstrated in the prototype), SHEMP can secure its logs inside of the hardware—MyProxy cannot. I plan to show the impact minimization by comparing the size of the audit trails (in terms of number of logs to sift through for forensics) for the various approaches, as well as a security analysis of the logs themselves (i.e., how well they are protected against tampering from an intruder).

9.2 Allowing for client mobility

The idea behind client mobility is to allow users to use the system from various locations, possibly on different platforms with different security properties. Concretely, proper design choices (such as using portable languages, etc.) can ensure that SHEMP allows clients to move across machines, platforms, and devices, and the SHEMP prototype will demonstrate these abilities.

More subtlety (and more importantly), it is important to show that SHEMP remains secure in scenarios involving mobile clients. Our previous experiments have illustrated the danger of simply placing the key on a device (i.e., a USB dongle) and allowing users to move. I plan to offer an analysis of these earlier results and compare them to results of a similar experiment run against SHEMP.

9.3 Being usable to application developers and users

So far, this document has outlined my plan to show that SHEMP can achieve security and mobility. As with most (if not all) systems which try to provide security, the mechanisms which make SHEMP secure can also make it hard to use. One way that the security and flexibility of SHEMP is achieved is through the use of SHEMP's policy language. In order to show that SHEMP is usable to application developers and users, I need to show that application developers and administrators (users will likely have an administrator (e.g., the CA) construct policies on their behalf) can understand and construct valid policies to solve real security problems. Usability in this context means that the policy mechanism must be a valid medium for developers and users to express their mental models.

I intend to show that the policies are usable through the use of a number of case studies. The case studies will outline real applications (taken from Dartmouth's Grid community) which could use SHEMP. Once the applications are designed, I will give the design to a small group of subjects who would likely fill the roles of the Repository Admin, the Platform Admin, the CA, and users. I will be interested in evaluating whether the parties can generate a meaningful set of policies which represent a given mental model, how long it takes them, and their feedback regarding the difficulty of their task. I will not implement these applications, as that falls into the realm of software development (although one sample application I will develop is discussed in Section 10) and does not offer much in the way of evaluating the usability of the policies themselves.

Concretely, evaluating the policy for these applications involves having the Platform Admin assign attributes to a number of machines which represent different security levels, having the Repository Admin assign attributes to a repository, and having the CA construct a KUP which accurately represents Alice's wishes. Then, some of the subjects will act as application developers and generate a specific policy governing their application. Once complete, subjects will be asked to predict the outcome of a number of operations which make use of the policies that the subjects just generated. Requests will be made and the policies will be run through an XACML PDP to see if the request should be allowed according to the application policies as well as the KUP (one important feature of XACML is that it allows for the combining of policies—such as the KUP and the application's policy—when making access control decisions). The predictions will be compared to actual results in an effort to see how well the system represents the subjects' mental models.

There are a number of potential applications which arose after meeting with the Greed Grid developers/administrators, and these applications will be used as the initial case studies.

All of the applications would use secure hardware on the client in a similar manner. The PC's private key would live inside of the hardware device if there is one present. Depending on the hardware, this scheme

could render the PC's private key unusable if specific conditions were not met on the client (e.g., the OS or client application is not trusted). Additionally, the secure hardware can be used to uniquely identify the client machines.

Case Study I: UberFTP The first possible application which could benefit from SHEMP is the UberFTP program. UberFTP currently supports the use of Proxy Certificates for authentication. Extending the system to use SHEMP would involve making the server aware of the policy and attributes included in the PC's PCI extension, and using that information to make access control decisions.

A SHEMP-enabled UberFTP server could possibly export different portions of the ftp directory to clients based on their environment. For example, the server may not make non-anonymized medical data available to clients coming from public terminals. It may also not allow the `PUT` operation to be performed from anyone on public terminals.

Case Study II: GSI Enabled MySQL A second application which would benefit from SHEMP is the GSI Enabled MySQL database. Apparently, a large problem in the Grid space involves restricting access to database information. SHEMP could be used to restrict subsets of data and/or operations (e.g., the `JOIN` operation) to parties operating within specific parameters. For example, a database of medical information way wish to hide any record's `NAME` field to any client asking from a machine that is not in the hospital due to HIPPA regulations.

Case Study III: GSSKLOG The GSSKLOG application is used by Grid installations (such as Green Grid) to allow users to obtain AFS tokens. One possible place that SHEMP can enhance the service is to restrict the circumstances under which the service will hand out administrator tokens. For example, if Alice the AFS Admin requests an admin token from her desk, the service will grant it, but if she attempts to get administrative privileges from a machine outside of the firewall, the service will deny the request.

Case Study IV: Grid Job Management Another application of interest would involve a higher level Job Management application which Grid users can use to gather statistics about their computations. For example, Dr. Bob submits a large set of brain scans to the grid for some crunching application to chew on for the next few days. Now, when Dr. Bob leaves his office and has lunch in the cafeteria, he may want to check the progress of his job, but since he is on a public terminal, he should not be able to get any results which show a correlation between name and result—-he should just get aggregate results. When he goes to the hospital, and has a break, he might be able to tinker some parameters of his test as well as see aggregate date, but he is still not able to get results linking name and result, only when he returns to his desk can he get that information.

Case Study V: Chiron The Grid community relies on "virtual data" for much of its computational work. Some data is not gathered from measurements directly, but is derived from other data through the application of computational procedures. "Virtual data" is an explicit representation of such computational procedures.

The Chiron system is under development by Grid staff at the University of Chicago, Argonne National Labs, and Dartmouth College. The Chiron system is a "Virtual Data Grid Portal", which allows applications and users to access virtual data. Chiron allows users to manage user accounts, publish data to the Grid,

configure Grid resources, as well as a number of other applications which are beyond the scope of this document. SHEMP could be used to enhance these tasks by only allowing certain operations (on accounts, data, applications, etc.) to be performed under certain circumstances. For example, Charlie may not be able to change his password unless he is on his own desktop machine.

9.4 Enabling relying parties to make reasonable trust judgments

As stated in the thesis proposal, all of the above properties are great, but if the system cannot enable relying parties to make reasonable trust judgments then there is no point to the system. To clarify, I am not concerned with issues regarding UI development or the standard concerns of the Human-Computer Interaction Security community. I am interested in investigating whether SHEMP allows relying parties to conclude what they ought to conclude. Thus, the ability to enable such trust judgments should be viewed as a correctness condition for SHEMP. Using an extended version of Maurer's calculus (with my own extensions which allow for the modeling of authentication and time), I plan to formally state and prove SHEMP's correctness condition.

10 Putting it all together: a real application

While analysis and proofs are good tools to reason about many aspects of the system, putting the pieces together and building a real application is a critical part of claiming that SHEMP "can be used to build real applications." Since the primary motivation for SHEMP is to "make desktops usable for PKI", I will build and evaluate an application which uses SHEMP for the basic PKI operations (encryption/decryption and signing/verification).

The primary goals of this exercise are to ensure that all the pieces of SHEMP work together, and to evaluate the performance of SHEMP. While performance is not the most interesting part of this research, if the system is so slow that it is perceived as unusable, then there is no point. One nice auxiliary fact about this exercise is that it is exploratory in nature—no one uses Proxy Certificates for encryption/decryption or signing/verification. Any results in this space are original results in themselves.

The following brief use cases will serve as the starting point for conceptual-level designs for the application. These are sketches, and are not meant to be read as complete design documents.

Use Case I: Encryption/Decryption

1. Alice sits at a Bear machine in her office, requests a PC from a repository (which houses her long-term private key), and receives one which allows her to perform encryption (the temporary private key on her machine is "sealed" so that if the Enforcer detects a violation of the integrity policy, the TPM's PCRS are scrambled and the TPM will not release the key for use).
2. Next, she receives a message from Bob, and Bob has encrypted the message with the public key in her long-lived ID certificate. In order to decrypt the message, she needs to contact the "encryption proxy" service which is running on the repository (because it will need to access her long-term private key which lives in the repository). The encryption proxy fetches the environmental information from Alice's current PC's PCI extension (policy and attributes), and does a check to see if it should decrypt

messages under the current environment. If so, the message is decrypted with her real private key on the repository, re-encrypted with her current PC's public key, and returned to Alice.

One distinct advantage of this approach is that the plaintext is only exposed at the key repository. Since the repository application (as well as the encryption proxy service) runs inside of secure hardware, the plaintext is only exposed inside of secure hardware. Depending on the strength of the hardware, this approach keeps the plaintext safe, even if the machine hosting the repository has been compromised.

3. If Alice's platform has not been tampered with in a way which Bear can detect, then Alice can use the short-lived TPM private key to decrypt the message.

4. Now, if Alice goes to the library and uses a machine without any secure hardware, then (if she has constructed her KUP correctly) she does not want her PC to be used for the re-encryption because she does not want to expose plaintext on a library machine.

Use Case II: Signing/Verification

1. Again, Alice sits at a Bear machine in her office, requests a PC from a repository (which holds her long-term private key), and receives one which allows her to perform signing (the short-term private key is "sealed" so that if the Enforcer detects a violation of the integrity policy, the TPM's PCRS are scrambled and the TPM will not release the key for use).

2. Next, she wants to sign a message and send it to Bob. Since the short-term private key on Alice's machine will expire shortly (meaning that Bob will not be able to verify her signature after expiration), she needs to contact the "timestamping service" and enlist its help. The timestamping service will first check Alice's KUP to see if her environment and policy allow her to generate signatures. If so, it will record the time that the signature was generated. When Alice's PC expires (a number of hours after it was generated), the service will generate a "closeout certificate" containing the hashchain timestamp on all of the signatures generated within the timeframe that her PC was valid and the public key of the expired certificate.

3. When Bob wants to verify Alice's signature, he fetches her closeout certificate and checks to see if the signature is contained in the closeout certificate. If so, then the public key in the closeout certificate can be used to verify the signature. If an adversary has access to the short-term private key on Alice's machine after it has expired and attempts to sign something, the adversary's signature will not be found in the closeout certificate. Thus, Bob will not be able to verify such a signature.

Alternate Use Case II: Signing/Verification via Proxy

1. Alice sits at a Bear machine in her office, requests a PC from a repository (which holds her long-term private key), and receives one which allows her to perform signing (the short-term private key is "sealed" so that if the Enforcer detects a violation of the integrity policy, the TPM's PCRS are scrambled and the TPM will not release the key for use).

2. Next, she wants to sign a message and send it to Bob. Since the short-term private key on Alice's machine will expire shortly (meaning that Bob will not be able to verify her signature after expiration), she needs to contact the "signing proxy" and enlist its help. Alice first signs the message with the temporary private key on her machine. She then sends the message to the signing proxy which is

running on the repository (because it needs access to Alice's long-term private key). The proxy service will make a log entry that Alice is trying to sign something and check Alice's KUP to see if Alice's current environment allows her to generate signatures. If so, the service will verify Alice's signature generated with the short-term key, separate the message from the signature, sign the message with Alice's long-term private key, and return the message to Alice. The end result is that the message is signed by Alice's long-term key if the current environment and Alice's KUP allow it.

3. Now, when Bob wants to verify the signature, he uses Alice's long-term public key (as he would normally do). This way, Bob can verify the signature even after Alice's PC has expired. This approach allows SHEMP-users and non-SHEMP-users to securely communicate without non-SHEMP-users having to install more software or learn new techniques; only SHEMP-users have to deal with the proxies.

As stated earlier, performance is not the most interesting aspect of this application, but since a third party is contacted for all operations, I expect some slowdown. To this end, I will run performance tests to see how much overhead is introduced when a third party is involved in the key operations. The baseline will be a simple Java application which performs the cryptographic operations using a locally-stored keypair (as opposed to the Proxy Certificate approach) on the local machine (i.e., without third-party involvement).

11 Shrinking the TCB

Some of the feedback I received from the committee was regarding the concern that SHEMP was simply gluing MyProxy, secure hardware, and some policy framework together. The committee wanted me to explain the "deeper" and "non-obvious" aspects of SHEMP. I realize that this analysis is not the same type of evaluation outlined in Section 9 and Section 10, but it is worth discussing.

The Keyjacking results suggest that the reason current desktops are not usable as PKI is that the system designers operated under the assumption that the entire desktop is trusted. The implication is that the Trusted Computing Base for modern PKI clients includes the entire desktop—the OS (which has more code than the space shuttle) and many applications (some of which are inseparable from the OS). The experiments in the Keyjacking portion of this thesis show that when one malicious executable running with user privileges is introduced into the TCB (i.e., runs on the desktop), the security of the entire system is undermined. Additionally, the experiments show that it is difficult (if not impossible) for users to generate accurate mental models of how, when, and why their private key is being used.

SHEMP solves these issues by shrinking the TCB and giving users a medium to express their key usage desires in a way that is applicable to their domain (as opposed to the "low", "medium", and "high" Microsoft key policy choices), thus allowing users to construct valid mental models and relay them to relying parties and applications.

First, SHEMP gets the keys off of the desktop altogether, and places them in safe place: the repository. Not only does the repository protect the keys themselves, it protects the repository application. Since the repository application is orders of magnitude smaller than a general purpose OS (and applications which are tightly coupled to the OS), SHEMP greatly shrinks the TCB.

Second, although humans do not directly contribute to the TCB in the standard sense, placing the keys under the jurisdiction of one entity (the Repository Admin) versus an entire user population also serves to reduce the number of entities that must be trusted in order for the system to function and remain secure.

Third, SHEMP reduces the amount of trust that must be placed in the platforms which use the keys (i.e., the desktops). The SHEMP administrators (the Platform and Repository admins) vouch for the trustworthiness of machines, as well as describe the methods by which relying parties can verify the trustworthiness themselves (i.e., via the use of secure hardware "challenge" protocols). Assuming the hardware is designed and implemented correctly, such challenges can never be used to overstate the security properties of the device, giving relying parties a verifiable upper bound on the trustworthiness of a specific platform.

Putting the pieces together illustrates a system which has a significantly smaller TCB that the status quo. The private keys no longer live on (or directly interact with) big, bloated desktops that are typically cared for by end-users and have 6 critical security patches released in a single day. They now live in a secure place, are used by one application, and are cared for by a specialist. Furthermore, when a user makes a request, the relying party no longer has to trust the requester's machine and wonder if the request was intended. Under SHEMP, the relying party has been given the environmental attributes and policy in the PC. These allow the relying party to decide for themself how much they should trust the requester's platform, and whether the requester really intends their key to be used for this type of request in the first place—thus enforcing the user's mental model.

12 Summary

This document has attempted to outline my strategy for evaluating SHEMP on an analytical level (Section 9), a practical level (Section 10), and a philosophical level (Section 11). Some of this work has been completed (such as a number of the experiments developed in Keyjacking, as well as some of the design and coding of SHEMP), but the synthesis of the pieces is not yet complete. My hope is that once construction is complete, the analysis outlined in this document is sufficient to show that SHEMP is a real solution to a real problem: desktops are not usable as PKI clients.

Acknowledgements

This research has been supported in part by the Mellon Foundation, NSF (CCR-0209144), AT&T/Internet2 and the Office for Domestic Preparedness, Department of Homeland Security (2000-DT-CX-K001). This paper does not necessarily reflect the views of the sponsors.

References

[1] R. Anderson. TCPA/Palladium Frequently Asked Questions. `http://www.cl.cam.ac.uk/users/rja14/tcpa-faq.html`.

[2] R. Anderson and M. Bond. API-Level Attacks on Embedded Systems. *Computer*, October 2001.

[3] M. Blaze, J. Feigenbaum, J. Ioannidis, and A.D. Keromytis. The KeyNote Trust-Management System Version 2. IETF RFC 2704, September 1999.

[4] D. Boneh, X. Ding, G. Tsudik, and C. Wong. A method for fast revocation of public key certificates and security capabilities. In *10th USENIX Security Symposium*, pages 297–308. USENIX, 2001.

[5] Burrows, Abadi, and R. Needham. A Logic of Authentication. Technical Report 39, DEC, 1990. SRC Research Report.

[6] D. Chadwick, A. Otenko, and E. Ball. Role-Based Access Control with X.509 Attribute Certificates. *IEEE Internet Computing*, March-April 2003.

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

[8] C. Ellison. SPKI Requirements. IETF RFC 2692, September 1999.

[9] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B.M. Thomas, and T. Ylonen. Simple public key certificate. IETF Internet Draft, draft-ietf-spki-cert-structure-06.txt, July 1999.

[10] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B.M. Thomas, and T. Ylonen. SPKI Certificate Theory. IETF RFC 2693, September 1999.

[11] P. England, J. DeTreville, and B. Lampson. Digital Rights Management Operating System, December 2001. United States Patent 6,330,670.

[12] P. England, J. DeTreville, and B. Lampson. Loading and Identifying a Digital Rights Management Operating System, December 2001. United States Patent 6,327,652.

[13] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A Trusted Open Platform. *Computer*, pages 55–62, July 2003.

[14] P. England and M. Peinado. Authenticated Operation of Open Computing Devices. In *Information Security and Privacy*, pages 346–361. Springer-Verlag LNCS 2384, 2002.

[15] P. Eronen and P. Nikkander. Decentralized Jini Security. In *Network and Distributed System Security*, February 2001.

[16] S. Farrell and R. Housley. An Internet Attribute Certificate Profile for Authorization. IETF RFC 3281, April 2002.

[17] E. Felten. Understanding Trusted Computing. *IEEE Security & Privacy*, pages 60–62, May/June 2003.

[18] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[19] K. Fu, E. Sit, K. Smith, and N. Feamster. Dos and Don'ts of Client Authentication on the Web. In *USENIX Security*, 2001.

[20] T. Garfinkel, M. Rosenblum, and D. Boneh. Flexible OS Support and Applications for Trusted Computing. In *9th Hot Topics in Operating Systems (HOTOS-IX)*, 2003.

[21] P. Gutmann. How to recover private keys for Microsoft Internet Explorer, Internet Information Server, Outlook Express, and many others - or - Where do your encryption keys want to go today? `http://www.cs.auckland.ac.nz/~pgut001/pubs/breakms.txt`.

[22] IBM Research Demonstrates Linux Running on Secure Cryptographic Coprocessor, August 2001. Press release.

[23] A. Iliev and S.W. Smith. Privacy-Enhanced Credential Services. In *2nd Annual PKI Research Workshop*. NIST, April 2003.

[24] N. Itoi. Secure Coprocessor Integration with Kerberos V5. In *9th USENIX Security Symposium*, 2000.

[25] S. Jiang, S.W. Smith, and K. Minami. Securing Web Servers against Insider Attack. In *Seventeenth Annual Computer Security Applications Conference*, pages 265–276. IEEE Computer Society, 2001.

[26] C. Kaufman, R. Perlman, and M. Speciner. *Network Security - Private Communication in a Public World*, chapter 15. Prentice Hall, 2nd edition, 2002.

[27] Keynote home page. `http://www.cis.upenn.edu/~keynote/`.

[28] R. Kohlas and U. Maurer. Reasoning About Public-Key Certification: On Bindings Between Entities and Public Keys. *Journal on Selected Areas in Communications*, pages 551–560, 2000.

[29] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the 9th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–177, November 2000.

[30] M. Lorch, J. Basney, and D. Kafura. A Hardware-secured Credential Repository for Grid PKIs. In *4th IEEE/ACM International Symposium on Cluster Computing and the Grid*, April 2004.

[31] R. MacDonald, S.W. Smith, J. Marchesini, and O. Wild. Bear: An Open-Source Virtual Coprocessor based on TCPA. Technical Report TR2003-471, Department of Computer Science, Dartmouth College, 2003.

[32] J. Marchesini and S.W. Smith. Virtual Hierarchies - An Architecture for Building and Maintaining Efficient and Resilient Trust Chains. In *NORDSEC2002 - 7th Nordic Workshop on Secure IT Systems*, November 2002.

[33] J. Marchesini, S.W. Smith, O. Wild, and R. MacDonald. Experimenting with TCPA/TCG Hardware, Or: How I Learned to Stop Worrying and Love The Bear. Technical Report TR2003-476, Department of Computer Science, Dartmouth College, 2003.

[34] J. Marchesini, S.W. Smith, and M. Zhao. Keyjacking: The Surprising Insecurity of Client-Side SSL. Technical Report TR2004-489, Department of Computer Science, Dartmouth College, 2004.

[35] U. Maurer. Modelling a Public-Key Infrastructure. In *ESORICS*. Apringer-Verlag LNCS, 1996.

[36] P. McGregor and R. Lee. Virtual Secure Co-Processing on General-purpose Processors. Technical Report CE-L2002-003, Princeton University, November 2002.

[37] Microsoft. Crypto API Documentation. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/securi%ty/security/cryptography_portal.asp`.

[38] Microsoft .Net Passport. `http://www.passport.net/Consumer/default.asp`.

[39] S. Nazareth. SPADE: SPKI/SDSI for Attribute Release Policies in a Distributed Environment. Master's thesis, Department of Computer Science, Dartmouth College, May 2003. `http://www.cs.dartmouth.edu/~pkilab/theses/sidharth.pdf`.

[40] S. Nazareth and S.W. Smith. Using SPKI/SDSI for Distributed Maintenance of Attribute Release Policies in Shibboleth. Technical Report TR2004-485, Department of Computer Science, Dartmouth College, 2003.

[41] nCipher Security Products. `http://www.ncipher.com/products/`.

[42] J. Novotny, S. Tueke, and V. Welch. An Online Credential Repository for the Grid: MyProxy. In *Proceedings of the Tenth International Symposium on High Performance Distributed Computing (HPDC-10)*. IEEE Press, August 2001.

[43] Department of Defense. Department of Defense Trusted Computer System Evaluation Criteria. DoD 5200.28-STD, December 1985.

[44] The OpenSSL Project. `http://www.openssl.org`.

[45] S. Pearson, editor. *Trusted Computing Platforms: TCPA Technology in Context*. Prentice Hall, 2003.

[46] M. Periera. Trusted S/MIME Gateways, May 2003. Senior Honors Thesis. Also available as Computer Science Technical Report TR2003-461, Dartmouth College.

[47] Permis home page. `http://www.permis.org/`.

[48] T. Perrin, L. Bruns, J. Moreh, and T. Olkin. Delegated Cryptography, Online Trusted Third Parties, and PKI. In *1nd Annual PKI Research Workshop*. NIST, April 2002.

[49] S. Proctor. Sun's XACML Implementation. `http://sunxacml.sourceforge.net/`.

[50] Pubcookie: open-source software for intra-institutional web authentication. `http://www.washington.edu/pubcookie/`.

[51] Securely Available Credentials-Framework. `http://www.imc.org/ietf-sacred/index.html`. draft-ietf-sacred-framework.

[52] Securely Available Credentials-Protocol. `http://www.imc.org/ietf-sacred/index.html`. draft-ietf-sacred-protocol-bss.

[53] Securely Available Credentials-Requirements. `http://www.imc.org/ietf-sacred/index.html`. RFC3157.

[54] D. Safford. Clarifying Misinformation on TCPA. `http://www.research.ibm.com/gsal/tcpa/tcpa_rebuttal.pdf`, October 2002.

[55] D. Safford. The Need for TCPA. `http://www.research.ibm.com/gsal/tcpa/why_tcpa.pdf`, October 2002.

[56] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architectrue. Technical report, IBM Research Division, RC23064, January 16, 2004. to Appear at the 13th Annual USENIX Security Symposium.

[57] F. Schneider. Secure Systems Conundrum. *Communications of the ACM*, 45(10):160, October 2002.

[58] B. Schneier. *Applied Cryptography*. Wiley, 2nd edition, 1996.

[59] S. Schoen. Who Controls Your Computer? Electronic Frontier Foundation Reports on Trusted Computing. `http://www.eff.org/Infra/trusted_computing/20031002_eff_pr.php`, October 2003.

[60] RSA Security. PKCS#11 - Cryptographic Token Interface Standard. `http://www.rsasecurity.com/rsalabs/pkcs/pkcs-11/index.html`.

[61] RSA Security. PKCS#12 - Personal Information Exchange Syntax Standard. `http://www.rsasecurity.com/rsalabs/pkcs/pkcs-12/index.html`.

[62] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWAtt: Software-based Attestation for Enbedded Devices. to Appear at 2004 IEEE Symposium on Security and Privacy, May 2004.

[63] S.W. Smith. Secure Coprocessing Applications and Research Issues. Technical Report Los Alamos Unclassified Release LA-UR-96-2805, Los Alamos National Laboratory, August 1996.

[64] S.W. Smith. WebALPS: A Survey of E-Commerce Privacy and Security Applications. *ACM SIGecom Exchanges*, 2.3, September 2001.

[65] S.W. Smith. Outbound Authentication for Programmable Secure Coprocessors. *International Journal on Information Security*, 2004.

[66] S.W. Smith, N.C. Goffee, S.H. Kim, P. Taylor, M. Zhao, and J. Marchesini. Greenpass: Flexible and Scalable Authorization for Wireless Networks. Technical Report TR2004-484, Department of Computer Science, Dartmouth College, 2004.

[67] S.W. Smith, E. Palmer, and S. Weingart. Using a High-Performance, Programmable Secure Coprocessor. In *Financial Cryptography*, pages 73–89. Springer-Verlag LNCS 1465, 1998.

[68] S.W. Smith and S. Weingart. Building a High-Performance, Programmable Secure Coprocessor. *Computer Networks*, 31:831–860, April 1999.

[69] N. Stam. Inside Intel's Secretive 'LaGrande' Project. `http://www.extremetech.com/`, September 19, 2003.

[70] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant processing. In *In Proceedings of the 17 Int'l Conference on Supercomputing*, pages 160–171, 2003.

[71] Trusted Computing Platform Alliance. TCPA Design Philosophies and Concepts, Version 1.0. `http://www.trustedcomputinggroup.org`, January 2001.

[72] Trusted Computing Platform Alliance. TCPA PC Specific Implementation Specification, Version 1.00. `http://www.trustedcomputinggroup.org`, September 2001.

[73] Trusted Computing Platform Alliance. Main Specification, Version 1.1b. `http://www.trustedcomputinggroup.org`, February 2002.

[74] S. Tuecke, V. Welch, D. Engert, L. Pearlman, and M. Thompson. Internet X.509 Public Key Infrastructure Proxy Certificate Profile. `http://www.ietf.org/internet-drafts/draft-ietf-pkix-proxy-10.txt`, 2003.

[75] Unpatched IE security holes. `http://www.pivx.com/larholm/unpatched/`.

[76] G. Vanrenen and S.W. Smith. Distributing Security-Mediated PKI. In *1st European PKI Workshop Research and Applications*. Springer-Verlag, 2004.

[77] V. Welch, I. Foster, C. Kesselman, O. Mulmo, L. Pearlman, S. Tuecke, J. Gawor, S. Meder, and F. Siebenlist. X.509 Proxy Certificates for Dynamic Delegation. In *3rd Annual PKI R&D Workshop Pre-Proceedings*, pages 31–47, April 2004.

[78] XACML 1.1 Specification Set. `http://www.oasis-open.org`, July 24, 2003.

[79] B.S. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, May 1994. Also available as Computer Science Technical Report CMU-CS-94-149, Carnegie Mellon University.

[80] B.S. Yee and J.D. Tygar. Secure Coprocessors in Electronic Commerce Applications. In *1st USENIX Electronic Commerce Workshop*, pages 155–170. USENIX, 1995.