SHEMP: Secure Hardware Enhanced MyProxy

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

in

Computer Science

by

John Marchesini

DARTMOUTH COLLEGE

Hanover, New Hampshire

March 7th, 2005

Examining Committee:

(chair) Sean W. Smith

Hany Farid

Doug McIlroy

Leendert van Doorn

Charles K. Barlowe, Ph.D.
Dean of Graduate Studies

Abstract

In 1976, Whitfield Diffie and Martin Hellman demonstrated how public key cryptography could enable secure information exchange between parties that do not share secrets. In order for public key cryptography to work in modern distributed environments, we need an infrastructure for finding and trusting other parties' public keys, i.e., a Public Key Infrastructure (PKI). While PKI applications differ in how they use keys, all applications share one assumption: users have keypairs.

This thesis begins by examining the security aspects of some of the standard keystores and their interaction with the Operating System. We establish that desktop keystores are not safe places to store private keys, and our experiments demonstrate the permeability of such keystores. Additionally, desktop keystores are immobile, difficult to use, and make it hard or impossible for relying parties to make reasonable trust judgments. We show that these problems stem from the fact that the *Trusted Computing Base* (TCB) of modern desktops is too large and ill-defined, which makes standard desktops suboptimal PKI clients.

Since we would like to use desktops as PKI clients and cannot realistically expect to redesign the entire desktop, this thesis presents a system that works within the confines of modern desktops to shrink the TCB needed for PKI applications. Our system is called *Secure Hardware Enhanced MyProxy* (SHEMP), and combines a number of techniques and technologies to shrink the TCB in space and allow the TCB's size to vary over time. In addition, the SHEMP system addresses the problems of immobility and usability, and allows relying parties to make reasonable trust judgments. Using analysis, experiments, and formal methods, we conclude that SHEMP makes standard desktops suitable for use as PKI clients.

The contributions of this thesis include the discovery of techniques used to identify weaknesses in modern desktops; a prototype, analysis, and correctness proof of a sys-

tem which makes desktops usable as PKI clients (SHEMP); a novel approach for reasoning about TCBs; and a formal framework for proving properties of PKI systems such as SHEMP.

Preface

During my time as a research assistant in Dartmouth's PKI lab, I have had the opportunity to work on a wide range of projects. In the early stages, it seemed as though every research project was designed to fit some particular niche, and the results were only applicable to the question at hand. I would approach each new research question with fervor, rarely searching for non-obvious connections to my previous results, and thus keeping a local view of each project. In hindsight, I realize that I was plotting points.

Only in the last year or so have I stepped back and tried to look at my results from a more global perspective, trying to find connections between the points. In doing so, a clear theme has emerged. All of my previous research efforts are asking variations of the same question: "Why should I trust the results of a given computation?" If I begin with the assumption that a specific program running on an isolated machine produces correct results, then what happens to the program as I put it into the maze of machines, networks, and software that constitutes the modern computing landscape? Should I still trust the results?

Having taken more of a bottom-up approach to writing a thesis, I had the opportunity to focus on numerous aspects of the trust issue in some detail. I have explored the areas of software security, trusted computing, secure systems, PKI, and formal methods, and I have built systems as well as broken them. Getting such exposure was the motivating factor for pursuing a Ph.D. in the first place, and I am fortunate to have had the chance to work in so many areas.

I am also fortunate to have had the chance to work with my advisor, Sean Smith. Sean had the patience to deal with my wandering curiosity, half-baked ideas, and frequent lack of clarity. He influenced and inspired me in a number of ways both professionally and personally, and in the end, Sean helped me grow in many dimensions. He made graduate

school a fun and rewarding experience.

There are a number of other individuals who deserve mention. First, I would like to thank my friend and office mate, Alex Iliev. He taught me many things, provided hours of good conversation, and most of all, he helped me achieve balance. Alex also instilled in me an appreciation for rigor, a quality which does not come naturally to me. Second, I wish to thank my father for giving me a computer when I was eight (which warped my brain in the proper way), and for reminding me occasionally that a Ph.D. really is worth the effort. Third, I would like to thank my committee for their comments and support, and for keeping me on my toes. Last, I owe many thanks to Wendy who stood by my side and put up with me throughout this experience.

This thesis was defended on March 4, 2005. Portions of this thesis appear in my thesis proposal document, available as Dartmouth College Technical Report TR2004-525 [69]. Portions of Chapter 2 have been published in the *Proceedings of the 2nd Annual PKI Research and Development Workshop* [74] and *Computers and Security* [75]. Parts of Chapter 4 were published in the *Proceedings of 20th Annual ACSAC Conference* [73]; a revised and extended discussion of this research can be found in Section 11.4 of *Trusted Computing Platforms: Design and Applications* [117]. A summarized version of this thesis is available as Dartmouth College Technical Report TR2005-532 [71].

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 thesis does not necessarily reflect the views of the sponsors.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Because public-key cryptography can enable secure information exchange between parties that do not share secrets a priori, *Public Key Infrastructure* (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., some applications use the key for message encryption and signing, while others uses it for authentication), all applications share one assumption: users have keypairs. Where these user keypairs are stored and how they are 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 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 to perform cryptographic operations. 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 keypair without having to rely on the underlying *Operating System* (OS), thus enhancing portability.

All of these key storage approaches have something in common: in order for the private key to be used, it must somehow be accessible to the set of software that exists on the desktop[1]. The security properties (or lack thereof) of modern desktops makes this access problematic. In order for an application to use the private key, it must expose highly sensitive material (the private key) to an insecure and hostile environment (the desktop). As we will explore in Chapter 2, the lack of desktop security makes it difficult or impossible for relying parties to make reasonable trust judgments, and thus decreases and/or eliminates the utility of desktops as PKI clients.

In addition to security concerns, desktops suffer a number of other limitations which hinder their use as PKI clients. Among these limitations are the lack of mobility and the lack of usability. Modern computing populations are becoming increasingly mobile, and a PKI client that fails to accommodate this mobility (such as the modern desktop) is ineffective in certain environments. Furthermore, as we will discuss in Chapter 2, modern desktops make it difficult for users and application developers to construct valid mental models of the system. As a result, it becomes difficult to use the system correctly.

The goal of this thesis is to make desktops usable as PKI clients. We begin by examining the core reasons why desktops are unsuitable for use as PKI clients. We then establish criteria for a solution, and introduce a system called *Secure Hardware Enhanced MyProxy* (SHEMP) which meets our criteria and makes desktops usable as a PKI clients.

In the remainder of this chapter, we introduce the major components of this thesis, provide a basic outline for the rest of the thesis, and present the contributions of this work.

[1] In this thesis, we define the desktop as a computer running a standard commodity OS (e.g., Windows). Our definition covers personal computers, workstations, and laptop computers

1.1 Keystores

Most modern keystores fall into one of four basic categories:

- A *software token* stores the key on disk in some sort of encrypted format. Examples include the default CSP for Windows and the Mozilla/Netscape Web browser. Sandhu et al. discuss other software tokens in depth [107].

- A *hardware token* 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). 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: the *Cryptographic API* (CAPI) for Microsoft [78], and RSA's PKCS#11 [109] 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* (HSMs)) such as nCipher's nShield [85].

- A *secure coprocessor* stores the key, performs key operations internally using specialized cryptographic hardware, and can even house applications directly. An example of a secure coprocessor is the IBM 4758 [18, 119]. These devices can also be used as cryptographic accelerators or HSMs.

- A *credential repository* is a dedicated machine that stores private keys for a number of users. When a user Alice wishes to perform private key operations, she must first authenticate to the repository. The repository then certifies a temporary key with Alice's permanent key via a digital signature, issues a temporary credential, or ac-

tively participates in the requested key operation. Examples of credential repositories include MyProxy [88], hardened MyProxy [64], and SEM [10].

Software and Hardware Tokens In Chapter 2, we will examine the security aspects of some of the standard keystores and their interaction with the desktop. We will show that software tokens are not safe places to store private keys, and we will demonstrate the permeability of keystores such as the Microsoft default CSP and the Mozilla keystore. Our experiments will show that in many cases, it is possible for an attacker to either steal the private key or use it at will.

In addition to being unsafe, standard software keystores have the disadvantage of being immobile. Once a private 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 we will discuss in Chapter 2, since this process can make the key vulnerable to attack, such solutions may offer mobility at the expense of security. As user populations become more mobile and begin to use multiple devices, this immobility becomes more problematic.

Hardware 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 typically still able to use the key at will. However, 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 (but the tokens we experimented with did not have Apple or Linux support at the time of our experimentation).

As we will explore in Chapter 2, the security problems of software and hardware tokens stem from the facts that the *Trusted Computing Base* (TCB) is too large and ill-defined, and that usability issues make it hard for users and application developers to "do the right thing." These shortcomings make it impossible for relying parties to make reasonable trust judgments about the system.

Secure Coprocessors In previous work, we examined secure coprocessors (e.g., [114, 139, 140]): 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 can be used to shrink the TCB significantly. Secure coprocessors have been shown to be feasible as commercial products [18, 119] and can even run Linux and modern build tools [44].

We have explored using secure coprocessors for *trusted computing*—both as general designs (e.g., [94]) as well as real prototypes (e.g., [46])—but repeatedly were hampered by their relatively weak computational power. Their relatively high cost also inhibits widespread adoption, particularly at clients. Their lack of ubiquity, coupled with their sometimes awkward programming environments, lead us to conclude that secure coprocessors are difficult to use, especially for application developers.

In previous work, we used inexpensive commodity hardware to secure an entire desktop [67, 72, 73]. While the security properties are not as strong as a real secure coprocessor (such as the IBM 4758), our approach shrinks the standard TCB of a general purpose desktop. Our platform (called *Bear/Enforcer*) will be discussed further in Chapter 4.

Credential Repositories Credential repositories can provide safe storage facilities for private keys as well as give users mobility. The repository approach allows an organization to focus security resources on the repository, thus providing economies of scale. In terms of secure key storage, repositories significantly shrink the TCB. The private key no longer relies on a general purpose (and buggy) desktop for safe storage, but instead on a dedicated server which is presumably administered by a professional and is running only what the administrator allows. Repositories also allow users to access their private key from multiple machines, thus giving users mobility.

However, when a user Alice wishes to use her private key to perform some operations,

Keystore	Secure	Usable	Mobile	Allows Judgments
Software Token	no	no	no	no
Hardware Token	no	no	maybe	no
Coprocessors	yes	maybe	no	yes
Repositories	maybe	no	yes	no

Table 1.1: A summary of modern keystores.

she must bring the key to her desktop, or use a protocol which allows her to use the private key on the repository (and rewrite her application to use this new protocol). Thus, repositories can be difficult to use, especially for application developers.

Recently, a new credential repository has been developed and embraced by the Grid computing community which provides both security and mobility to clients. Their repository is called *MyProxy* [88], and there have even been efforts to harden a MyProxy repository by using an IBM 4758 for key storage and cryptographic operations [64]. MyProxy will be discussed further in Chapter 4.

Desktops Are Not Usable as PKI Clients As we will establish in Chapter 3, a usable key storage solution must be secure, be usable, give users mobility, and allow relying parties to make reasonable trust judgments. As we have discussed in this section, none of the current approaches meet this criteria. Table 1.1 summarizes the status quo.

1.2 SHEMP Overview

The status quo is not satisfactory. Ideally, we need a way to use a desktop as a PKI client which answers "yes" in all of the columns of Table 1.1. Since we cannot redesign the entire desktop and expect anyone to use it, our solution must operate within the confines of modern desktops. Additionally, in order to remain usable to application developers, it must adhere to common development paradigms and practices.

Our solution is a system which we call SHEMP, and will be discussed in detail in Chapter 4, Chapter 5, and Chapter 6. Briefly, the idea behind SHEMP is to remove the private key from the desktop altogether, placing it in a SHEMP credential repository which is based on the MyProxy design. When a user Alice wishes to use her private key, she logs into the SHEMP repository from her client desktop, generates a temporary keypair on her desktop, and then requests a *Proxy Certificate* (PC) [126, 132] that includes the public portion of her temporary keypair and is signed by her permanent private key on the repository.

If Alice's policy permits the operation under her current security context, then a PC is generated, signed, and returned to Alice. The PC is only valid for a short period of time, and includes a snapshot of the environment in which the PC was generated. This snapshot describes the security attributes of the repository and client desktop, and allows applications to decide for themselves how trustworthy the private key described by PC really is.

The SHEMP system attempts to leverage secure hardware when it can, but it does not require secure hardware. Concretely, SHEMP allows keypairs on the repository and the client to be generated and used in secure coprocessors. Additionally, the framework for describing the security attributes of repositories and client desktops allows users and administrators to express the presence and quality of secure hardware. Finally, the SHEMP policy mechanism allows users, relying parties, and applications to make decisions based on the current security context, including the presence and quality of secure hardware.

Using SHEMP, Alice can walk up to any computer in her domain, generate a temporary keypair (possibly using secure hardware), and request that the SHEMP repository generate a PC for her. She can then give her PC to an application and request that the application perform some operation. Finally, the application carrying out the operation can use the SHEMP policy mechanism to decide whether or not to perform the operation based on Alice's current environment.

Making Desktops Usable for PKI The SHEMP system will be evaluated analytically and experimentally in Chapter 7 and formally in Chapter 8. Put simply, SHEMP meets the criteria shown in Table 1.1, and thus makes desktops usable as PKI clients. SHEMP obtains security by taking the private key off of the desktop altogether and storing it in a credential repository (which is possibly storing the key in secure hardware). Thus, the TCB when the key is not in use only includes the repository.

When Alice needs to use her private key, her desktop generates a temporary keypair (again, possibly storing the key in secure hardware) and attempts to obtain a PC which contains the public portion of the temporary key. If Alice's policy permits the operation, the TCB is extended to include the desktop—but only to store and wield a short-lived disposable keypair. When Alice logs out or her PC expires, the TCB shrinks to only include the repository again. As we will discuss in Chapter 7, should Alice's desktop be compromised and her temporary private key disclosed, SHEMP reduces the attacker's window of opportunity for misuse by issuing short-lived PCs.

SHEMP provides usability to users and application developers by including information about Alice's current environment in the PC. In essence, this information is a snapshot of the current TCB under which Alice generated her temporary keypair and PC. These snapshots also allow relying parties to make reasonable trust judgments.

SHEMP solves the mobility problem by allowing Alice to log on to a SHEMP repository and request a PC from anywhere in her domain. Some machines that Alice uses may be more secure than others—for instance, a public terminal in a common area may be significantly less secure than the desktop in her office which has secure hardware. Not only does SHEMP allow Alice to use a variety of machines with different security levels, it relays the security level to the relying party via the PC.

In essence, SHEMP makes desktops usable as PKI clients by first recognizing that the TCB should not be treated as a static entity. SHEMP views the effective TCB as the union

of the repository's TCB with the client's, and recognizes that this TCB can vary in size (by using secure hardware and by potentially limiting the key operations in accordance with a user's policy) and in time (by issuing short-lived PCs). By viewing the TCB as a dynamic entity, SHEMP keeps the TCB small, and allows it to expand only when needed. Furthermore, SHEMP allows users to control the TCB via policy. SHEMP maintains security while giving users mobility by recognizing that client machines may have varying levels of security, and that users and relying parties need these security properties in order to construct valid mental models of the system as well as make reasonable trust judgments.

1.3 Thesis Contributions

In this thesis, we examine the usability of standard desktops as PKI clients. As we discuss in Chapter 2, much effort and many resources are being put towards rolling out real PKI systems which rely on standard desktops as their clients. The first contribution of this work, and indeed a fundamental one, is the discovery of the problem: desktops are not usable as PKI clients. Despite the barrage of marketing literature suggesting that users are better off by using private keys stored on standard desktops, our experiments show that such approaches are vulnerable to an attack we call *keyjacking*.

Our approach to making desktops usable as PKI clients involves the design and implementation of the SHEMP system, which is our second contribution. Additionally, SHEMP employs some novel construction techniques which are of interest in their own right. The first of these techniques is taking the view that the TCB is a dynamic entity which varies in space and time. Previous discussions of the TCB (such as what is found in the "Orange Book" [90]) are limited to treating the TCB as a purely spatial entity (sometimes called a "security perimeter"). SHEMP also attempts to describe the state of the TCB to those who need it most: relying parties. The second technique involves the use of PCs for decryption

and signing applications (described in Chapter 6). Current uses of PCs focus solely on authorization.

Finally, in order for relying parties to make reasonable trust judgments about SHEMP, they need a framework in which to express and reason about the flow of trust in the system. As we will discuss in Chapter 8, this task is best accomplished with the use of formal methods. A number of calculi have been developed for this task, but none of them are robust enough to deal with the variety of issues found in SHEMP (such as multiple certificate formats, the concept of time, and revocation). In Chapter 8 we present a new calculus for reasoning about PKI systems, and use it to offer a formal proof of SHEMP's correctness. We also use the calculus to reason about a number of PKI systems which we could not model using the other calculi. Thus, we believe that our calculus is a contribution in its own right.

In short form, this thesis makes the following contributions:

- Demonstration of architectural flaws within Microsoft's Cryptographic API, a suite of attacks which exploit those flaws, and the realization that such flaws make desktops unsuitable for use as PKI clients,
- The establishment of criteria for making desktops usable as PKI clients,
- A working system which meets that criteria: SHEMP,
- The view that the TCB should be treated as a dynamic entity,
- The implementation of applications which use Proxy Certificates for decryption and signing, and
- A formal framework for reasoning about PKI systems.

1.4 Thesis Outline

This thesis is organized as follows: Chapter 2 describes why desktops are not usable as PKI clients. We present our experiments and an analysis of the results. Chapter 3 establishes criteria which any solution to this problem must meet, providing analysis of where and why previous efforts have been unsuccessful, and how they differ from SHEMP.

Chapter 4 introduces the building blocks used to build SHEMP. Chapter 5 gives a detailed explanation of the design and implementation of the SHEMP system. Chapter 6 describes applications which we designed and implemented, and introduces a number of applications where SHEMP could provide an improvement over current approaches.

Chapter 7 provides an analytical and experimental evaluation of the SHEMP system. We show how SHEMP meets the criteria put forth in Chapter 3, we present the results of our usability study and performance tests, and we discuss how we would attack the SHEMP system.

Chapter 8 presents our formal framework for reasoning about PKI systems; we show how our framework is robust enough to model systems that other frameworks cannot. Then, we use our new framework to prove that SHEMP is correct and that a number of applications we built are also correct.

Finally, Chapter 9 presents a summary and analysis of the thesis, and introduces some areas for possible future work.

Chapter 2

Keyjacking

In the last decade, the Web has become the dominant paradigm for electronic access to information services. The *Secure Sockets Layer* (SSL) is the dominant paradigm for securing Web interaction. For a long time, SSL with *server-side authentication*—where, during the handshake, the server presents a public-key certificate and demonstrates knowledge of the corresponding private key—was perhaps the most accessible use of PKI in the lives of ordinary users.

However, in the full vision of PKI, all users have keypairs, not just the server operators. Within the SSL specification, a server can request *client-side authentication* where, during the handshake, the client also presents a public-key certificate and demonstrates knowledge of the corresponding private key. The server can then use this information for identification, authentication, and access control on the services it provides to this client.

Client-side PKI exploits the natural synergy between these two scenarios. Because the Web is the way we do business and client-side SSL permits servers to authenticate clients, we are beginning to see some of the necessary building blocks to achieve the client-side vision:

- Many modern operating systems (e.g., all flavors of Windows and Mac OSX) include

a keystore and a set of CSPs which can be used by any application on the machine.

- Modern browsers which are designed to be used across multiple platforms (e.g., Netscape/Mozilla) now include keystores for a user's keypairs.

- Enterprises (and other distributed populations) are arranging for users to obtain certified keypairs to live in these keystores. Some populations are even making plans to distribute USB tokens to users in order to store their keypairs.

- Providers of Web information services are starting to use client-side SSL as a better alternative than passwords to authenticate users.

Using client-side PKI can alleviate the need for users to remember multiple passwords for multiple services. It also reduces the risk of an attacker capturing a user's password either by guessing or via a keyboard sniffer.

Does It Work? In previous work, our lab has examined the effectiveness of server-side SSL [137] and of digital signatures on documents [53]. In this chapter, we examine the question: *does client-side PKI work*?

- When browsers use a private key in contemporary desktop environments, is it reasonable for the user at the client to assume that his private key is used only to authenticate services he was aware of, and intended?

- Is it reasonable for the user at the server to assume that, if a request is authenticated via client-side SSL, that that client was aware of and approved that request?

The perception of users—not only the users at the client, but also the application authors and administrators at the server—plays a critical role in determining whether client-side PKI works. If the natural mental models of the system do not match the actual system

behavior, then users have no basis to make reasonable trust judgments. In security settings such as client-side PKI, this inability to reason about the system can thwart the security efforts that the system's designers have implemented.

Our Agenda We wish to stress that we believe that PKI is a much better way than the alternatives to carry out authentication and authorization in distributed, multi-organizational settings. PKI does not require shared secrets or a previously-established direct trust relationship between the two parties. Further, PKI permits many parties to make assertions, and allows for non-repudiation of those assertions—Bob can prove to Cathy that Alice authorized this request to Bob.

However, rolling out client-side PKI and migrating existing information services to use it requires considerable resources and effort. Weaknesses in the underlying technology risk undermining this effort. We provide a critical examination of the current client-side PKI approach precisely because we want the PKI vision to succeed. Indeed, the remainder of this thesis is devoted to a system (SHEMP) which strives to realize the PKI vision.

Chapter Outline In the next section, we examine the status quo. In Section 2.2, we pose the question which drives our experiments. Section 2.3, Section 2.4, Section 2.5, and Section 2.6 describe our experiments. We conclude this chapter in Section 2.7.

2.1 The Current State of Affairs

2.1.1 Web Information Services and Web Applications

Currently, the Web is the dominant paradigm for information services. Typically, the browser issues a request to a server and the server responds with material which the browser renders.

Language of the Interaction From the initial perspective of a browser user (or the crafter of a home page), the "requests" correspond to explicit user actions, such as clicking a link or typing a URL; the "responses" consist of HTML files.

However, the language of the interaction is richer than this, and not necessarily well-defined. The HTML content a server provides can include references to other HTML content at other servers. Depending on the tastes of the server operator and the browser, the content can also include executable code; Java and Javascript are fairly universal. This richer content language provides many ways for the browser to issue requests that are more complex than a user might expect, and not necessarily correlated to user actions like "clicking on a link."

As part of a request, the browser will quietly provide parameters such as the browser platform and the REFERER (sic)—the URL of the page that contained the link that generated this request.

In the current computing paradigm, we also see a continual bleeding between Web interaction and other applications. For example, in many desktop configurations, a server can send a file in an application format (such as PDF or Word), which the browser hands off to the appropriate application; non-Web content (such as PDF or Word) can contain Web links, and cause the application to happily issue Web requests.

Web Applications Surfing through hypertext documents constituted the initial vision for the Web and, for many users, its initial use. However, in current enterprise settings, the interaction is typically much richer: users (both of the browser and server) want to map non-electronic processes into the Web, by having client users fill out forms that engender personalized responses (e.g., a list of links matching a search term, or the user's current medical history) and perhaps have non-Web consequences (such as registering for classes or placing an Amazon order). In the standard way of doing this, the server provides an

HTML `form` element which the browser user fills out and returns to a *common gateway interface (CGI)* script (e.g., see Chapter 15 in [87]).

The `form` element can contain `input` tags that (when rendered by the browser) produce the familiar elements of a Web form: boxes to enter text, boxes with a "browse" tag to enter file names for upload, radio buttons, check-boxes, etc. For each of these tags, the server may specify a `name` which names the parameter being collected from the user and a default `value`. The server content associates this form with a `submit` action (typically triggered by the user pressing a button labeled "Submit"), which transforms the parameters and their values into a request for a specific URL. If the `submit` action specified the `GET` method, the parameters are pasted onto the end of the URL; if the action specified the `POST` method, the parameters are sent back in a second request part.

However, the submit URL specifies an executable script, not a passive HTML file, in the Web directory at the server. When a server receives a request for such a script, it invokes the script. The script can interrogate request parameters such as the form responses, interact with other software at the server side, and also dynamically craft content to return to the browser.

2.1.2 Security Mechanisms

In enterprise settings, the server operator may wish to restrict content to authorized browser users. When the browser user is requesting a service via a `form`, the server operator may wish to authenticate specific attributes about the user, such as identity and the fact that the user authorizes this request. The Web paradigm provides several standard avenues to perform such a task.

Client Address The server may restrict requests to client machines with specific hostname or IP address properties.

Passwords With *basic authentication* (or the *digest authentication* variant), the server can require that the user present a username and password, which the browser collects via a special user interface channel and returns to the server. The server requesting the authentication can provide some text that the browser will display in the password-prompt box. Alternatively, the server may also collect such authenticators as part of the `form` responses from the user.

With these various forms of password-based authentication, the server operator would be wise to take steps to ensure that sensitive data is protected in transit. Some of the common approaches include offering the entire service over an SSL channel, and having the form submitted by the `POST` method, so the responses are not cataloged in histories, logs, `REFERER` fields, etc..

Indeed, if neither the user nor server otherwise expose a user's password, and if the user has authenticated that he is talking to the intended server, then a strong case can be made that a properly authenticated request requires the user's awareness and approval. The password had to come from somewhere!

Password-based systems have a number of risks. Users may pick bad passwords or share them across services; the authentication is not bound to the actual service (i.e., we have no non-repudiation); the adversary may mount online guessing attacks (Pinkus et al. have recently considered some interesting countermeasures [98]); users may not check that they are connected to correct server, making them vulnerable to bogus sites that look similar (i.e., "Spoofing" or "Phishing" [31, 137, 138]).

Cookies The server 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” [101] and Microsoft’s “Passport” [79]—use one of the above mechanisms 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.

Cookie-based authentication can also be risky. Fu et al. [33] discuss many design flaws in cookie-based authentication schemes; PivX [127] discusses many implementation flaws in IE that allows an adversarial site to read other sites’ cookies.

2.1.3 Validating User Input

Besides authenticating the user, another critical security aspect of providing Web services is ensuring that the input is correct. Failing to do so can lead to a number of issues. For example, an adversarial user can exploit server-side script vulnerabilities by carefully crafting *escape sequences* that cause the server to behave in unintended ways. The canonical example here is a server using user input as an argument in a shell command; devious input can cause the server to execute a command of the user’s choosing. Another example occurs on the application level, where an adversarial user can change the request data, such as form fields or cookie values. The canonical example here is a commerce server that collects items and prices via a form, and allows a malicious user to purchase an item for a lower price than the vendor intended.

Standard good advice is that the script writer thoroughly vet any user input [37], and also verify that critical data being returned has not been modified [99].

2.1.4 Client-Side PKI

When prodded, PKI researchers (such as ourselves) will recite a litany of reasons why PKI is a much better way than the alternatives to carry out authentication and authorization in distributed, multi-organizational settings. In practical settings, many enterprises are adopt-

ing PKI technology because it allows single sign-on, minimizes the impact of keyboard sniffers, and is being heavily marketed by PKI vendors.

As we mentioned in the introduction, using various keystores and client-side SSL is a dominant emerging paradigm for bringing PKI to large populations. Some organizations currently using client-side SSL include Dartmouth College, MIT, the Globus Grid project, IBM WebSphere, many enterprise PKIs, Federal PKI projects, and many suppliers of VPN software.

On the application end, numerous players preach that client-side SSL is a better way to authenticate users than passwords. We cite a few examples culled from the Web:

- The W3C: "SSL can also be used to verify the users' identity to the server, providing more reliable authentication than the common password-based authentication schemes." [121]

- Verisign: "Digital IDs (digital certificates) give web sites the only control mechanism available today that implements easily, provides enhanced security over passwords, and enables a better user experience." [47]

- Thawte: "Most modern Web browsers allow you to use a Personal Email Certificate from Thawte to authenticate yourself to a Web server. Certificate-based authentication is much stronger and more secure than password-based authentication." [96]

- Entrust: "... identify or authenticate users to a Web site using digital certificates as opposed to username/password authentication where passwords are stored on the server and open to attacks." [13]

Recent research on user authentication issues also cite client-side SSL as the desired (but impractical) solution [33, 98]. The clear message is that Web services using password-based authentication would be much stronger if they used client-side SSL instead.

At the Server As noted earlier, SSL permits the browser and user to establish an encrypted, integrity-protected channel over which to carry out their Web interaction: request, cookies, form responses, basic authentication data, etc. The typical SSL use includes server authentication; newer SSL uses permit the browser to authenticate via PKI as well. The server operator can require that a client authenticate via PKI and can restrict access based on how it chooses to validate the client certificate. Server-side CGI scripts can then interrogate client-certificate information (often via environment variables), along with the other parameters available.

At the Browser Different browsers take different approaches to storing keys and certificates. Our experiment focuses on the two browsers which are the most commonly used: Netscape/Mozilla and Microsoft's *Internet Explorer* (IE) .

Netscape/Mozilla stores its security information in a subdirectory of the application named `.netscape` (`.mozilla` in Mozilla). There are two files of primary interest: `key3.db` which stores the user's private key, and `cert8.db` which stores the certificates recognized by the browser's security module. Both of these files are binary data stored in the Berkeley DB 1.85 format [6]. Additionally, the information in these files is encrypted with a keyphrase so that any application capable of reading the Berkeley DB format is still required to provide a password to read the plaintext or to modify the files without detection. A detailed description of the techniques used to securely store user's keys is beyond the scope of this thesis, but we point readers to the relevant literature (e.g., [39, 40, 43, 81, 86]) for details.

IE relies on the Windows system keystore and CSP to store the private key. One unfortunate result of this tight coupling between IE and the OS is that versions of IE which run on MacIntosh computers have no support for storing or using private keys.

By default, Windows uses its own CSPs to store the private key, which generate *low*

Figure 2.1: The Microsoft CSP's password prompt dialog.

security keys (i.e., not password-protected) by default. Many organizations (such as the *Department of Defense* (DoD) and even Microsoft) recommend against this behavior, noting that the key is only as secure as the user's account [80]. This implies that if an attacker were to gain access to a user's account or convince the user to execute code with the user's privileges, the attacker would be able to use the private key at will, without having to go through any protections on the key (such as a password challenge).

One way to remedy the lack of password protection is to "export" the private key, placing it in a password protected `.pwl` file (for IE 3 and earlier) or a `.pfx` file which stores the key in PKCS#12 (for IE 4 to current versions). Once the key is exported, a user must then "import" it at a higher security level: *medium-security*, which prompts the user when the key is used; or *high-security*, which requires a password to use the key (assuming the user does not check the box marked "Remember password", which immediately demotes it to a low-security key. See Figure 2.1).

While exporting and re-importing the private key may seem like a cumbersome process, it has become a standard practice in many organizations. In fact, the DoD guidelines for the Defense Message System outline the process in detail [49].

2.2 The Question

We believe that PKI is valuable and that secure Web information services are important. We also realize that any deployment will require considerable effort and user education (as we participate in such a deployment here at Dartmouth). Hence, we believe that it is important to ask the fundamental question: *does it work?*

Discussions of usability and security stress the importance the system behaving as the user expects [141], and the dangers in creating systems whose proper use is too complex [1, 134]. It is advertised that by using client-side PKI, a client can assume that his private key is used only to authenticate services he was aware of and intended, and a server operator can assume that the client was aware of and approved that request. In the common understanding of the Web, the user must choose to click in order to issue a request. With the protections promised by medium-security and high-security CSPs, we additionally expect the user to see and approve a warning dialog before a private key is used.

If we encourage user populations to enroll in client-side PKI, and encourage service providers to migrate current services to use client-side SSL authentication and to roll out new services this way, have we achieved the desired goals: that service requests are authenticated from user A only when user A consciously issued that request?

To this end, we carried out a series of experiments in order to evaluate the effectiveness of using the desktop, browser, and client-SSL as a component of a client-side PKI. (However, some of our attacks have a wide range of applications, and could potentially be used to subvert other authentication schemes as well. We focus on PKI because it is claimed to be the strongest—and in theory, it could be.)

We were not focused on bizarre bugs or extremely carefully constructed applications, but on general usability. If users on either end follow the "path of least resistance"—standard out-of-the-box configurations and advice—do they construct a mental model which

matches actual system behavior?

2.3 Experiment 1: Stealing Keys

2.3.1 Historical Vulnerabilities

This research began when we noticed some of the weakness of client-side PKI and browser keystores in the literature. Perhaps the most comprehensive list of problems with Microsoft's key storage system over the years comes from Peter Gutmann [35]. Three vulnerabilities in particular caught our attention.

The first vulnerability applies to situations where the private key is stored without password protection (the Microsoft CSP's default behavior). With a tool such as the "Offline NT Password & Registry Editor" [89], it is possible for an attacker to access a user's account in a few minutes, given physical access to the computer on which the account resides. Since the private key is not password-protected, an attacker can use the private key of the account's owner at will for as long as they are logged on. Additionally, an attacker could export the key to a floppy disk (password-protecting it with a password that the attacker chooses), and then use tools like Gutmann's or our modified version of OpenSSL to retrieve the key offline.

The second vulnerability comes from the format in which the private key is stored on disk once it has been exported (in a `.pwl` or `.pfx` file). Gutmann's *breakms* tool performs a dictionary attack to discover the password used to protect the file and outputs the private key.

The third vulnerability involves the `CryptExportKey` function found in the CryptoAPI, which Gutmann raised concerns about back in 1998. Specifically, with the default key generation (i.e., no password protection) and an exportable key, any program running

under the user's privileges may call the `CryptExportKey` function and silently obtain a copy of the user's private key.

In the Microsoft "low-security" model of client-side PKI, it seems that one has to trust the entire system (OS, IE, the CSP, etc.) for the client-side vision to work. If an attacker compromises one piece of the system (e.g., an executable with a user's privileges), then because of tight coupling, the attacker can violate other parts the system (e.g., the user's private key).

2.3.2 Stealing Low-Security Keys

We wondered if, with its new security emphasis, Microsoft had fixed some of these vulnerabilities, either directly or perhaps as a result of decoupling IE and the OS.

We began our experiments with the low-security key—a key which can be used by any application running with the user's privileges without warning the user that the key is in use. Immediately, we noticed that generating low security keys is still the Microsoft CSP's default behavior. We were curious to see if the latest versions of the CryptoAPI and CSP have remedied the `CryptExportKey` issue, perhaps by warning the user when their private key is being exported. Our conclusion: "no." We were able to construct a small executable which, when run on a low-security (default) key, quietly exports the user's private key with no warning.

2.3.3 Stealing IE Medium-Security and High-Security Keys

The Windows CryptoAPI does permit users to import keypairs at medium or high security levels. With both of these levels, use of the private key will trigger a warning window; in the high-security option, the warning window requests a password. Consequently, the previous attack may not work; when the executable asks the API to export the key, the user

may notice an unexpected warning window. So our attack strategy had to improve.

API Hijacking

Before we discuss the specifics of stealing medium and high security keys, a brief introduction to the general method of *API Hijacking* is in order. The goal of API Hijacking is to intercept (hijack) calls from some process (such as IE) to system APIs (such as the CryptoAPI)[1].

Delay Loading API Hijacking uses a feature of Microsoft's linker called "Delay Loading". Typically, when a process calls a function from an imported *Dynamic Link Library (DLL)* , the linker adds information about the DLL into the process in a place referred to as the *imports section*. We present a brief overview here (see [97] for details).

When a process is loaded, the Windows loader reads the imports section of the process, and dynamically loads each DLL required. As each DLL is loaded, the loader finds the address of each function in the DLL and writes this information into a data structure maintained in the process's imports section known as the *Import Address Table (IAT)* . As the name suggests, the IAT is essentially a table of function pointers.

When a DLL has the "Delay Load" feature enabled, the linker generates a small stub containing the DLL and function name. This stub (instead of the function's address) is placed into the imports section of the calling process. Now, when a function in the DLL is called by a process for the first time, the stub in the process's IAT dynamically loads the DLL (using `LoadLibrary` and `GetProcAddress`). This way, the DLL is not loaded until a function it provides is actually called—i.e., its loading is delayed until it is needed.

For delay loading to be used, the application must specify which DLLs it would like to delay load via a linker option during the build phase of the application.

[1] Very recently, other researchers have suggested an attack that replaces the original DLLs [92]. However, the countermeasures suggested for that attack do not defend against ours.

DLL Injection So, how does an attacker use delay loading on a program for which he can not build (possibly because he does not have the source code—i.e., IE)? The answer is to redirect the IAT of the victim process (e.g. IE) to point to a stub which implements the delay loading *while the process is running*.

The strategy is to get the stub code as well as the IAT redirection code into an attack DLL, and *inject* this DLL into the address space of the victim process. Once the attack DLL is in the process, the IAT redirection code changes the victim's IAT to point to the stub code. At that point, all of the victim process's calls to certain imported DLLs will pass through the attack DLL (which imported DLLs are targeted and which functions within those DLLs are specified by the attack DLL—i.e., the attacker gets to choose which DLLs to intercept). This implements a software man-in-the-middle attack between an application and certain DLLs on which the application depends.

The Windows OS provides a number of methods for injecting a DLL into an process's address space (a technique commonly referred to as "DLL Injection"). The preferred method is via a "Windows Hook", which is a point in the Windows message handling system where an application can install a routine which intercepts messages to a window.

Hijacking the CryptoAPI

Using the techniques above, we were able to construct a couple of programs which, running at user privileges only, allowed us to intercept function calls from IE to the CryptoAPI. This is particularly useful for stealing medium or high security private keys which display warning messages when used (in a client-side SSL negotiation, for example).

The idea is to wait for IE to use the key (hence, displaying the warning or prompting for a password), and then get a copy of the private key for ourselves—*without triggering an extra window that might alert the user.*

The Attack Essentially, the attack code is two programs: the *parasite*—an attack DLL with the IAT redirection code and the delay loading stubs, and the *grappling hook*—an executable to register a hook which is used to inject the parasite into IE's address space.

We implemented this attack as follows:

1. Get the parasite and grappling hook onto the victim's machine (perhaps through a virus or a remote code execution vulnerability in IE or any part of the Windows OS).

2. Get the grappling hook running with the user's privileges. This installs a Windows hook which gets the parasite injected into IE's address space.

3. The parasite changes IE's IAT so that calls to desired functions in the CryptoAPI (crypt32.dll and advapi32.dll) are redirected to the parasite.

4. At this point, we have complete control and are aware of what IE is trying to do. For example, if we specify `CryptSignMessage` to be redirected in our parasite, then every time IE calls this function (e.g., to do an SSL client-side authentication), control will pass to our code.

5. We know that the user is expecting to see a warning in this case, so we take advantage of the opportunity to do something nefarious—like export the private key. In our current demo, the adversarial code exports the private key, so the warning window will say "exporting" instead of "signing" at the top[2]. This potential detection mechanism could be remedied by hijacking the call which displays the warning. In fact, doing so would allow us to disable all such warnings, but we did not implement this UI hijacking.

[2] In our demo, we fail the IE request, so the user sees a "404" error.

Non-exportable Keys The bottom line is that with a DLL and small executable running with the victim's privileges, the private key—even with medium-security or high-security protections—can be stolen if it is exportable. The obvious solution is to make keys non-exportable, and we verified that this countermeasure prevents the attack.

2.4 Experiment 2: Malicious Use of Keys via Content-only Attacks

Since making keys non-exportable stops outright theft, we had to revise our attack strategy. We wondered if we could just use the key at will without having to actually steal it, as this would be just as devastating and would work against non-exportable keys. So we began our second experiment with the question: "Can we exploit the complexity of the language of interaction in order to use the key as we wish, even if we are limited to just serving content to the browser?"

2.4.1 GET Requests

The language of Web interaction—even when restricted to HTML only, and no Javascript—makes it very easy for a server S_A to send content to a browser B that causes the browser to issue an arbitrary request r to an arbitrary server.

If one wants this request r to be issued over SSL, we have found that a reliable technique is to use the HTML `frameset` construction, itself offered over server-side SSL. Figure 2.2 sketches this scenario. To borrow client-side authentication, the adversary needs to convince the browser's user to visit an SSL page at the evil server. Using the ordinary rules of Web interaction, the evil server can provide content that causes the browser to quietly issue a SSL request, authenticated with the user's personal certificate, to the victim

server.

Figure 2.3 shows some sample HTML for implementing the attack using a `GET` request. HTML permits an adversarial server to send a frameset to a browser. The browser will then issue requests to obtain the material to be loaded into each frame. A deviously crafted frameset (such as the one in Figure 2.3) appears to be an ordinary page. If an adversarial server includes a form response in the hidden frame, the browser will submit an SSL request to an arbitrary target server via `GET`. In many scenarios, browsers will use client-side authentication for the `GET`; with the devious frameset, the user may remain unaware of the request, the use of his personal certificate, and the response from the target.

Basic Techniques A `frameset` enables a server S_A to specify that the browser should divide the screen into a number of frames, and to load a specified URL into each frame. The adversarial server can specify *any* URL for these frames. If the server is careful with frame options, only one of these frames will be visible at the browser. However, the browser will issue all the specified requests.

This behavior appears to violate the well-known security model that "an applet can only talk back to the server that sent it" because this material is not an applet.

We stress that this is different from full-blown cross-site scripting. S_A is not is using a subtle bug to inject code into pages that are (or appear to be from) other servers. Rather, S_A is using the standard rules of HTML to ask the browser to itself load another page.

Framesets and SSL Ye and Smith noticed that if server S_A offers a frameset over server-side SSL, but specifies that the browser load an SSL page from S_B in the hidden frame, then many browser configurations will happily negotiate SSL handshakes with both servers—but the browser will only report the S_A certificate [137].

So, we wondered what would happen if S_B requested client-side authentication. In

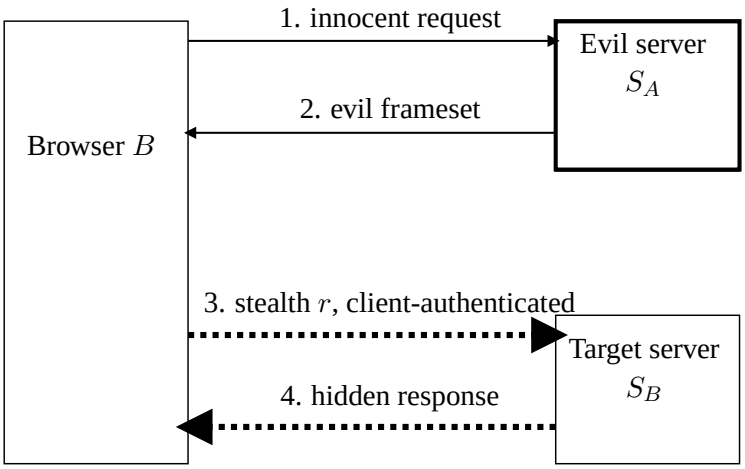

Figure 2.2: The setup for borrowing client-side authentication.

Mozilla 1.0.1 on Linux (RedHat 7.3 with 2.4.18-5 kernel), using default options, the browser will happily use a client key to authenticate, without informing the user. In IE 6.0/WindowsXP, using default options and any level key, the browser will happily use a client key to authenticate without informing the user, if the user has already client-side authenticated to S_B. If the user has not, a window will pop-up saying that the server with a specified hostname has requested client-side authentication; which key, and is it OK? In Netscape 4.79/Linux (RedHat 7.3 with 2.4.18-5 kernel), using default options, the browser will pop-up a window saying that the server with a specified hostname has requested client-side authentication; which key, and is it OK? Then the browser will authenticate.

The request to S_B can easily be a `GET` request, forging the response of a user to a Web form.

2.4.2 POST Requests

Some implementors preach that no sane Web service should accept `GET` response to Web forms. However, services that use `POST` responses are also vulnerable. If we extend the adversary's tools to include Javascript, then the adversarial page can easily include a `form`

```
<html>
<frameset rows="*,1" cols="*,1" frameborder="no">

<frame src="f0.html" name="f0" scrolling="no">
<frame src="blank" name="b0" scrolling="no">
<frame src="blank" name="b1" scrolling="no">
<frame src="https://cobweb.dartmouth.edu:8443/cgi-bin/test.pl?
    debit=1000&
    major=None%3B%20I%27m%20withdrawing%20from%20the%20college"
   name="f1" scrolling="no">
</frameset>
<noframes> no frames </noframes>
</html>
```

Figure 2.3: Borrowing client-side authentication via a GET request.

element with default values, and an `onload` function that submits it, via an SSL `POST` request, to S_B.

Figure 2.4 sketches an attack using the `POST` method. A web page such as this uses Javascript to cause the browser submit an SSL request to an arbitrary target server via `POST`. In many scenarios, browsers will use client-side authentication for the `POST`. If an adversarial server specifies that this page be loaded into a hidden frame, then the user may remain unaware of the request, the use of his personal certificate, and the response from the target.

2.4.3 Implications

As we noted earlier, it is continually touted that client-side SSL is superior to password-based authentication. Suppose that the operator of an honest server S_B offers a service where authorization or authentication are important. For example, perhaps S_B wanted to prove that its content was served to particular authorized parties (and perhaps to prove that those parties requested it—one thinks of Pete Townshend or a patent challenge), or perhaps S_B is offering email or class registration services, via `form` elements, to a campus

```
<html>
<head>
<SCRIPT LANGUAGE=javascript>
  function fnTemp()
   {
     document.myform.submit();
   }
</script>
</head>
<body onload="fnTemp()">

<form name="myform" method="post"
  action="https://cobweb.dartmouth.edu:8443/cgi-bin/test.pl">
<input name="debit" value="1000">
<input name="major" value="Hockey">
<input type="submit" value="Submit Form">
</form>
</body>
</html>
```

Figure 2.4: Borrowing client-side authentication via a POST request.

population. If S_B had used server-side SSL and required basic authentication or some other password scheme, then one might argue that a service can be executed in a user's name only if that user authorized it, or shared their password.

However, suppose S_B uses "stronger" client-side SSL. With Mozilla and default options, a user's request to S_B can be forged by a visit to an adversarial site S_A. With IE and default options, a user's request can be forged if the user has already visited S_B.

We note that this authentication-borrowing differs from the standard single-sign-on risk that, once a user arms their credential, their browser may silently authenticate to any site the user consciously visits. In our scenario, the user's browser silently authenticates to any site of the adversarial site's choosing.

We could not demonstrate a way for the adversary, using the tools of sending standard HTML and Javascript to users with standard browsers, to forge a response to a file upload `input` tag (see further discussion below) or to forge `REFERER` fields (although `telnet`

links look promising).

2.4.4 Browser Configurations

The answer to our question for this experiment was: "Yes, with most standard out-of-the-box configurations, we can use the key without the user's permission." The seemingly natural defense to such attacks is to properly configure the browser to avoid them, and indeed, actions such as disabling Javascript lowers the risk of a successful attacks.

2.5 Experiment 3: Malicious Use of Keys via API Attacks

IE on Windows is by far the dominant client platform. In trying to establish such a proper browser configuration for IE, we noticed that IE would only prompt for a password on our high security key once per visit to a particular domain. Specifically, we would visit site A, perform a client-side authentication which prompted us for the password, leave site A, and then return—*only we were never prompted for the key's password again*. Furthermore, we could not find any IE browser configuration which would enforce this behavior (even the DoD guidelines leave browsers susceptible [49]), and we eventually discovered that Microsoft considers the advertised behavior to be a bug [135].

The inability to configure our browser so that the advertised behavior of a high-security key (which reads "Request my permission with a password when this item is to be used") led us to believe that the flaw must be at a lower level. So we began our third experiment with the question: "Can we use some of our previous techniques such as API Hijacking to understand what is happening and then to use the key without the owner's permission?"

2.5.1 The Default CSP is Broken

The first step was to convince ourselves that IE was really using our high security key to perform client-side authentication without requesting our permission, and watching network traffic with a sniffer confirmed our suspicion. We then attempted to reproduce the behavior we observed. Using API Hijacking, were were able to attach a debugger to IE and watch the parameters it passes to the CryptoAPI. Reverse engineering in this way allowed us to build a standalone executable which made the same sequence of calls to the CryptoAPI as IE does and uses the same parameters.

Our program opens the same keystore IE uses during a `CryptAquireContext`. Our code sits in an infinite loop taking a line of input from the command line. It then mimics the sequence of calls that IE makes to the CryptoAPI in order to get data signed: `CryptCreateHash`, `CryptHashData`, and `CryptSignHash`. Since our key is a high security, the first call to `CryptSignHash` prompts for a password, as expected. However, no subsequent calls prompt for a password, even if the data is completely different. Thus, the CSP is failing to "request my permission with a password when this item is to be used."

2.5.2 The Magic Button

In all of our explorations of the various IE configuration options, we came across one button in the "Internet Options" menu labeled "Clear SSL State". API Hijacking showed that this button will make IE call `CryptRelaseContext` in the CryptoAPI, resulting in a password prompt the next time the key is used. We also found that restarting the browser will result in a prompt the next time the key is used (in contrast, we were initially surprised that restarting the Web server did *not* result in another prompt).

These are more extreme measures than simply configuring the browser to behave rea-

sonably, but they were the best we could find, and are recommended by Microsoft [48].

2.5.3 Exploiting the CSP to Get Around the Magic Button

Armed with a little information as to how the CSP and IE work, we were curious to see if there was a way to defeat the magic button and browser restarts. Our goal was to make a program—running only with user privileges—which waits for IE to prompt the user to arm his high security key with the password, and then use the key to sign arbitrary messages—even after IE has been closed, or SSL state has been cleared.

Using our previous technique of API Hijacking, we reused the grappling hook from the attack in Section 2.3. However, the parasite is slightly different than the one mentioned in Section 2.3. In this attack, the parasite will spawn an *agent* process (called `iexplorer.exe`; the real IE is `iexplore.exe`) which communicates with the parasite over a named pipe. When IE goes away, the agent will persist and be able to use the key without prompting for the password.

The initial stages of the attack are identical to the one in Section 2.3. The attacker begins by getting the grappling hook and parasite on the victim's machine. Once the code is in place, the grappling hook begins executing, which will get the parasite injected into IE address space. Upon injection, the parasite changes IE's IAT so that calls to desired functions in the CryptoAPI (advapi32.dll and crypt32.dll) are redirected to the parasite.

Once the parasite is set up, it watches for IE to make a specific sequence of calls: `CryptAcquireContext`, `CryptCreateHash`, `CryptSetHashParam`, followed by two calls to `CryptSignHash`. This sequence indicates that IE is using a private key for the first time, which will result in a password prompt. As each of these calls occur, the parasite intercepts the call before the CryptoAPI has a chance to handle it, packs all of the arguments into a binary data structure, and passes them to the agent over the named

pipe. The idea here is that the agent is mirroring the exact sequence of calls (with the exact arguments) that IE is making.

When IE makes the first call to `CryptSignHash`, it is trying to get the size needed to store the signature. As with many Windows API functions, the client (IE in this scenario) makes the first call to get the size of the hash, allocates memory, and then makes a second call with the new chunk of memory so that the API can fill in the memory chunk with the signature.

Our dear parasite listens for this call (and knows that it is the first call). It forwards the parameters to the agent and waits for a reply. The agent receives the data and calls `CryptSignHash` in order to get the size of the signature. The agent packs the return value (i.e., the size) into a structure and sends a reply to the parasite.

Once the parasite receives the reply, it sets the arguments appropriately, making it look as though the CryptoAPI call actually happened and returned a size. However, the parasite does *not* allow the call to pass through to the CryptoAPI. The parasite simply returns true, making IE think that the call has succeeded (and of course, the return arguments are correct).

The attack is completed when the second call to `CryptSignHash` occurs, as this is the call that spawns the password prompt. When this call occurs, the parasite does not actually let the call pass through to the CryptoAPI; it has the agent sign the data instead. The result is that the agent is the program which is requesting the user password, so it may use the key indefinitely—*with no further password prompts*. IE gets the correct signature, so the SSL handshake (or other key operation) continues as normal. Examining the "Details" of the signing operation shows that "iexplorer.exe" (the name of our agent executable) is using the private key instead of "iexplore.exe"; this subtle name change is the only means of detection.

At this point, the user can close IE and the agent still has an "armed" key, which it can

use indefinitely. Our example agent puts itself into command line mode, allowing us to sign and decrypt arbitrary messages with the victim's key. A real attack would most likely have the agent act as a Trojan where it binds to a port and awaits remote signing and decryption commands.

2.5.4 The Punchline: No Configuration Prevents This Attack

We were unable to find any browser configuration which stops this attack because the problem is below the browser—it is with the CryptoAPI and the default CSP. The attack is possible because the system is designed with the assumption that the entire system is trusted. If one small malicious program with user privileges (such as can happen by a user clicking on an unknown attachment) finds its way into the system, the security can be undermined—even with high-security non-exportable keys, and even assuming everyone does the right thing, no matter how awkward: browser users clear SSL state or kill the browser after each session, and server application writers use forms with hidden nonces.

2.6 Experiment 4: Malicious Use of Keys on a USB Token

Many in the field suggest getting the private key out of the system 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 the just the device and the software which provides access to the device (i.e., its CSP) need to be trusted.

We had a couple of these devices (the Aladdin eToken and the Spyrus Rosetta USB token), so we decided to have a critical look at these. Since the keys on the devices we had were non-exportable, key theft seemed impossible (assuming we leave "rubber hose

cryptanalysis" and hardware attacks out of our attack model), but we wondered if we could use the key as in the previous attacks.

Spyrus Rosetta USB The Spyrus CSP was the most verbose one in all of our experiments. It was the only CSP which prompted every time the key is used. In our opinion, these devices work too well—it was not uncommon to get multiple password prompts while loading one page.

(This suggests a further line of inquiry: why does the actual usage of a client private key in such a session depart so radically from the user's perception of it? That is: visiting a site "once" should generate one warning, not an endless barrage.)

While the Spyrus CSP allows users to render the best mental model, the model it renders is not a particularly usable one. Our hypothesis is that we could simply ask for the password outright, and would probably get it because users are so trained to enter their password for this device—we could probably just hide in the noise.

Upon contacting Spyrus about the issue, they pointed us to the "Spyrus Rosetta CSI library." The library supports a "Policy Console" feature which allows the CSP to use the key for a user-specified time interval without asking for permission. This feature clearly enhances the usability of the device, but opens a window of opportunity for an attack.

Aladdin eToken The Aladdin eToken did not give us any option as to how often we wanted to be prompted for a password, and experiments showed that the Aladdin CSP's default behavior seems to follow a policy of one password authentication per application. This is virtually the same behavior we saw with the default Microsoft CSP and the high security key. In fact, it is a bit worse than the default CSP in that the "Clear SSL State" button has no effect on the token whatsoever. Within a few minutes, we were able to replicate the attack in Section 2.5, allowing us to use the key even after the intended application (e.g.,

IE) has been shut down.

Upon discussing the issue with Aladdin, they suggest setting a "Secondary Authentication" to prompt upon application request. This does indeed prompt for authentication each time the key is used. As discussed previously, the result is possibly multiple password prompts per page load.

2.7 Conclusions

For desktops to be usable as PKI clients, they must behave as expected—they must only allow transactions which the client is aware of and approved. If we trust the entire desktop, users "clear SSL state" or kill their browsers after each session, and application writers include and verify hidden nonces, then we might conclude that client-side PKI works and that desktops are reasonable client platforms. However, these are not reasonable assumptions—and as we have demonstrated, relaxing them even a little yields security trouble.

The results presented in this chapter suggests that 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).

While our attacks focus primarily on IE and client-side SSL (as this is a widely used PKI paradigm), they can be used to subvert *any* program which relies on the CryptoAPI for key storage and use, and on many CSPs (as shown in Section 2.6). The reason for this is that the problems lie in the design of the CryptoAPI (as discussed in Section 2.5).

2.7.1 Usability

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.

It should be easy for a user to perceive and approve of the use of their private key, and it should be easy for an application developer to build on this. To cite just a few design principles: [141]

- "The path of least resistance" for users should result in a secure configuration.
- The interface should expose "appropriate boundaries" between objects and actions.
- Things should be authenticated in the user's name only as the "result of an explicit user action that is understood to imply granting."

One might quip that it is hard to find a principle here that the current paradigm does *not* violate. In order for client-side PKI to work (and for desktops to be suitable for use as PKI clients), these principles should apply to both the client user as well as to the IT staffer or application developer setting up a Web page or designing a Web application.

The current paradigm makes it difficult or impossible for a user to construct an accurate mental model of the system. For constructs such as warning windows to be effective, the screen material to which it applies should be clearly perceivable by the user. Even

adopting the “Basic Authentication” model of letting the server demanding the authentication provide some descriptive freetext might help. Instead of “hostname wants you to authenticate,” the browser window might give some context, e.g., “...in order to change your class registration—are you sure?”. (Netscape’s Signed Forms went in this direction, but it permitted the server to provide HTML content that can enable some types of signature spoofing.)

To rephrase a point from our earlier work [137], the community insists on strict access controls protecting the client file system from server content, but neglects access controls protecting the user’s perception of the client user interface.

2.7.2 The Trusted Computing Base

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 support the system’s security policy—in this case, protect private keys. This set of software is often referred to as the TCB. A good discussion of the TCB can be found in the “Orange Book” [90], 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 by reducing the amount of code that they must get right. How can we shrink the trust boundary so that buggy desktops which have frequent “Critical Security Updates” are not the cornerstone of our secure systems? Trusting just the kernel does not solve the problem. Trusting a separate cryptographic token does not solve the problem.

Placing a private key on a complex system such as the modern desktop is problematic. As this chapter has illustrated, by exploiting the complexity, an attacker can trick users into giving away their keys 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 does not shrink the TCB. The token's CSP is still interacting with the whole system (the OS and CryptoAPI), and thus 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 IBM 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 [63, 77, 120, 122]).

Tokens with UI On a system level, we recommend that further examination be given to the module that stores and wields private keys: perhaps a trustable subsystem with a trusted path to the user. As a device which has a very rich and complex interaction with the rest

of the world, browsers can often behave in unexpected and unclear ways. Thus, browsers should not be the cornerstone of a secure system.

Many researchers have long advocated that private keys are too important to be left exposed on a general-purpose desktop. We concur. However, in light of our experiments, we might go further and assert that the user interface governing the use of the private key is too important to be left on the desktop—and too important to be left to the sole determination of the server programmer, through a content language not designed to resist spoofing. As we will discuss in Chapter 5, SHEMP attempts to give users a trusted path to their keystore.

2.7.3 Immobility

In addition to the usability, 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 exacerbate 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 tokens with them across devices and computing environments. This approach has a number of drawbacks. 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 (as discussed in Section 2.6). 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) by exporting the key to some intermediate format (e.g., PKCS#12 [110]) and then importing

the key at the destination. This approach suffers a number of drawbacks as well. First, some devices may not support the media—e.g., we are unaware of cell phones with floppy drives, although they may have Bluetooth or smartcard support. Second, as described in Section 2.3 the intermediate format may be insecure.

2.7.4 Threat Model

The experiments described in this chapter show that the natural mental model which arises for client-side PKI is not representative of the actual system's behavior. This fact, coupled with the underlying assumption that all of the system's components are trusted, creates opportunities for a number of devastating attacks.

The attacks described in this chapter assume that the attacker can get an executable to run on the victim's machine and with the victim's privilege level. Furthermore, the victim does not need to be an Administrator or "root" user. For the remainder of this thesis, we will define the minimum threat model to be an attacker which can run arbitrary code on the victim's machine with the victim's privilege level. As we will discuss in Chapter 3, secure hardware can minimize the impacts of a successful attack under this threat model, and can be used to defend against stronger threat models, such as where an attacker has physical access to critical servers.

Chapter 3

Establishing 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. As we saw in Chapter 2, modern desktops fail in all of these categories. Drawing on the lessons learned from previous attempts, we can begin to understand what it takes to make a successful solution.

In this chapter, we introduce criteria which must be met in order to make desktops usable PKI clients. In addition to defining a set of properties which a successful solution must possess, we will define those properties in a context-meaningful way, and explore other approaches which attempt to provide each property.

Since our aim is to build an actual system that meets the criteria, we must give some level of consideration to practicality. In order for a proposed solution to be of any practical interest, it must safely store and use private keys, give application developers flexibility while maintaining security, match the model of real world user populations, and allow relying parties to make reasonable trust judgments about the system. Additionally, we cannot expect to rewrite the desktop OS or even decouple applications such as IE from the OS, as such tasks are infeasible with modern ubiquitous, closed-source OSes.

Chapter Outline In the next section, we discuss and define the first property that a proposed solution must possess: security. We also examine a number of other relevant approaches which attempt to make PKI systems more secure. In Section 3.2, we establish the second property: usability. In Section 3.3, we examine the mobility requirement, and discuss previous attempts to make PKI users mobile. Section 3.4 briefly discusses the need to for relying parties to be able to make reasonable trust judgments (Chapter 8 is devoted to this topic, so it is only briefly discussed in Section 3.4). Finally, Section 3.5 concludes.

3.1 Security

The notion of security is difficult or impossible to quantify 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, we give a pragmatic definition of security in this thesis. We let the operating definition of security in this thesis involve minimizing the risk, impact, and window of opportunity for misuse of a user's private key.

3.1.1 Minimizing the Risk of Key Disclosure

The TCB and Security We define the TCB for PKI applications to be the private key and the set of software which stores and uses the private key directly (e.g., libraries that make up constructs such as the CryptoAPI). The security trouble of Chapter 2 results from the fact that this set of software is intertwined with the OS and applications (such as Internet Explorer), and no clear boundaries exist. The result is that the entire system must be trusted in order for the system to be secure; just one well-crafted piece of malware can subvert the entire desktop, rendering it ineffective as a PKI client.

The inversely proportional relationship between TCB size and security is no new discovery. Again, as stated in Chapter 2, secure system designers have stressed the importance of keeping the TCB small since the "Orange Book" days of the 1970's [90]. A small TCB reduces the attacker's target and reduces the likelihood that a program error will result in a security flaw. Thus, a small TCB as defined in this thesis can decrease the risk of an attacker stealing or misusing a legitimate user's private key.

The TCB and Secure Hardware Secure hardware can reduce the size of the TCB. Highly secure devices such as the IBM 4758 [18, 119] can effectively create an entirely separate security domain from their host. Since the device has a general-purpose OS, it can house applications as well as critical data (e.g., private keys), thus eliminating the need for the private key to come in contact with the host desktop at all. The end result is that the entire TCB can be placed in such a device and be totally protected under the threat model of this thesis (defined in Section 2.7).[1] However, as discussed in Chapter 1, devices like the IBM 4758 are expensive, which prohibits their widespread use on client platforms.

While few devices can isolate the TCB to the extent of the IBM 4758, other devices can reduce the TCB to a level which is still acceptable under the given threat model. For instance, Hardware Security Modules (HSMs) can get the key off of the desktop, perform key operations internally, and even protect against physical attack. However, the applications may still live on the desktop, leaving the device only as secure as the CSP.

Other approaches, such as our Bear [67, 72, 73] project and others [106], can extend a weaker level of security to the entire desktop. However, this is sufficient given the current threat model that an attacker can run code on the victim's machine with the victim's privileges.

A more detailed examination of the IBM 4758 and Bear approaches will be given in

[1] In fact, the IBM 4758 is secure against much stronger threat models, such as one in which an attacker has direct physical access to the device.

Chapter 4, but the relevance to this discussion is the fact that secure hardware can reduce the size of the TCB, and thus decrease the risk of private key disclosure.

A Solution Should use Secure Hardware When Available 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*. We envision repositories to be dedicated machines which run no extraneous processes, have no shared filesystem, and are not remotely administratable. Ideally, repositories should be dedicated servers (i.e., not running any other applications) which are armed with some level of secure hardware and which are locally administered by a specialized individual. We consider clients to be standard desktop machines that can service one user at a time. In terms of secure hardware, we envision a potentially heterogeneous environment where machines can have very secure hardware such as an IBM 4758 secure coprocessor, less secure hardware such as our Bear platform, or no secure hardware at all.

Organizations which aim to provide high levels of security will adopt a threat model which gives the attacker more power than what is assumed in this thesis. Under such a threat model, key repositories should be able to withstand a wide range of attacks. For instance, an organization may wish to assume that an attacker can get root privileges on the repository's host machine. This would imply that the attacker can watch any process's memory, and run any code of his choice on the host. Furthermore, the organization may wish to assume that an attacker has physical access to the secure hardware holding the private keys and can attempt to perform local hardware attacks. As a result, that organization's repository 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 IBM 4758 is achieves a high level of security, a general solution should be flexible enough to deal with any type of hardware on the repository. No matter what type of hardware is used, the system should be able to give all parties enough information about the repository so that it can make informed decisions.

The threat model for clients may be different. Again, a successful general solution should allow for such variations and 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 in SHEMP will be discussed in Chapter 5.

Moving the TCB for PKI Applications off of the Desktop Roughly speaking, the larger the desktop-resident TCB, the greater the resulting risk of private key disclosure. If we assume that client machines will be running standard OSes, then we should minimize the amount of the PKI TCB that resides on the client machine. The ideal scenario is one in which no part of the PKI TCB comes into contact with the client machine, although this approach also makes desktops unusable as PKI clients (as no part of the desktop—including applications—can be used in any PKI operation). A compromise could consist of keeping the PKI TCB out of the reach of a desktop until some portion of that TCB is needed, and only then, would the desktop be considered part or the PKI TCB.

3.1.2 Minimizing the Impact and Misuse Window

In Section 3.1.1, we explained how reducing the size of the TCB can decrease the risk of private key disclosure. In order to minimize the impact of a key disclosure and the window of opportunity for an attacker to misuse the key, we need some way to control the lifespan during which a compromised key can be used. A short key lifespan reduces the opportunity for misuse.

However, just issuing short-lived private keys to the population would increase the (al-

ready cumbersome) administrative burden of the PKI, as users would have to be re-keyed frequently. To remedy this, a number of systems rely on *delegation* to control the lifespan of a user's credentials. While the delegation frameworks differ across the types of credentials (e.g., some frameworks rely on short-lived Proxy Certificates [126, 132] while others rely on *Simple Distributed Security Infrastructure/Simple PKI* (SDSI/SPKI) certificates [20, 23, 24]), they share the same goal: issue a temporary credential which, when evaluated in conjunction with a long-term credential such as a PKI certificate, allows a relying party to make a reasonable trust judgment.

The above approach allows organizations to shrink the TCB by placing the long-lived private key in a secure place such as a credential repository. Then, when a client needs a short-term credential, the TCB is effectively extended to include the short-term credential—*but only for a short period of time.* Once the short-term credential expires or is destroyed (possibly by logging out), the TCB shrinks back to its initial size (i.e., the size before the user requested a short-term credential).

Should a short-term credential be compromised by an attacker, the attacker can only misuse the temporary credential for a short period of time, thus minimizing the impact and window of opportunity for misuse. As we discussed in Chapter 2, the status quo allows an attacker to misuse the victim's private key indefinitely—even across application restarts.

3.1.3 Relevant Approaches

There are a number of systems in existence which aim to shrink the TCB as it is defined in this thesis. However, many of them do not accommodate the use of secure hardware to decrease the risk of key disclosure. Nevertheless, there are numerous lessons to be learned from the collection of prior art.

MyProxy As briefly introduced in Chapter 1 (and discussed in detail in Chapter 4), the Grid community's MyProxy credential repository [88] minimizes the risk of key disclosure by removing the user's private key from a client machine and placing it in a repository. The MyProxy repository minimizes the impact and misuse window through the use of Proxy Certificates as a delegation framework. Lorch et al. have even investigated placing the private keys inside of a secure device (i.e., the IBM 4758) [64].

One place where MyProxy falls short of the criteria outlined in Section 3.1.1 is in the lack of support for secure hardware at the clients. The MyProxy system (even with the aid of an IBM 4758 on the server) does not utilize the secure hardware available on the client's machine for identification and/or temporary key generation and storage purposes. Additionally, MyProxy assumes that all clients are equal with respect to security, and thus does not provide any mechanism for users or applications to reason about the security properties of client machines.

Furthermore, even the hardened MyProxy [64] system (i.e., where the repository uses an IBM 4758 for key storage) only places the user's private key in secure hardware on the repository, as opposed to the entire system. Since the IBM 4758 is a general-purpose computing device, it could actually run the repository software itself. Since a hardened MyProxy repository only places the private keys in the coprocessor, it effectively leaves other portions of the TCB (such as the cryptographic libraries and applications) on the host. Putting the entire TCB in the device would allow 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 *Internet Engineering Task Force* (IETF) standards and workings of Kerberos can be found in the literature (e.g., [54, 108]). 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 itself and each party (often called a *principal*) in the system. When principals would like to communicate, the KDC generates a short-lived shared secret (i.e., a session key) and distributes that to the principals.

Kerberos is not quite a general purpose PKI platform, as it relies on secret key cryptography. Kerberos is mainly an authentication framework, whereas the scope of this thesis is more tuned toward traditional PKI systems which could enable secrecy and signature applications (e.g., S/MIME) as well as authentication.

As with MyProxy, there is a lack of accommodation for secure hardware in the general Kerberos design. Some research has looked into placing the KDC inside of an IBM 4758 secure coprocessor [51], but as with MyProxy, this just places a portion of the TCB inside the secure hardware, leaving cryptographic libraries and applications outside. Additionally, there are no provisions to take advantage of secure hardware on the clients. Since Kerberos stores credentials in memory, it is susceptible to the attacks described in Chapter 2.

Shibboleth Shibboleth is an Internet2 project which aims to develop a middleware that supports user authorization and authentication. Nazareth et al. [83, 84] give 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, while also accommodating any licensing agreements A has with B. 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.

As with Kerberos, 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 Shibbo-

leth Attribute Requester). Shibboleth is, in essence, a privacy-preserving authentication framework.

As with the other systems considered in this section, 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, even if it is present.

Trusted Third Parties One 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 a *Trusted Third Party* (TTP) [95] . 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. TTPs which aid in performing key operations are sometimes referred to as *Semi-Trusted Mediators* (SEMs) [10, 17].

TTPs and SEMs both decrease the risk of private key disclosure by removing all or part of the private key from the desktop. SEMs typically split the user's private key and place a share of it in the SEM, and recent work has even explored adding secure hardware to SEMs [129].

The major differences between the MyProxy approach and SEMs is that MyProxy places the entire key inside of a repository, as opposed to a portion. This results in the repository becoming a "fully-trusted mediator." Second, MyProxy's 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.

Greenpass The Greenpass project [34] is a delegation framework which uses SDSI/SPKI delegation on top of X.509 certificates in order to authorize guests to an organization's

wireless network. The idea is for a member of the population (Alice) which holds an X.509 certificate to delegate (i.e., by signing a temporary SDSI/SPKI certificate) to her friend Bob so that he can access her organization's resources for some relatively short amount of time, even though he is not a member of Alice's organization.

While there are similarities in the delegation framework between MyProxy and Greenpass, there are a number of important differences. First, there is no mention of secure hardware in Greenpass, and no way (within the current implementation) for the Greenpass system to relay information regarding secure hardware to relying parties. As we will discuss in Chapter 4, Proxy Certificates (the delegation framework used in MyProxy) provide a hook for including such security information.

Second, there are no constraints on where users store their key under the Greenpass system. There is no mention of a key repository, thus allowing users to store their private keys on their desktop (in a browser keystore) or on a USB token of some sort. As demonstrated in Chapter 2, storing private keys in such places can lead to security trouble.

3.2 Usability

The second feature that a proposed solution should provide is usability. Users, administrators, and application developers must be able construct accurate mental models of the system. As we saw in Chapter 2, failure in this area yields security trouble and can render the system useless.

In order for a proposed solution to be of any practical interest, it must work within the constraints of the modern desktop. It is unreasonable to assume that a general solution can require system-level software such as the OS or CryptoAPI to be redesigned and/or rewritten. A successful solution for making desktops usable PKI clients must account for the fact that the underlying desktop platform is insecure and attempt to provide as much

security as possible, given that constraint. While this approach is not likely to produce a perfectly secure system (if such a thing exists), it increases the probability that a solution is usable in modern real-world PKI environments.

3.2.1 User Requirements

From a user's perspective, the system must be easy to use. One design strategy which can enhance the user's experience involves hiding the system's complexity from the user. Clearly, there is a balance to be achieved; hiding too much complexity can have adverse effects, as can exposing too much. Ideally, the system should hide enough complexity so that users are not overwhelmed by configuration options (in which case, they are likely to misuse and/or misconfigure the system—most likely resulting in security trouble). However, enough complexity should be visible so that users can construct a valid mental model of the system.

In Chapter 2, we illustrated how IE fails this requirement. The proper way to use the system (i.e., using the "Clear SSL State" button or restarting the browser after each visit to a secure site) is somewhat complex and non-obvious. To make matters worse, too much complexity is hidden in certain places (i.e., our high-security private key does not ask for a password each time it is used, thus violating its advertised behavior). As a result, it is impossible for users to render an accurate mental model of what the system is really doing.

In addition to hiding complexity, a successful solution must present a set of well-defined operations to the users. Ideally, the operations should not have unexpected or unnoticeable side-effects, as they can also make it impossible for the user to construct an accurate mental model.

Last, the user interface should be straightforward, and should make the system difficult or impossible to use incorrectly. Again, the basic design principles indicate that the "path of

least resistance" should result in a secure configuration, and the the interface should expose "appropriate boundaries" between objects and actions [141]. Chapter 2 and Whitten and Tygar [134] illustrate the impact of failing to follow these principles.

3.2.2 Administrator Requirements

The requirements for an administrator's view of the system can be different. In many systems, the administrator is a special entity who has a deeper knowledge of the system, and as a result, can be burdened with some of the system's complexity. In fact, in many scenarios, it is the administrator's role to insulate the end user from the system's complexity. However, in order to deal with this complexity, administrators should be given tools which aid in system configuration and use. Furthermore, the toolkit should make it difficult for administrators to do the wrong thing, and easy for them to do the right one.

One feature which can make a system usable to administrators is flexibility. In the context of this thesis, administrators should have the ability to define organization-specific properties such as notions of security. For instance, organization A may refer to machines behind their firewall as secure, while organization B may define secure machines as ones patched within the last week. Giving administrators such flexibility can make the system more usable, provided that the mechanisms which provide that flexibility are not too complicated to use.

3.2.3 Developer Requirements

In order to get parties to write applications for the system, it should expose common programming paradigms to developers, and allow them to use the solution to build and deploy real applications. This requires that the platform must be easily programmable with modern tools, and must also allow easy maintenance and upgrade of its software. Forcing develop-

ers to conform to awkward or constraining mechanisms limits the usability of the system from a developer's viewpoint.

Since the problem this thesis addresses involves making desktops usable for PKI, applications which are likely to be developed for such a system will be wielding or relying on users' private keys. Thus, such applications will likely need to reason about the security-specific state of the system and TCB. The underlying system should expose as much of that state (and as many of the properties of the TCB) as possible, so that applications can make informed decisions.

Last, as discussed in Section 3.1.1, secure hardware can reduce the risk of private key disclosure. To this end, a system which makes desktops usable for PKI should allow applications to take advantage of secure hardware, if present.

3.3 Mobility

The third feature that a proposed solution must provide is mobility. Modern user populations increasingly use multiple computing platforms at multiple locations. As noted in Chapter 1, many current PKI systems either make it difficult for the user to move their private key or make it vulnerable to attack during and/or after transit.

A PKI solution must allow users to move throughout their domain, and across their computing platforms. Most importantly, the solution should not put the private key at risk of disclosure any time the user moves geographically or uses different devices. A good solution should take into account the trustworthiness of the client platform, thus disallowing the key to migrate to untrustable client machines (or severely limiting its use).

In practice, the credential repository approach is closest to achieving the goal of mobility. However, we have not identified a credential repository which takes the trustworthiness of clients into account (this issue will be discussed further in Chapter 5).

3.3.1 Relevant Approaches

There are approaches which allow users to migrate their private key across platforms, in addition to the standard (and risky) "export/re-import" method described in Chapter 1.

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 [101] and Microsoft's Passport [79]—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.

Cookies fail to fully meet the mobility part of the criteria primarily because they are an application-specific construct, and are only usable by Web browsers. A general solution to making desktops suitable for PKI must be usable by a number of applications, including but not limited to, Web browsers.

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 [104], a draft describing a framework [102] and a protocol [103]. 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.

The Sacred project is a key migration system proper, not a general purpose PKI solution. One undesirable property of Sacred is that is requires that the private key move along with the user. If the user sits at a machine with some malware (such as our keyjacking software)

installed, then the user's key is easily compromised. Sacred provides the now infamous "armored car to a cardboard box."

3.4 Making Reasonable Trust Judgments

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, 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?).

Since such trust judgments are often complex and important, it is best to use formal methods to aid the decision-making process, and to determine if the system is sound. If the system is sound, then it should use Bob's private key if and only if Bob authorized the request.

A number of formal frameworks exist to reason about bindings between entities and keys and the protocols used to establish these bindings [12, 57, 59, 76]. The framework best suited to reasoning about PKI systems (as opposed to distributed systems in general) is

	MyProxy	Kerb	Shib	TTP	Grnpss	Cookies	Sacred
PKI system	yes	no	no	yes	yes	no	no
HW at server	maybe	maybe	no	maybe	no	no	no
HW at client	no	no	no	maybe	no	no	no
Key on desktop	no	no	yes	maybe	yes	yes	yes
Delegation	yes	yes	yes	no	yes	no	no
Mobility	yes	yes	no	no	yes	no	yes

Table 3.1: A brief summary of the relevant approaches.

the work of Ueli Maurer [76]. His "deterministic model" is simple and expressive enough to capture most of the design concepts employed by SHEMP. Maurer's deterministic model (and our model, which is based on Maurer's) will be discussed in detail in Chapter 8. We will also use our model to prove the correctness of SHEMP (and a number of applications) from a PKI perspective.

3.5 Conclusions

In this chapter, we have examined the issues which need to be addressed in order to make desktops usable PKI clients: security, usability, mobility, and the ability to reason about trust. Our focus is on practicality; rewriting the OS or critical system components is not an option. Furthermore, a successful solution should account for a wide range of platforms with an even wider range of security properties.

In the context of this thesis, a solution increases security by decreasing the risk, impact, and misuse window of a private key disclosure. Concretely, decreasing the risk of disclosure can be achieved by shrinking the size of the TCB, by getting it off of the desktop, and by taking advantage of secure hardware where available. Allowing the TCB to vary in time (possibly using delegation) can minimize the impact and window for misuse in the event of a private key disclosure.

As we explored in Chapter 2, an unusable system can quickly become an insecure one. To this end, any solution to the desktop PKI problem must be usable. Additionally, it is not sufficient for the system to only be usable by end users; administrators and application developers must be able to use the system as well. A successful solution should consider all of the parties involved in the system, and be designed to fit all of their needs.

In order to make desktops suitable PKI clients, system designers must consider the usage patterns of real-world computing populations. As users increasingly migrate between and across platforms, access to their private key should follow—without undermining the security of the system.

The last and most important property that a solution must provide is the ability for relying parties to make reasonable trust judgments about the system. This topic will be covered in detail in Chapter 8; without this property, none of the others are meaningful.

In addition to establishing criteria for solving the desktop PKI problem, we examined numerous approaches, and briefly discussed where they succeed and fail. The approaches are summarized in Table 3.1. In the next chapter, we introduce our solution, named SHEMP, and discuss its design and implementation in detail. In Chapter 7, we show how SHEMP meets the requirements outlined in this chapter.

Chapter 4

SHEMP Building Blocks

So far in this thesis, we have established the fact that modern desktops are not usable as PKI clients and we have uncovered the underlying causes. As we discussed in Chapter 2, modern keystores are susceptible to attacks which result in the attacker being able to either steal a user's private key or use it at will. Such vulnerabilities are the result of a large TCB which is formed by the tight coupling between the OS, keystore, and applications. Desktops are not usable as PKI clients because users must trust the entire desktop in order to trust the keystore.

In addition to discovering and demonstrating the problem, we have also established criteria which a solution to the problem must meet in order to be considered successful. In Chapter 3, we explained that, in order for a system to make desktops usable for PKI, it must increase security, give users mobility, be usable, and allow relying parties to make reasonable trust judgments. We also introduced a number of potential solutions to the problem and showed where each of them failed to meet the criteria entirely.

In the next chapters, we introduce our solution: SHEMP. The philosophy behind SHEMP is to use a credential repository to get keys off of the desktop and give users mobility. When users need their keys for some operation, they generate a temporary keypair on their desk-

top, and ask the repository to sign a short-lived Proxy Certificate (PC) containing the public portion of the temporary keypair. This approach confines the TCB to the repository until the user requests a PC, at which point the TCB expands to cover the desktop. The SHEMP policy framework potentially limits what can go wrong at the desktop, even if it is included in TCB. When the PC expires or the user destroys it (i.e., by logging out), the TCB is confined to the repository again. In order to strengthen security even further, SHEMP takes advantage of secure hardware, if available, on the repository as well as on the client desktop. As we will explore in Chapter 7 and Chapter 8, SHEMP meets the criteria of Chapter 3 and makes desktops usable PKI clients.

As we examined in Chapter 3, a number of existing systems satisfy parts of the criteria put forth in that chapter. Rather than reinvent solutions to fit each of the requirements of our criteria, we begin by examining the prior art and extending relevant ideas and designs to fit our needs. The relevant prior art comprises our toolkit and form the basic conceptual building blocks we use to build SHEMP.

Chapter Outline Our toolkit is made up of three categories. In Section 4.1, we discuss a *credential repository* (MyProxy) and *delegation framework* (PCs) which are used to get keys off of the desktops and give users mobility. In Section 4.2, we cover how *secure hardware* can be used as a basic keystore, both at repositories and clients when available. Section 4.3 describes a *policy language* which is used to express key usage and delegation policies at the repository as well as express attributes of repositories and clients. Finally, Section 4.4 concludes.

4.1 MyProxy and Proxy Certificates

The first component we use to build SHEMP is the MyProxy credential repository, which we use to shrink the TCB and give users mobility.

System Overview The Grid community's MyProxy system [88] is a credential repository designed to allow Grid users to obtain and delegate access to their credentials from multiple locations on the Grid. Early versions of the MyProxy system had users store their long-term credential (i.e., private key) on their desktop. Users would then generate a short-lived delegation credential (i.e., a PC, discussed below), and store it in the MyProxy credential repository. Then, when the user (or a process running on the user's behalf) moved locations, it could login to the MyProxy repository, and obtain the short-lived credential. This short-lived delegation credential, when used in conjunction with the long-term credential issued to the user (i.e., the user's X.509 identity certificate), could then be used to identify the user, or show that a process has some properties granted by the user.

More modern versions of MyProxy [64] take a slightly different approach. They use the MyProxy repository to store the long-term credential, thus getting the private key off of the user's desktop altogether (in fact, Lorch et al. [64] store the key in an IBM 4758 secure coprocessor at the repository). When a user (or process running on a user's behalf) needs to use a credential for authentication or authorization, it logs in to the MyProxy repository and requests that a short-lived PC be generated. As before, the PC along with the user's long-term credential can then be used for authentication or authorization.

The MyProxy system is attractive for two reasons. First, the latter version (which we base the SHEMP design on) gets the user's private key off of the desktop entirely, and thus shrinks the TCB. When a user or process needs to use a credential, the TCB expands to include the desktop (via delegation)—but only for short period of time. This approach

shrinks the TCB in space and time which, in turn, gives MyProxy a security advantage over the standard desktop PKI approach. (Another security advantage is that the the private keys are administered by a professional instead of end users.) Second, the MyProxy system gives users mobility. Since the user's private key is stored in a central location, it can be accessed from many locations without having to be transported by hand (i.e., exporting/re-importing or using a protocol like Sacred [102, 103, 104]).

As noted in Chapter 3, Kerberos and Semi-trusted Mediators (SEMs) are similar solutions which could have possibly served as a starting point for the SHEMP design. MyProxy stood out over Kerberos because it is a general-purpose PKI system, whereas Kerberos uses symmetric keys. In comparison to SEMs, MyProxy is attractive because it gets the private key off of the desktop altogether (SEM just gets a piece off), which shrinks the TCB further. Additionally, MyProxy allows for delegation via PCs; SEMs have no such notion. Finally, MyProxy's use of PCs allows the user to be offline when the PC needs to be used (by some entity which the user delegated to).

Proxy Certificates A detailed description of the SHEMP design will be given in Chapter 5, but the main idea is to store the user's private key in a credential repository, and use it to sign a short-lived delegation credential (i.e., a PC [126, 132]) which the client can use.

The delegation framework used for the short-lived keys and certificates should be simple, widely accepted in practice, and should not require extra infrastructure such as *Certificate Revocation Lists* (CRLs). There are numerous frameworks and certificate formats to choose from, such as X.509 [29], X.509 Proxy Certificates [126, 132], Permis [14, 15], Keynote Credential Assertions [9, 55], and SDSI/SPKI [20, 23, 24]. Nazareth et al. provide an overview of a number of these delegation systems, as well as comparisons between them [83].

We chose X.509 Proxy Certificates for a number of reasons. First, they are standard-

ized 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 [32]. Fourth, they allow dynamic delegation without the help of a third party, allowing clients to obtain a PC without having to endure the cumbersome vetting process at the Certificate Authority. Last, the PC 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, discussed below).

In all fairness, SDSI/SPKI certificates could probably be used as well. However, since MyProxy is based on PCs, and they suit our needs well, we decided to use them. Additionally, our lab has a produced a number of projects which use the SDSI/SPKI format [34, 83], and using PCs gives us an opportunity to explore something new. Furthermore, some of our lab's projects have encountered difficulties when passing SDSI/SPKI certificates to relying parties [34].

In terms of the criteria of Chapter 3, PCs are used to minimize the risk of key disclosure by allowing the TCB to expand and shrink over time. When a user does not need access to their credential, the TCB need only cover the repository. When the user needs to use their credentials from some desktop, then the TCB is expanded to cover the desktop. Concretely, this TCB expansion is achieved by issuing a PC for a temporary keypair stored on the desktop. When the user is finished (or the PC expires), the TCB effectively shrinks back to covering only the repository. Second, the use of PCs minimizes the impact of a private key disclosure. If an attacker compromises a desktop-resident private key (i.e., a key described by a PC), then the attacker can use the private key at will—but only for a short period of time since PCs have a short lifespan.

4.2 Secure Hardware

Over the years, our lab has built a number of systems which involve and/or enhance secure coprocessors. We have designed secure coprocessors, designed platforms which are hardened by coprocessors, and used secure coprocessors as foundations for applications. Secure hardware is interesting in the context of SHEMP because it can be used to reduce the size of the TCB, thus reducing the risk of a key disclosure.

4.2.1 The IBM 4758

Most of our initial systems were constructed around the IBM 4758, as Sean Smith brought it to our lab from IBM [18, 114, 118, 119]. Members of our group have used these devices to enhance privacy [46], harden PKI [52, 70, 115], and enhance S/MIME [94].

The IBM 4758 is a secure coprocessor which provides secure storage facilities, cryptographic acceleration, and a general-purpose platform on which to run third-party applications. The IBM 4758 is a very secure device, having been validated to FIPS 140-1 Level 4. It can withstand both software and hardware attacks, and can effectively provide a different security domain from its host machine. A good overview of the IBM 4758 and its capabilities can be found in the literature (e.g., [18, 114, 118, 119]). We briefly describe some of the features which are useful in the context of building a system such as SHEMP.

Modern versions of the IBM 4758[1] allow application developers to execute their applications (written in `C/C++`[2]) directly inside the device. Commercial versions ship with a general-purpose OS called `CP/Q++`, and experimental versions can even run the Linux OS. Since applications can be run directly inside the device, the IBM 4758 effectively allows

[1] Colleagues at IBM Research tell us there is a next-generation device called the IBM PCIXCC [2]. At the time this project began, no developer kits were available for experimentation. Our remarks and analysis of the IBM 4758 apply to an experimental version of the model 002 device.

[2] We successfully placed a small Java virtual machine in the device, but have not published those results.

applications to run in an entirely different security domain than the host desktop.

In addition to an execution environment, the IBM 4758 provides cryptographic services to applications. Using dedicated hardware (such as DES and modulo arithmetic engines), the IBM 4758 can perform symmetric and asymmetric cryptographic operations faster than many general-purpose desktops.

The device also offers a secure storage service to applications, allowing them to store cryptographic secrets such as private keys. The device is designed to only allow particular applications to access certain memory regions. This memory partitioning keeps processes from accessing one another's cryptographic secrets, thus enhancing process isolation.

The device is equipped with an internal PKI which it uses to perform what it calls *Outbound Authentication* (OA) (modern trusted computing platforms often refer to this feature as *attestation*). OA enables applications running inside of the device to cryptographically authenticate themselves to remote parties using PKI [116].

The strength of the IBM 4758 comes from the fact that all of the above components (the general-purpose execution environment, the cryptographic acceleration hardware, the secure storage service, and the OA subsystem) are housed in a tamper-resistant cage, and plugged into the host computer's PCI bus. Should an attacker attempt to physically open the device (or otherwise attack it to probe the memory for cryptographic secrets), the device will automatically clear the memory, erasing any secrets and possibly making it impossible for applications to authenticate themselves any longer.

As previously noted, the IBM 4758 effectively provides an entirely different security domain from its host computer. Data and applications inside the device never need to be exposed to the host. Clearly, such security comes with a price (approximately $3000 US), which prevents installation at every client.

4.2.2 Bear/Enforcer

In some sense, secure coprocessors such as the IBM 4758 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) [93, 123, 124, 125]) and Microsoft's *Palladium* (now renamed the *Next Generation Secure Computing Base* (NGSCB) [25, 26, 27, 28]). These efforts may benefit from Intel's *LaGrande* initiative [120].

These new initiatives target a different tradeoff: lower-assurance security that protects an entire desktop (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 [63, 77, 122] have also explored alternative approaches in this "use a small amount of hardware security" space.

More recent projects have involved constructing a "virtual" coprocessor out of commodity hardware. Our initial design and prototype is based on the TCG specification (see [93, 123, 124, 125]) and is called *Bear/Enforcer* [67, 72]. We discuss the Bear/Enforcer platform in depth, as it is what our prototype is built on.

The Basic Framework To start with, we need a way for Alice, working within existing hardware, software, and protocols, to reach some conclusion about a computation occurring on Bob's computer. The TCPA/TCG specification's *Trusted Platform Module* (TPM) gives us a basic tool (described below). However, this tool binds a secret to a specific full-blown software and data configuration on a given machine, which makes it difficult to deal with two problems:

- In most applications where a relying party Alice needs to authenticate a remote pro-

gram P on Bob's machine, the overall software and data configuration on a platform often needs to change (e.g., for upgrades), even though P remains the same.

- In current distributed security infrastructures, Alice wants to make her trust decision based on whether P proves knowledge of a long-lived private key matching a long-lived X.509 identity certificate, and Bob does not want to have to go back to a CA each time his software or data changes.

We addressed both problems by indirection. The TPM and boot process verifies that our *Enforcer* security module (described below) and supporting software is unmodified; the Enforcer then checks the more dynamic parts of the system against a configuration file signed by a (possibly remote) *Security Administrator*, Cathy. The TPM releases private keys to the Enforcer only when it boots correctly; but the Enforcer only releases the program private key when it satisfies the current configuration. Thus, by delegating configuration judgment to Cathy, a CA can issue a long-lived certificate to Bob's application.

The TPM We quickly review the basic functionality of the TPM that is currently available. More information on the TCPA/TCG technology can be found in our technical reports [67, 72], the TCPA/TCG specifications [123, 124, 125], and other literature [45, 93, 105], as well as `www.trustedcomputinggroup.org`.

The TPM in our commodity hardware has 16 *Platform Configuration Registers* (PCRs), each 20 bytes long. The TCPA/TCG specification reserves eight PCRs for specific purposes, leaving eight for applications. The TPM provides a *protected storage* service to its machine. From the programming perspective, one can ask the TPM to *seal* data, and specify a subset of PCRs and target values. The TPM returns an encrypted blob (with an internal hash, for integrity checking). One can also give an encrypted blob to the TPM, and ask it to *unseal* it. The TPM will release the data only if the PCRs specified at sealing

now have the same values they had when the object was sealed (and if the blob passes its integrity check).

It is also possible to create keys which are bound to a specific machine configuration with the `TPM_CreateWrapKey` function. This alleviates the need to create a key and then seal it, allowing both events to be performed by one atomic operation.

TPM protected storage can thus bind secrets to a particular software configuration, if the PCRs reflect hashes of the elements of this configuration. The TPM also has the ability to save and report the PCR values that existed when an object was sealed.

The TPM can perform RSA private-key operations internally. Besides enabling management of the key tree, this feature permits the TPM to do private-key operations with other stored objects that happen to be private keys (if the PCRs and authorization permit this) without exposing the private keys to the host platform. One special use of a TPM-held private key is the `TPM_Quote` command. If the caller is authorized to use a TPM-held private key, the caller can use the `TPM_Quote` command to have the TPM use it to sign a snapshot of the current values of the PCRs. Another useful feature of a TPM-held key is exposed via the `TPM_CertifyKey` call. This function allows a TPM-held private key to sign a certificate binding a TPM-held public key to its usage properties, including whether it is wrapped, and to what PCR values.

Certification TCPA/TCG provides additional functionality for tasks like proving that a TPM is genuine and *attesting* to the software configuration of a machine. The TCPA/TCG specification—and subsequent research [106]—lays out some fairly complex procedures. However, Alice does not want to carry out a complex procedure—she just wants to verify that a remote program knows a private key matching the public key in an X.509 certificate. Upon careful reading of the specification, it appears the TPM can provide equivalent functionality. We provide a new code module that has the TPM create what it terms an “identity

key pair" and then obtain an "identity certificate" from what we call YACA ("yet another CA"). This module then uses the TPM to create a wrapped key pair bound to a configuration which includes itself—and then has the TPM use the identity private key to certify that fact. Finally, the module needs to return to a standard X.509 CA (which could be the same YACA) with the identity certificate and the certificate created for this wrapped key pair, in order to obtain a standard X.509 certificate.

Threat Model The TCPA/TCG design cannot protect against fundamental physical attacks. If an adversary can extract the core secrets from the TPM, then they can build a fake one that ignores the PCRs. If an adversary can manage to trick a genuine TPM (during boot) to storing hash values that do not match the code that actually runs (e.g., perhaps with dual-ported RAM or malicious DMA), then secrets can be exposed to the wrong software. If the adversary can manage to read machine memory during runtime, then he may be able to extract protected objects that the TPM has unsealed and returned to the host.

However, the TPM can protect against many attacks on software integrity. If the adversary changes the boot loader or critical software on the hard disk, the TPM will refuse to reveal secrets. Otherwise, the verified software can then verify (via hashes) data and other software. Potentially, the TPM can protect against runtime attacks on software and data, if onboard software can hash the attacked areas and inform the TPM of changes.

Design Our goal is to bind a private key to program P. How do we permit Bob to carry out appropriate updates to the software that constitutes program P, without rendering this private key unavailable? How do we ensure a malicious Bob cannot roll back a patched program to an earlier version that we now know is unsafe? How do we permit a CA to express something in a certificate that says something meaningful about the trustworthiness of P over future changes—both of software as well as of more dynamic state?

In some sense, everything is dynamic, even X.509 key pairs. However, in current PKI paradigms, a certificate binds an entity to a key pair for some relatively long-lived period. But if this entity P is to be a remote program offering some type of service, the entity will have to change in ways that cannot be predicted at the time of certification. To address this problem, we decided to organize system elements by how often they change: the relatively *long-lived* core kernel; more *medium-lived* software; and *short-lived* operational data

As noted above, we add two additional items to the mix: a remote Security Administrator who controls the medium-lived software configuration via public-key signatures, and an Enforcer software module that is part of the long-lived core.

The Security Administrator signs a description of the medium-lived software which represents a good configuration of the medium-lived software. The Security Administrator's signed description acts as a security policy for the medium-lived software. For simplicity, the Security Administrator's public key can be part of the long-lived core (although we could have it elsewhere). A Security Administrator's security policy could apply to large sets of machines, and in theory, the Security Administrator may in fact be part of a different organization. For example, Verisign or CERT might set up a Security Administrator who signs descriptions of what are believed to be secure configurations of the program(s) in question, and distributes these descriptions to a number of organizations to use as a security policy. This approach allows one entity to bless the configurations for multiple sites without having to run all of the servers itself.

The TCPA/TCG boot process (via our modified boot loader) ensures that the long-lived core boots correctly and has access to its secrets. The Enforcer (within the long-lived core) checks that the Security Administrator's security policy is correctly signed, and that the medium-lived software matches this policy. The Enforcer then uses the secure storage API to retrieve and update short-lived operational data, when requested by the other software.

Our design binds the protected secrets to the Enforcer and long-lived core instead of

the the medium- and short-lived components of the system. This approach alleviates the need to get a new certificate each time the medium- or short-lived components change—presumably quite often.

To prevent replay of old signed policies, the Security Administrator could include a serial number within each description, as well a "high water mark" specifying the least serial number that should still be regarded as valid. The Enforcer saves a high-water mark as a field in a freshness table; the Enforcer accepts a signed policy only if the serial number equals or exceeds the saved high-water mark. If the new high-water mark exceeds the old, the Enforcer updates the saved one. (Alternatively, the Enforcer could use some type of forward-secure key evolution.)

Structure In order to make our system usable, we chose designs that coincide with familiar programming constructs. These choices may also made our system easier to build—since we could re-use existing code.

For short-lived data, we wanted to give the programmer a way to save and retrieve non-volatile data whose structure can be fairly arbitrarily. In systems, the standard way that programmers expect to do this is via a filesystem. A *loopback filesystem* provides a way for a single file to be mounted and used as a filesystem; an *encrypted loopback filesystem* allows this file to be encrypted when stored [11]. So, a natural choice for short-lived data was to have the Enforcer save and retrieve keys for an encrypted loopback filesystem.

For the medium-lived software, we needed a way for a (remote) human to specify the security-relevant configuration of a system, and a tool that can check whether the system matches that configuration. We chose an approach in the spirit of previous work on kernel integrity (e.g., [4, 128]).

The Security Administrator (again, possibly on a different machine or part of a different organization) prepares a signed security policy of the medium-lived component; the long-

lived component of our system uses this policy to verify the integrity of the medium-lived component.

Another question was how to structure the Enforcer itself. The natural choice was as a *Linux Security Module* (LSM)—besides being the standard framework for security modules in Linux, this choice also gives us the chance to mediate (if the LSM implementation is correct) all security-relevant calls—including every inode lookup and `insmod` call. We envisioned this Enforcer module running in two steps: an initialization component, checking for the signed configuration file and performing other appropriate tasks at start-up, and a run-time component, checking the integrity of the files in the medium-lived configuration.

Implementation Experience The Enforcer is an LSM which operates with a 2.6.x Linux kernel. The Enforcer can either be built as a dynamically loadable module, or it can be compiled directly into the kernel. Further details can be found in earlier technical reports [67, 72].

The Enforcer uses the `/etc/enforcer/` directory to store its signed policy, public key, etc. (Having the kernel store data in the filesystem is a bit uncouth, but was the best solution and is not completely unprecedented.) When the kernel initializes the Enforcer, the Enforcer registers its hooks with the LSM framework. If built as a loadable module, the Enforcer verifies the policy's signature at load-time; if compiled into the kernel, the Enforcer verifies it when the root filesystem is mounted.

At run-time, the Enforcer hooks all inode permission checks (which happen as a file is opened). The Enforcer calculates a SHA-1 of the file and compares it to the SHA-1 listed in the policy; if the values do not match, it reacts according the option: log the event to the system log, fail the call, or panic the system. Tapping each inode read operation would be better from a security standpoint, in that it would check the file's integrity each time the file is read. While this would alleviate any time-of-check-time-of-use issues which arise

```
4d37d651b ... 4a57afd4f2 deny file.txt
```

Figure 4.1: An example line from the Enforcer's security policy.

between opening a file and another party writing to it, it would also be quite expensive and would still not work for things like log files.

Tools The Bear/Enforcer package includes a number of utilities to produce the security policies used by the Enforcer. The security policies are constructed by the Security Administrator, and an example line from a policy is depicted in Figure 4.1. The figure shows the SHA-1 hash of the file `file.txt`. Should the Enforcer detect an integrity violation, it will deny the request. In order to give the Enforcer the ability to perform key operations (such as verifying the policy's signature), we used an open source big integer package [50]. Our utilities also include a key generation and policy signing tool.

Trust Linux with the TPM and our Enforcer LSM enables in practice what prior work only enabled in theory: a way to bind a general-function desktop or server program—including its configuration and operational data—to a long-lived private key.

If someone tampers with a file on the server which is guarded by the Security Administrator's security policy, the program will not be able to prove knowledge of the private key to the relying party Alice. A CA who wants to certify the "correctness" of such a platform essentially certifies that the long-lived core operates correctly, and that the named Security Administrator will have good judgment about future maintenance. (Essentially, this approach generalizes the "epoch" idea of outbound authentication in the IBM 4758 [116].)

In our scheme, the TPM testifies directly, through use of PCRs, to the long-lived components of our server: the hardware and BIOS, the kernel and current Enforcer, and the Security Administrator's current public key. The Security Administrator then testifies to the medium-level software, and the Enforcer (already verified) ensures that the current sys-

	Component	How Protected	Who Vouches
Short-term	Log files Web Pages Prog. keys	File system App policy Enc Loopback	System or Application Admin
Med-term	/bin/apache /bin/openca /bin/ls /sbin/insmod	Enforcer's run-time check	SA via signed policy
Long-term	SA's pubkey Enforcer Kernel HW + BIOS	Policy check TPM Unseal	CA via TPM Attestation

Table 4.1: The Bear/Enforcer components and relationships.

tem matches the Security Administrator's signed policy.

The operational data of the program is controlled by various users, per Bob's policy. These users are authenticated via the kernel and medium-level configuration that has already been testified to. Their content is saved in a protected loopback filesystem, ensuring that it was valid content at some point. Table 4.1 illustrates the components and relationships used in the Bear/Enforcer system.

Figure 4.2 sketches the components and their lifespans in Bear/Enforcer. To enable a client to make a trust decision about dynamic content based on a long-lived application key pair, we introduce indirection between the long-lived components and the more dynamic components. Our intention is, like the IBM 4758, the TPM/Linux platform would let the end user buy the hardware, which could authenticate these components to "Yet Another CA."

Limitations One limitation of the Bear/Enforcer approach is its susceptibility to attacks from the "root" user. Once secrets have been released from the TPM and reside in memory, root can access them by snooping memory. While this problem arises as a result of standard

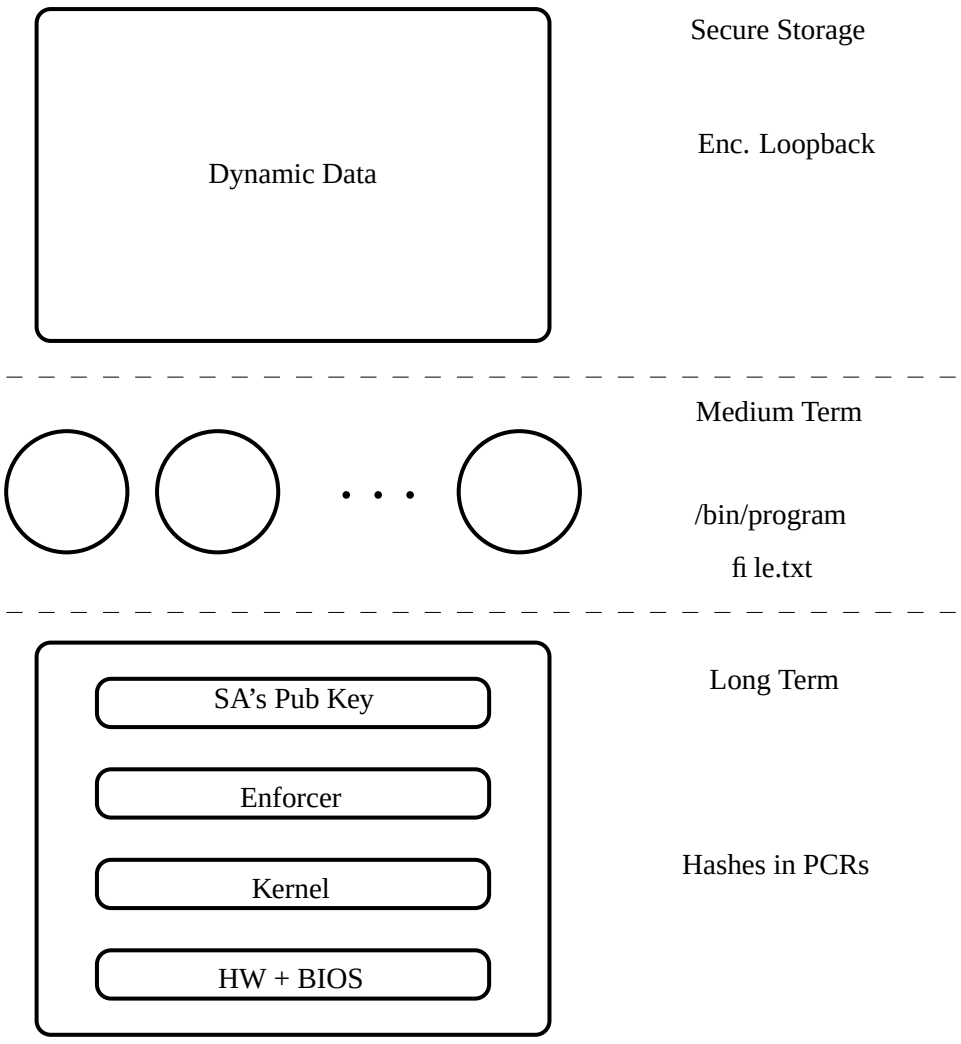

Figure 4.2: Sketch of the fbw of protection and trust in Bear/Enforcer.

Protects Against	Standard Desktop	Bear/Enforcer	IBM 4758
Malware at user priv. (keyjacking)	no	yes	yes
Malware at root priv.	no	no (yes w/ SELinux)	yes
Physical attack	no	no	yes

Table 4.2: Comparison of relevant secure hardware approaches.

Linux design, we can stop this attack by combining Bear/Enforcer with the National Security Agency's *Security Enhanced Linux* (SELinux). In previous work, our lab has examined combining Bear/Enforcer and SELinux to produce an application called "compartmented attestation", which relies on the absence of a root-spy [73].

4.2.3 Summary

The primary use of secure hardware is in the area of trusted computing. Secure hardware provides two features which make it attractive for such applications: a separate security environment from the hardware's host, and the ability to attest about processes running on the hardware. SHEMP benefits from both of these features, and thus attempts to use secure hardware when it is available.

In this section, we covered two platforms: the IBM 4758 and Bear/Enforcer. There are other secure devices and platforms, but these are both unique. The IBM 4758 is the only such platform to have been awarded a FIPS 140-1 Level 4 certification, and Bear/Enforcer is the only freely available open-source platform based on the TCPA/TCG technology. Table 4.2 compares the two platforms. Our SHEMP prototype was developed on the Bear/Enforcer platform due to its larger memory capacity, ease of programmability, and its ability to run Java (which gave us access to class libraries).

4.3 Policy

The last tool we examine is a policy framework. In order to enhance the expressiveness and usability of the SHEMP system, users must be able to relay their wishes regarding key usage to relying parties and applications. 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 which comprise the current TCB).

In one role, the policy framework should allow a relying party Bob, upon receiving a PC for Alice, to be able to discover the conditions under which Alice's PC was generated. Then, Bob can decide for himself whether to trust Alice, given her current environment. As we will explain in detail in Chapter 5, SHEMP administrators assign attributes to clients and repositories. When Alice makes a request for the repository to generate a PC for her, the repository will include the attributes of the client desktop and the repository in the PC itself. These attributes essentially define the Alice's TCB. When Alice presents her PC to Bob, he can examine the attributes himself, and then make a trust decision based on Alice's TCB.

In another role, the policy framework should allow a keyholder Alice to express her wishes about uses of her private key—potentially based on the security level of the repository and client platform. For example, users may wish to restrict access to cryptographic operations that the repository will perform with their private key; applications may wish to restrict certain data or operations. Without this ability, a successful attacker could fully impersonate the victim or use the victim's key for any operation. The policy framework must be flexible enough to allow SHEMP administrators to specify domain-specific attributes to machines, and easy enough to use that users and application developers can construct

policies which accurately govern their resources.

There are a number of policy frameworks which can be used to accomplish these tasks, and the SHEMP system design is agnostic to which framework is used. However, for the sake of implementation, we chose to use the *eXtensible Access Control Markup Language* (XACML) [136]. 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, and makes an access control decision. We chose XACML because, from a practical standpoint, XACML is generic enough to perform express a wide range of attributes, and has an open-source implementation (thanks to Seth Proctor at Sun Microsystems) [100] which is implemented in the language of our prototype: Java. As we will show in Chapter 7, it is possible to build XACML-generating policy tools which make XACML easy enough to use for administrators and application developers.

4.4 Summary

In this chapter, we introduced the building blocks used to build the SHEMP system. In Chapter 5, we discuss the SHEMP architecture, and present the parties, processes, and system components involved in SHEMP. We also describe how we use the building blocks to build our prototype implementation of the SHEMP system. In Chapter 6, we describe how we used SHEMP to build some real-world applications; we also introduce some new opportunities for application development.

Chapter 5

SHEMP Architecture

Armed with the building blocks presented in Chapter 4, we can design and implement the SHEMP system, which meets the criteria of Chapter 3 and makes desktops usable for PKI. Concretely, the goal of the SHEMP system is to allow a relying party Bob to be able to make valid trust judgments about Alice upon receiving a PC from her. Bob should have some reason to believe that Alice authorized the issuance of her PC for the stated purpose, and is aware of—and intended—the request.

Chapter Outline In Section 5.1, we examine the SHEMP architecture and introduce the parties, procedures, and protocols which make up the SHEMP system. In Section 5.2, we discuss our implementation experience. Finally, Section 5.3 summarizes the chapter.

5.1 SHEMP Architecture

5.1.1 SHEMP Entities

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. Figure 5.1 depicts the

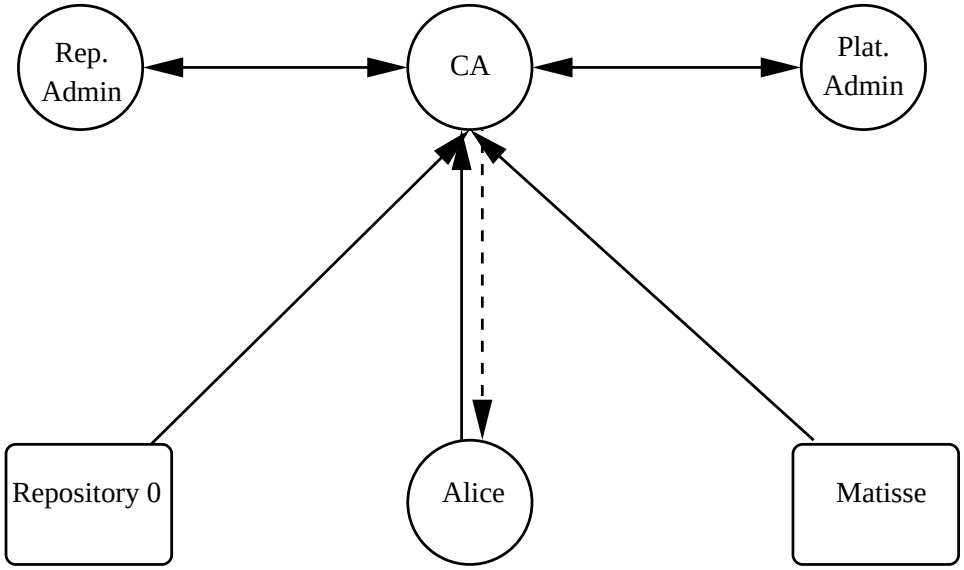

Figure 5.1: The parties in the SHEMP system.

parties and trust relationships 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."

Initially, we focus on three entities involved in SHEMP: a CA, a user (Alice in Figure 5.1), and a user's machine (Matisse in Figure 5.1). 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 5.1.

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 5.1.

For an application running on Alice's machine (Matisse) to trust certificates signed by the CA (such as Alice's), it usually needs to have the CA's certificate installed in its

keystore. This relationship is represented by the edge from Matisse to the CA in Figure 5.1. 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 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 client platforms in the domain (such as Matisse), and at least one key repository (depicted as *Repository 0* in Figure 5.1).

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 5.1.

The Platform Administrator is in charge of the desktop 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 desktops that will be using the keys stored in the repository, the CA must trust the Platform Administrator. Since the Platform Administra-

tor 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 5.1 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 desktop 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 5.1

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 (e.g., a 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 5.1. The set of entities in Figure 5.1 is the smallest set which is necessary and sufficient to describe the system.

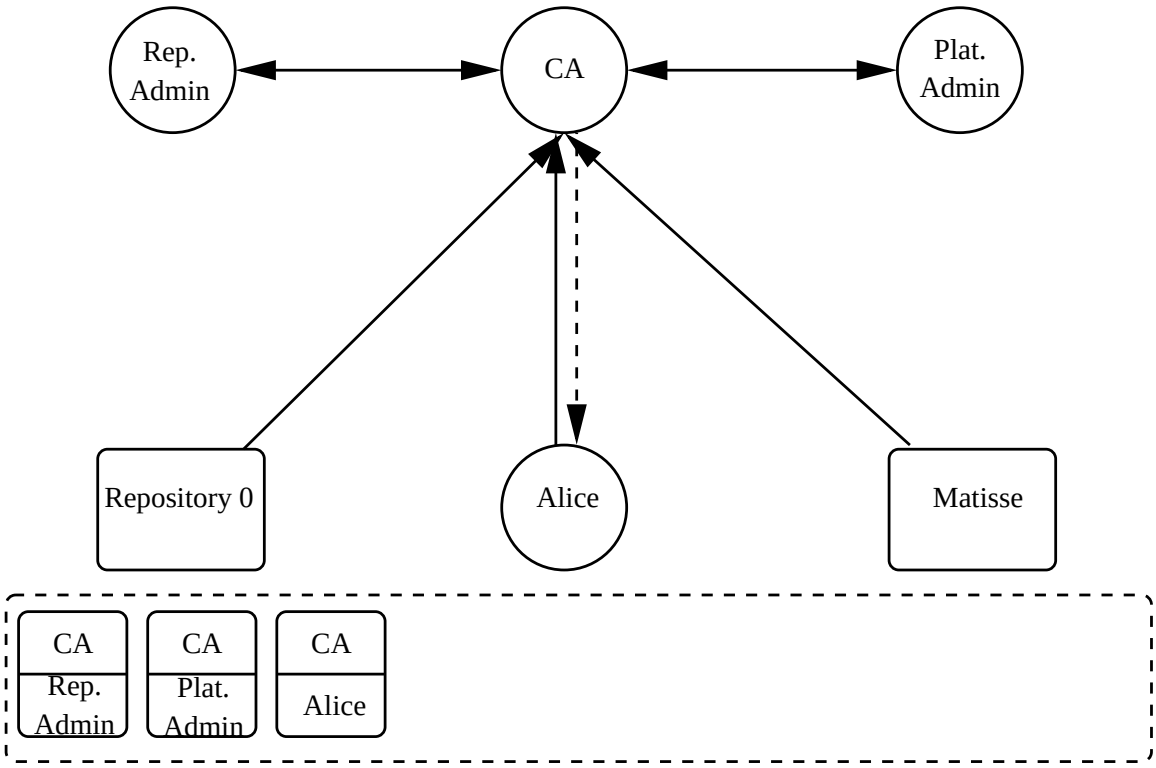

Figure 5.2: The entities, trust relationships, and initial certifi cates in SHEMP.

5.1.2 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 5.1, a number of certificates can be immediately issued. Figure 5.2 illustrates these initial certificates; they are contained in the dashed box which could possibly represent a directory (possibly a *Lightweight Directory Access Protocol* (LDAP) directory) where users go to locate 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

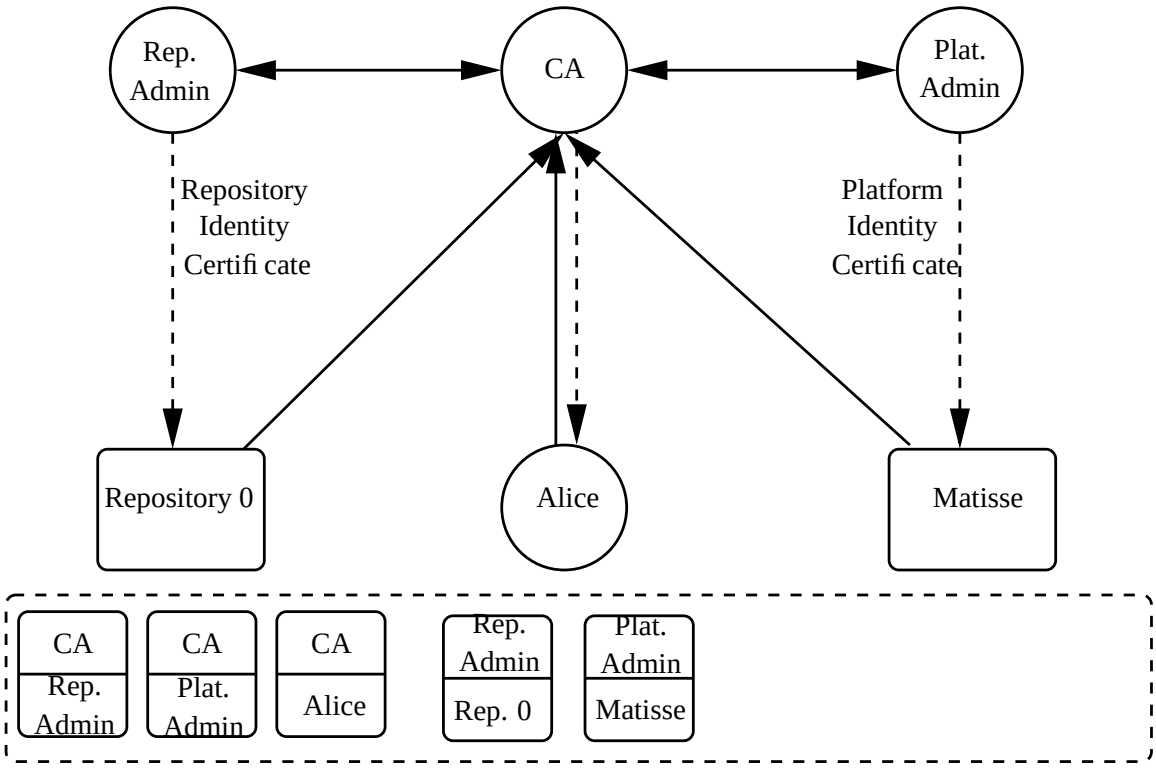

Figure 5.3: The administrators issue identity certifi cates to the repository and Matisse.

themselves.

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 5.2 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 5.2.

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 repository runs on. For instance, if the repository runs in an IBM 4758, then the keypair ought to be generated inside the device,

thus reducing the risk of a key compromise. 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. Figure 5.3 depicts the certificate issued from the Repository Administrator to Repository 0 as a dashed edge. The resulting certificate is added to the certificate store.

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 to it (e.g., an IBM 4758 or TPM, if available).

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 5.3). 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 Public Endorsement Key could be used, providing a more secure identifier than a hardware MAC address (which is easily spoofed).

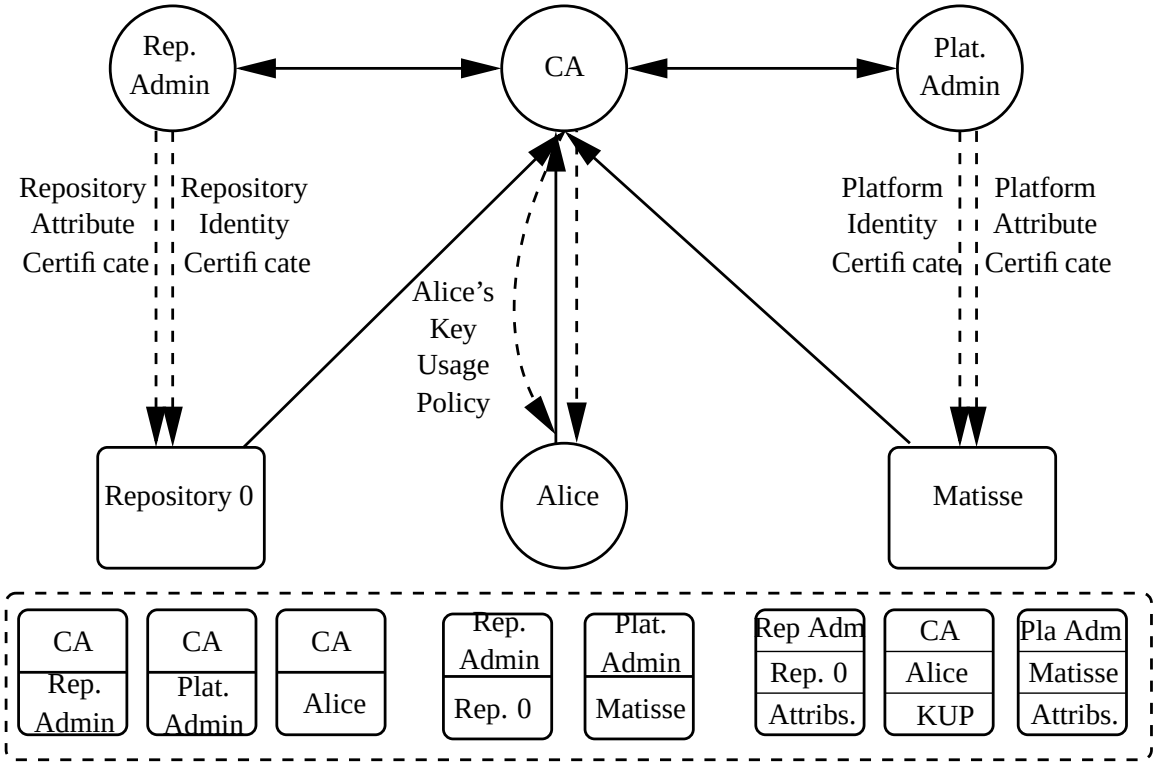

Figure 5.4: The administrators issue attribute certifi cates to the repository and client platform.

5.1.3 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 key usage and delegation policy to the user's identity. In the case where the attributes are assigned to machines (i.e., the repository and client platforms), the attributes can be used by a relying party to reason about the security level of the TCB.

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 signed XML statement called 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 5.4.

The story continues as 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. The attributes are placed into a signed XML statement called the *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 XML attributes which express the security level of the machine. The PAC is signed by the Platform Administrator and is placed in a well-known location such as an LDAP directory. This procedure and the resulting certificate is shown in Figure 5.4.

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 (i.e., the same X.500 Distinguished Name in her X.509 identity certificate) and is placed into the LDAP along with her identity certificate. Alice's private key is then loaded into the repository (actually, it is generated there and the CA receives a *Certificate Request Message Format* (CRMF) request), and setup is complete (see Figure 5.4).

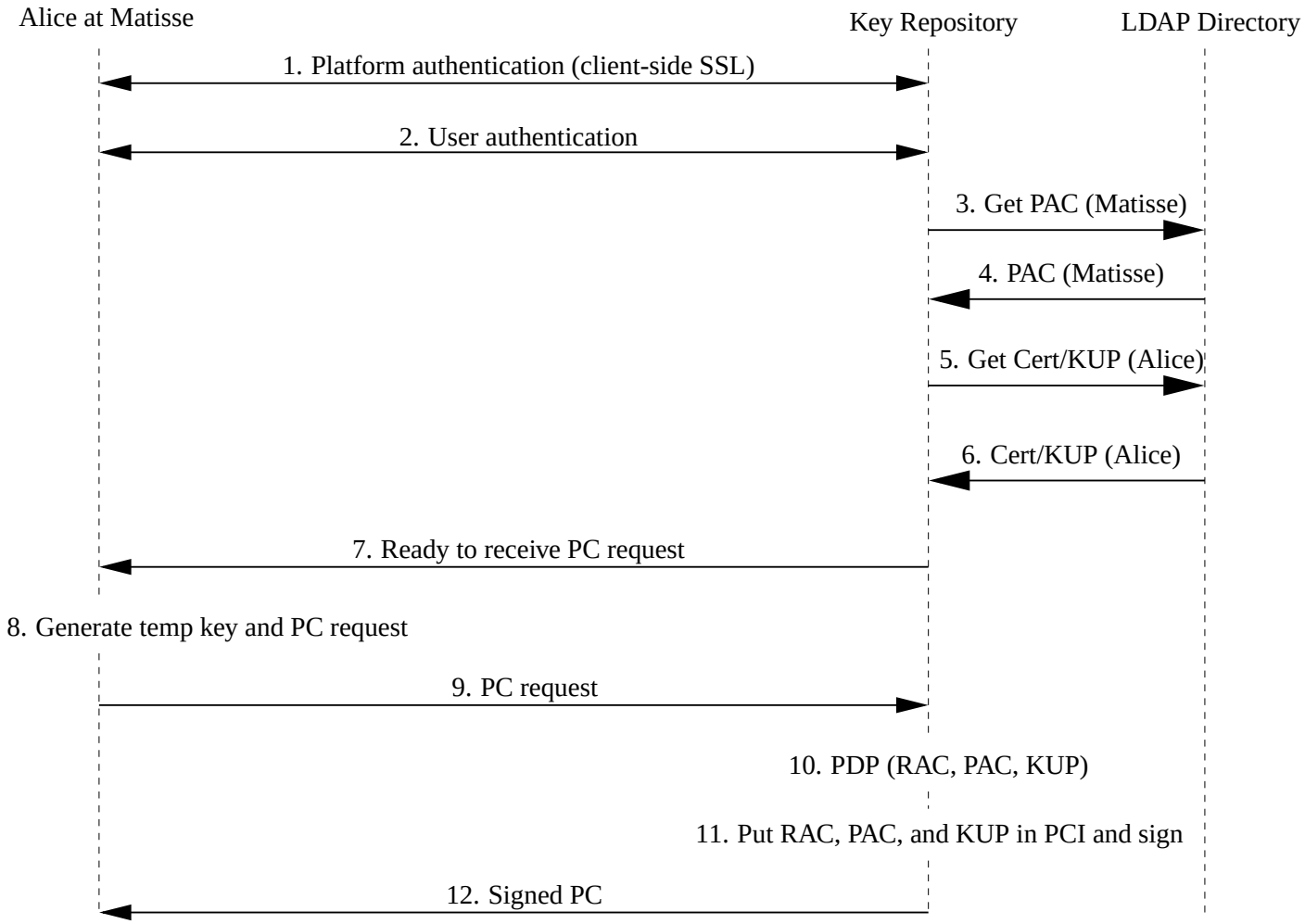

Figure 5.5: The basic protocol for generating a Proxy Certifi cate under the SHEMP system.

5.1.4 The System in Motion

Once setup is completed, Alice is free to wander throughout the domain and use her repository-resident private key from any client desktop. 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 SHEMP client software installed, and hence has a Platform Identity Certificate and PAC in the directory). For illustration, assume Alice walks up to the client named Matisse.

The protocol for establishing a Proxy Certificate is shown in Figure 5.5. Steps 3, 4, 5, 6,

10, and 11 represent our enhancements to the basic MyProxy approach. Step 8 also differs somewhat in that under SHEMP, the temporary keypair can be generated in the most secure manner that a client has at its disposal (e.g., in a TPM or an IBM 4758).

Figure 5.5 illustrates a scenario where the user Alice is at a client desktop named Matisse. Matisse first connects to the repository and establishes a client-side SSL connection. The repository and platform identity certificates (and the corresponding private keys installed by the appropriate administrator) 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 uses a username/password (as does the MyProxy system). For stronger security, Alice could use an 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 5.5, 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 wait 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 [91]. 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. As with standard X.509 certificates, Alice fills in the certificate information and proves possession of the 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 contained in the certificates. This XACML request and Alice's KUP are then evaluated to determine whether the operation is allowed. Concretely, an XACML Policy Decision Point running on the repository (as part of the repository software) makes this decision.

If the PC generation operation is allowed, the repository will place the RAC, PAC, and KUP into the Proxy Certificate's Proxy Certificate Information extension, and then sign the Proxy Certificate with Alice's private key. Placing the RAC, PAC, and KUP into the PCI allows the Proxy Certificate's relying party to see attributes of the client platform and the repository without having to search for them in a directory. The attributes contained in the certificates tell Bob what kind of environment Alice is operating in. Thus, Bob can decide whether he should allow access to resources based on Alice's current environment. 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, forms 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 by the repository.

5.2 Implementation

In Section 5.1, we gave a conceptual view of the SHEMP system. In this section, we present a specification- and implementation-level view of SHEMP. We begin by describing the prototype environment. We then walk through setting up the system from the view of an administrator, discussing what the SHEMP setup tools do and how they are implemented. Finally, we illustrate using the system from the perspective of a user, this time explaining what the SHEMP client software is doing.

5.2.1 Prototype Environment

The SHEMP prototype is comprised of several programs which are used to configure the system, run a SHEMP repository with decryption and signing proxy applications (discussed in Chapter 6), and let clients access the repository. The prototype was developed on the Linux OS, specifically, the Debian distribution on the "unstable" branch. The repository and client portions of the prototype are written in Java and were developed on the *Java 2 Platform, Standard Edition* (J2SE) version 5.0. The *SHEMP Admin* (`shadmin`) tool which is used for setup is written in Perl, and was developed on Perl 5.6.

In addition to borrowing design concepts from MyProxy, we also borrowed some implementation libraries. Our goal was to make the SHEMP programming interface resemble the MyProxy interface as much as possible, as this reduces the learning curve for MyProxy application developers to write applications for SHEMP. To this end, we developed the prototype as part of the Globus Java *Commodity Grid Kit* (CoG kit) [130], which is a standard development kit for the Grid community. This decision gave us access to libraries for performing tasks such as minting Proxy Certificates.

As briefly mentioned in Chapter 4, our policy framework is implemented in the XACML policy language. The prototype utilizes the SunXACML libraries which are freely available from Sun Microsystems [100], and are written in Java.

Once early versions of the SHEMP repository and client software were stable, we constructed a small testbed consisting of four machines which were used for development and testing. Two of the machines run Bear/Enforcer—one hosting the repository and one hosting client software. The other two machines are standard desktops, one with a firewall installed, and one without. Our idea was that this heterogeneous collection of machines likely represents real environments where SHEMP may be used.

5.2.2 SHEMP Setup

In order to demonstrate setting up the SHEMP system, we walk through the three tasks performed by the `shadmin` tool: installing a repository, installing a client, and adding a user to the repository. We begin with the assumption that the CA has issued CA-certificates (i.e., the `basicConstraints` X.509 extension is set to true) to the Platform and Repository Administrators.

Repository Installation To begin, the Repository Administrator needs to install the repository software, install a keypair for the repository, and assign it attributes. In our testbed, the Repository Administrator installs the repository software on a Bear/Enforcer machine by unpacking the repository code in the encrypted loopback filesystem.[1] If we were using an IBM 4758, the Repository Administrator could install the code inside of the device.

Once the software is installed and the repository is running, the Repository Adminis-

[1]The Repository Administrator, possibly acting as the machine's Security Administrator should also set up a security policy which protects the repository code and the private keys.

trator will launch the `shadmin` tool, and select the set of operations for Repository Administrators. This set of operations will allow the Repository Administrator to import the CA's certificate into the repository's Java keystore, thus allowing the repository to accept client-side SSL connections from platforms which have a certificate chain rooted at the same CA. The `shadmin` tool calls the `keytool` program (part of the J2SE) to import the CA's certificate into the repository's keystore.

The Repository Administrator can then generate a new keypair and certificate which will uniquely identify the repository (again, the `shadmin` tool relies on the `keytool` program to perform these tasks). The public portion of this keypair is put into a certificate request, and signed by the Repository Administrator. The `shadmin` tool calls the `openssl ca` program to sign the repository's certificate with the Repository Administrator's private key. The certificate's *Distinguished Name* (DN) should be the least spoofable identifier available; for our prototype, it is the TPM's Public Endorsement Key. Once the certificate is signed, the Repository Administrator imports it into the repository's keystore. The keypair is then used to negotiate SSL connections with clients, and the certificate is the Repository Identity Certificate of Figure 5.4.

Once the repository is identified, the Repository Administrator must assign it a set of attributes which will be included in the Repository Attribute Certificate (RAC). In addition to these attributes, the RAC contains the same DN for the repository as the identity certificate, as well as the issuer's DN (i.e., the DN of the Repository Administrator). The `shadmin` tool constructs an XML document like the one depicted in Figure 5.6, and then calls the `openssl dgst` program to hash the envelope and sign the hash with the Repository Administrator's private key. Finally, the `shadmin` tool places the signature in the `envelopeSignature` element of the RAC.

```
<attrcert>
   <envelope>
      <subjectDN>CN=77:6e:84 . . . ,L=L,ST=ST,C=C</subjectDN>
      <issuerDN>CN=RepAdmin1, . . . ,ST=PA,C=PA</issuerDN>
      <attributes>
          <attribute>HasTPM=true</attribute>
          <attribute>BehindFirewall=true</attribute>
      </attributes>
   </envelope>

   <envelopeSignature>2b586b . . . 1402e086</envelopeSignature>
</attrcert>
```

Figure 5.6: An example RAC.

Client Installation The installation procedure for clients is fairly similar to the procedure for a repository. The Platform Administrator needs to install the SHEMP client software, generate a keypair and certificate for the platform, and assign it attributes. The location of the client install varies, depending on the target platform. In our testbed, machines running Bear/Enforcer run the SHEMP client software from the loopback filesystem. Machines without secure hardware just ran the client code as a normal program. Had we used an IBM 4758, we could have placed the client code directly inside of the device.

The Platform Administrator's first step is to install the CA's certificate into the platform's Java keystore. The `shadmin` tool calls Sun's `keytool` to accomplish this task. Using the same procedure as the Repository Administrator, the Platform Administrator must next generate a new keypair and Platform Identity Certificate for the new platform, as well as the Platform Attribute Certificate (PAC) which contains the platform's relevant security properties. As before, the `shadmin` tool constructs an XML document similar to the one depicted in Figure 5.6 (the names, attribute, and signature will be different, but the structure is identical). The `shadmin` tool then calls the `openssl dgst` program to hash the envelope and sign the hash with the Platform Administrator's private key, and then places the signature in the `envelopeSignature` element of the PAC.

Adding a User Once at least one repository and platform have been installed, users can begin to use the system. In order for Alice to use the system, she needs to have a long-term keypair generated on the repository, have the public portion signed by a CA, and establish her Key Usage Policy which dictates how and where her key may be used.

Alice begins the process by visiting the Repository Administrator and requesting a that a new long-term keypair be generated for her. The Repository Administrator (possibly working in conjunction with the CA and Registration Authority) will perform the standard vetting process, ensuring that Alice is who she claims to be.

Once convinced, the Repository Administrator will launch the `shadmin` tool, and opt to add a user to the repository. The `shadmin` tool will first generate a keypair and unsigned certificate (containing the public portion of the newly generated keypair) for Alice using the `keytool`. The private portion of the keypair never leaves the SHEMP repository, and the unsigned portion (in the certificate) is then sent to the CA.

Upon receiving Alice's unsigned certificate, the CA operator signs it with the CA's private key, producing a standard X.509 identity certificate for Alice. The `shadmin` tool can be used by the CA operator for this task. It first ensures that the `basicConstraints` extension is set to "true" in Alice's certificate (thus allowing her to sign Proxy Certificates), and then calls the `openssl ca` program to sign the certificate with the CA's private key.

Once Alice's certificate is signed, the CA uses the `shadmin` tool to generate a KUP for Alice. In our prototype, the KUP protects Alice's private key, and restricts how Alice can generate Proxy Certificates and use her private key for decryption and signing. The idea is for Alice and the CA operator to construct a policy which allows Alice to use her private key for different operations (PC generation, decryption, and signing), based on Alice's environment at the time of the request. The `shadmin` tool generates the XACML policy (i.e., the KUP) which represents Alice's wishes, and then signs it with the CA's private key.

Once Alice returns from the CA, the Repository Administrator (using the `shadmin`

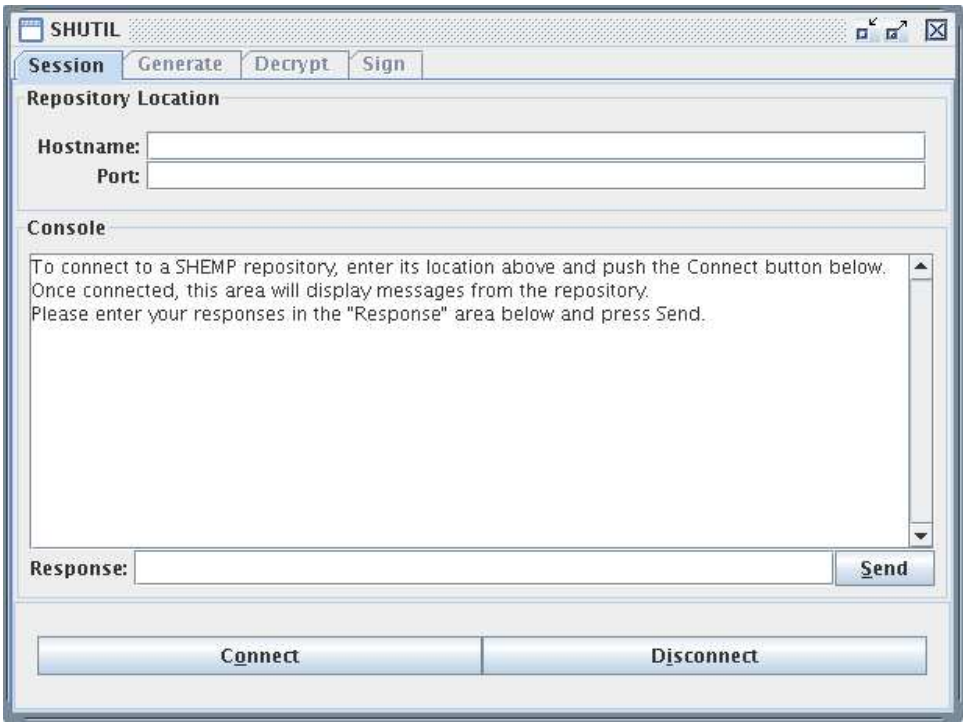

Figure 5.7: The SHUTIL login screen.

tool) installs the signed certificate in the repository keystore (via `keytool`), publishes it in the organization's certificate store (most likely, a directory), and sets up an account and password for Alice on the repository.[2] Once Alice has successfully been added to the repository, the situation looks like Figure 5.4.

5.2.3 Using SHEMP

Once the repository and client(s) are installed, and Alice has a keypair and certificate, Alice can begin to use the system. She begins by sitting at a client platform (for the remainder of this section, we will call this platform "Matisse"). Alice launches the *SHEMP Utility* (`shutil`), which presents her with the user interface depicted in Figure 5.7.

[2]Since our prototype uses passwords for user authentication, the `shadmin` tool sets Alice's password. If we had used biometrics or some other authentication technique, then the `shadmin` tool would need to set up that mechanism instead of a password.

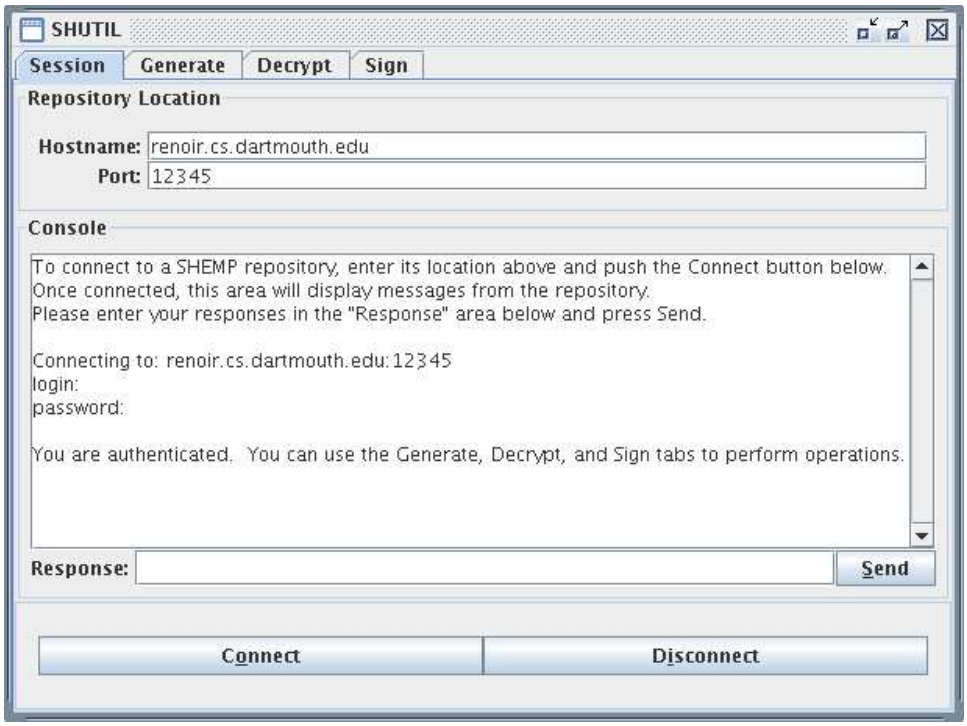

Figure 5.8: Alice logs on to the repository.

Alice enters the hostname and port where the repository is running, and presses the button labeled "Connect." The `shutil` will then attempt to make a client-side SSL connection to the repository, thus performing platform authentication (step 1 in Figure 5.5).

If a connection is established, the `shutil` initiates the user authentication step (step 2 in Figure 5.5) by repeatedly asking the repository for challenge strings to present to Alice. The `shutil` code simply displays these strings on Alice's console, as shown in Figure 5.8. Alice responds in the text box labeled "Response", and `shutil` forwards Alice's response back to the repository. In our prototype, the challenge strings will ask Alice for her username and password, and check these against a password file on the repository. If Alice provides a correct username and password, the repository sends a message to `shutil` which indicates that Alice is authenticated. This message causes `shutil` to give access to its operations (represented as the tabs in the figures): generate, decrypt, and sign.

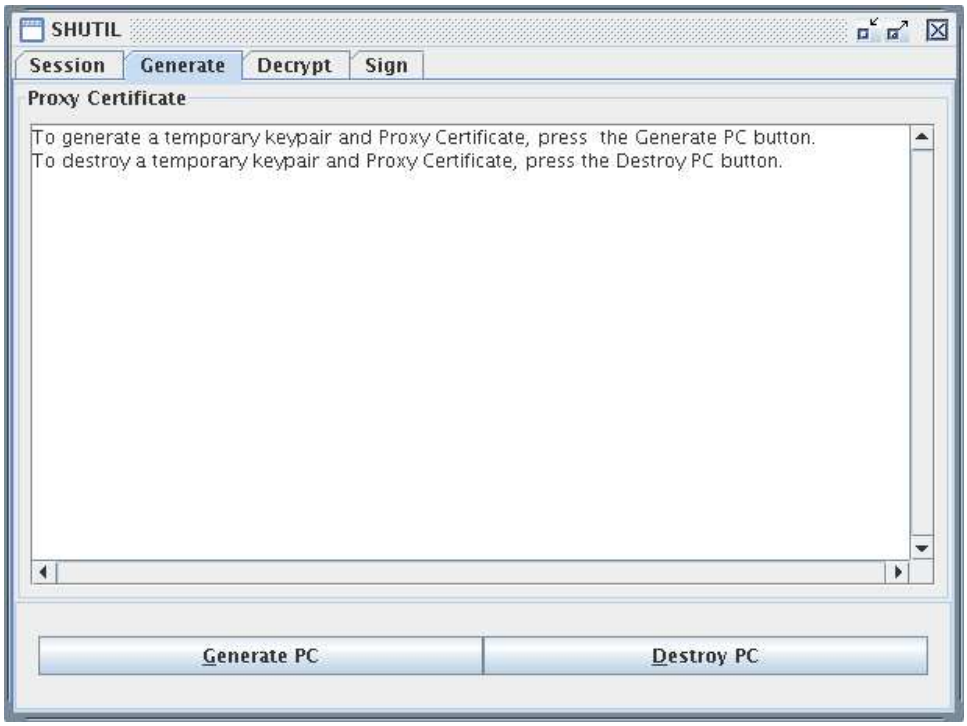

Figure 5.9: Alice requests a PC.

Typically, Alice will begin by selecting the "generate" option to generate a new Proxy Certificate. Alice simply presses the "Generate PC" button (see Figure 5.9) which causes the `shutil` to send a message to the repository. Upon receipt, the repository will gather all of the relevant attribute certificates (i.e., the RAC, PAC, and Alice's KUP), verify their signatures, and construct an XACML request with the attributes. If the signatures verify, then the repository sends a "ready" message to the `shutil`, indicating it is ready to receive the unsigned Proxy Certificate (steps 3–7 in Figure 5.5).

The `shutil` then generates a keypair using whatever means available (the Platform Administrator can specify the generation mechanism during system setup), and the public portion is placed in an unsigned Proxy Certificate and sent to the repository for signing (steps 8–9 in Figure 5.5).

The repository then runs the request (containing the attributes from the RAC and PAC)

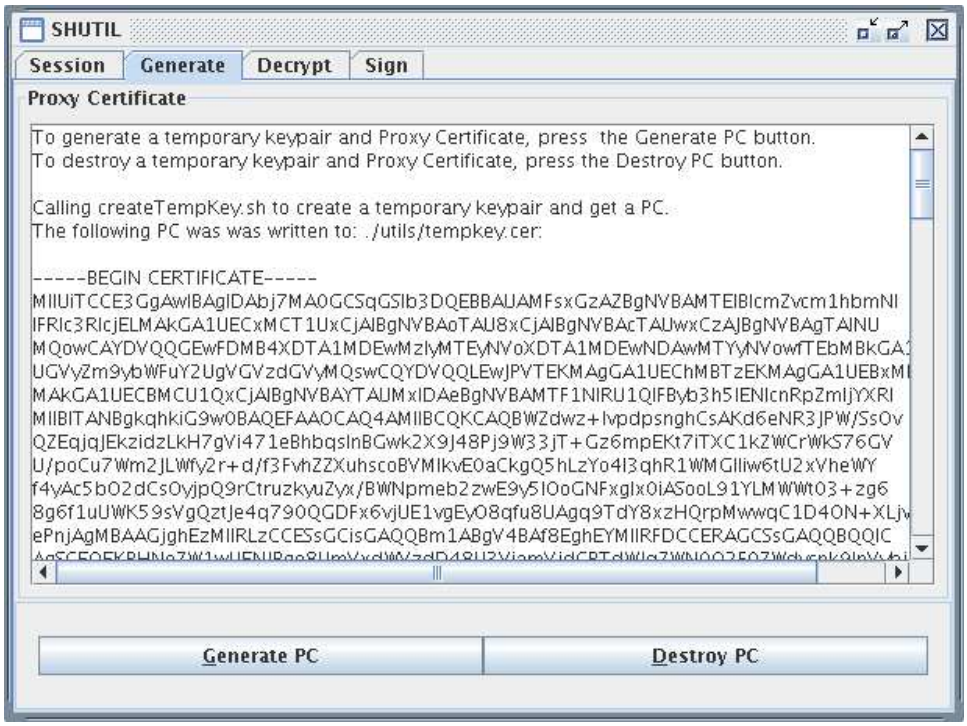

Figure 5.10: Alice successfully generates a PC.

along with Alice's KUP through the XACML Policy Decision Point to determine whether the current environment allows Alice to generate a Proxy Certificate. If so, the RAC, PAC, and KUP are placed in the Proxy Certificate's PCI extension, the Proxy Certificate is signed by the repository using Alice's private key, and the signed certificate is returned to Alice (steps 10–12 in Figure 5.5). The result is shown in Figure 5.10.

Alice can now use her Proxy Certificate for the next two hours (`shutil`'s and MyProxy's default). As we will discuss further in Chapter 6, the SHEMP repository offers two services which use Proxy Certificates for decryption and signing. Before SHEMP, Proxy Certificates have been limited to authentication and authorization applications. When Alice is done, she may log out and destroy her Proxy Certificate and the corresponding private key by pressing the "Destroy PC" button (see Figure 5.9).

Remote Use So far, we have discussed how Alice uses the SHEMP system from within her domain (i.e., Matisse is physically located in Alice's domain). In the case where Alice travels outside of her domain, Alice can no longer rely on her client machine to be under the Platform Administrator's jurisdiction.

In such scenarios, Alice can define a very restrictive KUP which covers untrusted external machines; she would be wise not to perform sensitive operations from such platforms. Alternatively, she could potentially used a token-based keypair to establish a Virtual Private Network to some machine inside the domain. While this is far from a perfect solution, it could give Alice some way to access a trusted machine remotely.

5.3 Summary

In this chapter, we introduced the parties, procedures, and protocols used in the SHEMP system. We illustrated the trust relationships between SHEMP entities, and described the mechanisms we used to express those relationships. We then described how we used the building blocks of Chapter 4 to implement our design. In Chapter 6, we describe how we use the SHEMP system to build applications.

Chapter 6

SHEMP Applications

As discussed throughout this thesis, SHEMP's goal is to make desktops usable for PKI. One way to measure SHEMP's success in this area is to try and build applications. In this chapter, we describe the implementation of two applications which were built as a part of the SHEMP repository: a decryption and signing proxy. While other approaches use third parties to aid in private key operations (e.g., [3, 10, 17]), our applications are novel contributions in themselves, as they explore the use of Proxy Certificates for standard private key operations.

We then present three designs for some standard Grid applications, and show SHEMP can enhance these designs. We also discuss the results of brainstorming sessions where we discuss applications that could possibly benefit from SHEMP. In Chapter 7, we elaborate on these designs and give the results of a user study we conducted which uses the applications to measure the usability of SHEMP's policy language.

6.1 Decryption and Signing Proxies

Traditional PKI uses of private keys include decryption, signing, and authentication. The Proxy Certificates (PCs) generated by SHEMP can by used for any of these operations, although the short lifespan of the Proxy Certificate adds some complexity. For example, if Bob encrypts something for Alice using her PC's public key, and the PC expires before Alice decrypts the message, then she loses the ability to decrypt the message. If Alice signs something with her temporary private key, and Bob attempts to verify the message after Alice's PC has expired, the signature is unverifiable.

Having Bob deal with Alice's long-term certificate would be ideal, but then Alice needs a way to ask the repository to perform private key operations on her behalf. The decryption and signing proxies are designed to solve this problem. They allow Alice to turn a message encrypted with her long-term public key into a message encrypted with her temporary public key, and turn a signature generated with her temporary private key into a signature generated with her long-term one.

Our design and implementation goal is to allow Bob to send encrypted messages to—and verify signatures from—Alice, using her long-term credential. Bob should not have to know anything about SHEMP or Proxy Certificates and should still able to securely communicate with Alice. Relaxing this constraint yields some interesting potential applications (discussed at the end of this section), but significantly increases the deployment overhead which often results in a system that may never make it beyond the prototype stage. Without the constraint, every Bob—*and every PKI application that Bob uses*—has to be rewritten or extended to deal with a specialized SHEMP message format if they want to communicate with a SHEMP user (e.g., Alice). The SHEMP design protects Alice from complexity via the Repository and Platform Administrators. Insulating Bob from the complexity introduced by PCs is consistent with the overall SHEMP design philosophy.

Before we discuss the proxy applications in detail, it should be noted that PKI authentication can be accomplished with the SHEMP system as is. The short lifespan has no real effect on authentication applications, as Proxy Certificates were developed with authentication scenarios in mind (and dynamic delegation, which will be discussed below).

Decryption We first consider the case where Bob wants to encrypt a message and send it to a SHEMP user Alice. We assume that Alice has an X.509 identity certificate issued by her local CA, and that the private key corresponding to the public key in her certificate is stored in a SHEMP repository. Furthermore, we assume that Alice's certificate is in a place where Bob can find it, such as a public directory.

Bob begins by locating Alice's certificate, most likely by performing a directory search. He then uses the public key in the certificate (i.e., Alice's long-term public key) to encrypt a message for Alice. Concretely, these tasks may all be performed by Bob's email client, a program like `openssl`, etc. Once encrypted, the ciphertext is transported to Alice.

Upon receipt, Alice needs to use her long-term private key (which lives on the repository) to decrypt the message and recover the plaintext. If Alice does not currently have a valid Proxy Certificate, then she completes the process described in Chapter 5 to obtain one.

Once Alice has a valid PC and has successfully logged on to the repository, she can access the repository's decryption proxy service via the `shutil` program. The `shutil` code will prompt Alice for a file containing the encrypted message from Bob, and for a file where it should place the result (see Figure 6.1). Once she has entered the filenames, Alice sends the message and her PC to the repository by pressing the "Decrypt" button.

Once the repository receives the message and Alice's PC, it needs to decide whether it should perform the requested operation. To make this decision, the repository first locates the RAC for itself, the PAC for the platform which Alice is making the request from (which

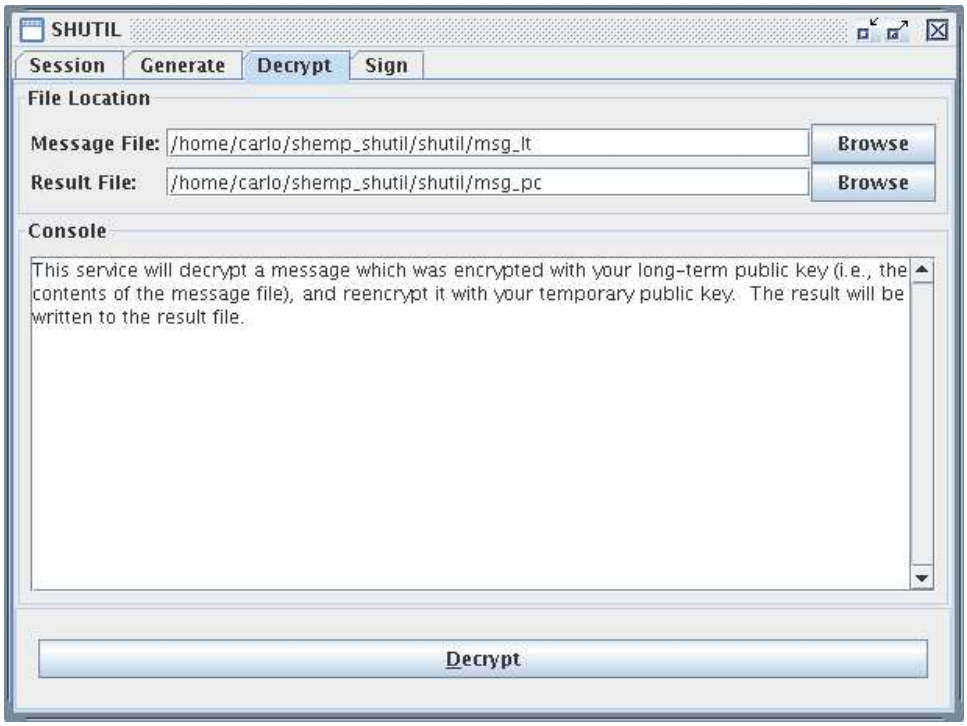

Figure 6.1: Alice sends a request to the decryption proxy.

is uniquely identified by the SSL session), and Alice's KUP. The repository verifies all of the signatures, constructs an XACML request with the attributes found in the RAC and PAC, and then passes the request and Alice's KUP to the XACML Policy Decision Point (PDP). The PDP evaluates the "decrypt" operation in Alice's KUP, and determines whether the decryption proxy should use Alice's private key in the current environment.

Assuming that Alice's KUP allows the operation, then the message is decrypted with Alice's long-term private key, and re-encrypted with the public key found in Alice's current PC. The message is then returned to Alice, where it is displayed in the `shutil` console and written to the result file that Alice specified (see Figure 6.2).

At this point, Alice can use her temporary private key and a decryption tool (such as the one included in the SHEMP package, `openssl`, etc.) to decrypt the message and recover the plaintext. This application allows Alice to use her private key for decryption from

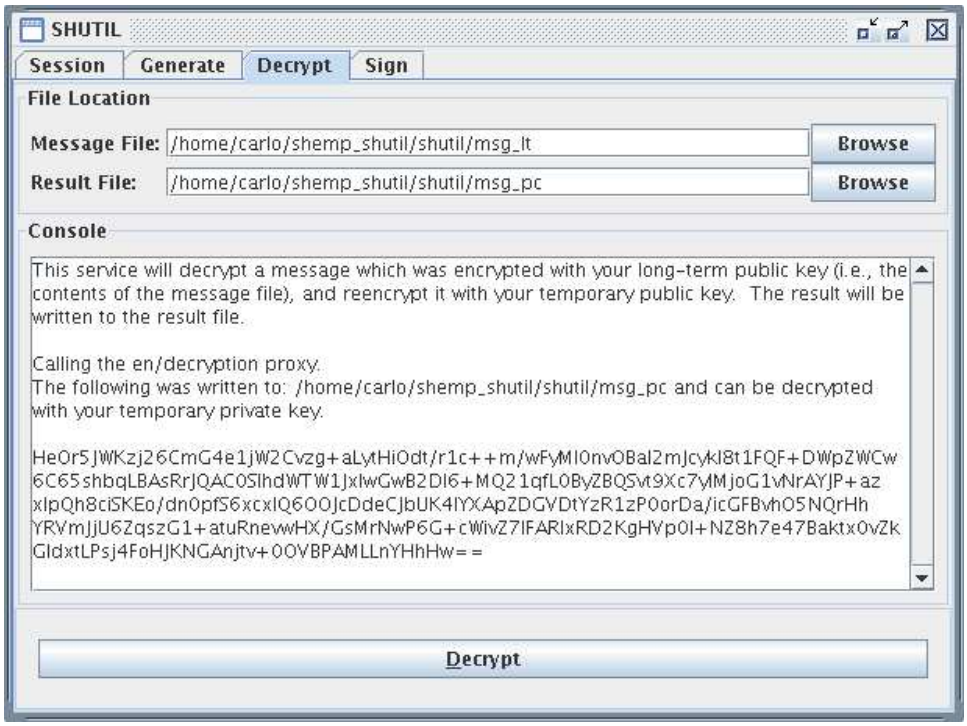

Figure 6.2: Alice successfully decrypts a message.

anywhere in the domain (provided her KUP allows it) without having to transport it. It also guarantees that the plaintext is only exposed on the repository (possibly in secure hardware) if Alice's environment and KUP prevent her from decrypting the message. Finally, this approach hides the entire SHEMP system from Bob, allowing him to send messages to Alice just as he would prior to SHEMP being installed in Alice's domain. The performance hit for the extra decryption and policy check will be examined in Chapter 7.

Signing We now consider the case where Alice wants to sign a message and give it to Bob. We begin with the same set of assumptions about Alice: she has an X.509 identity certificate issued by her local CA, the private key corresponding to the public key in her certificate is stored in a SHEMP repository, and her certificate is in a place where Bob can find it, such as a public directory.

If Alice does not already have a PC, she logs on to her repository and generates one

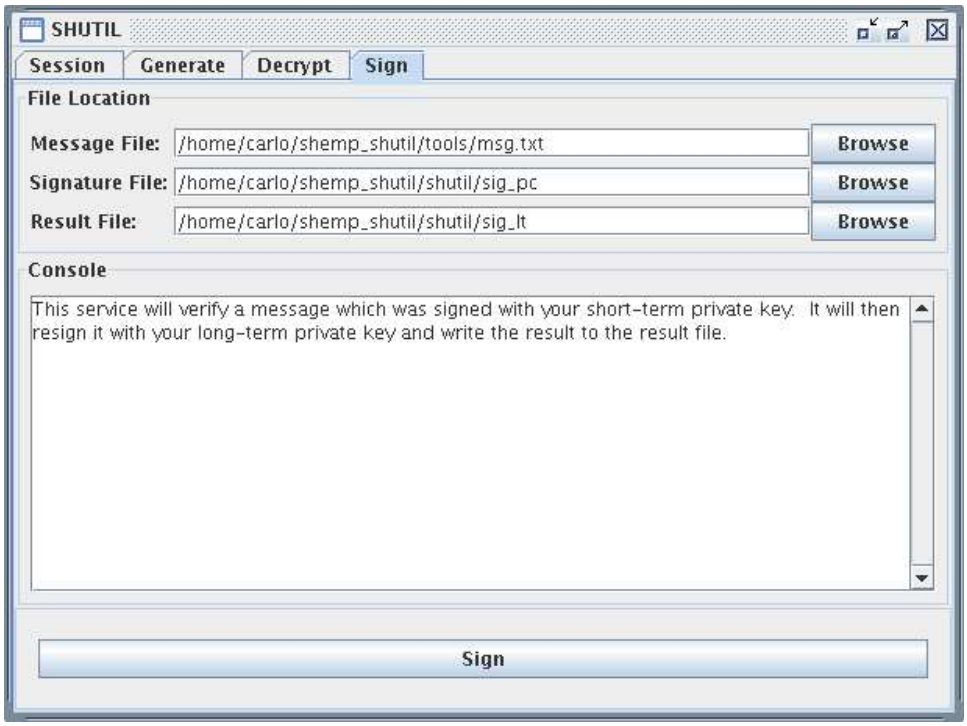

Figure 6.3: Alice sends a request to the signing proxy.

using the process described in Chapter 5. Alice then uses her temporary private key and a signing tool (again, such as the one included in the SHEMP package, `openssl`, etc.) to sign the message.[1]

Since Bob will only be able to verify the signature for the lifespan of the Proxy Certificate (two hours by default), Alice would like to sign the message with her long-term private key. She begins this process by logging into the repository and contacting the signing proxy service. The `shutil` code will prompt Alice for the file containing the message, the file containing the short-term signature (i.e., the signature calculated with Alice's temporary private key), and a file to where `shutil` will place the result. Alice then presses the "Sign" button to send the message, the short-term signature, and Alice's current PC to the repository. The situation is depicted in Figure 6.3.

Upon receipt, the repository first verifies that the message was actually signed with

[1] Actually, Alice typically calculates the message digest and signs that.

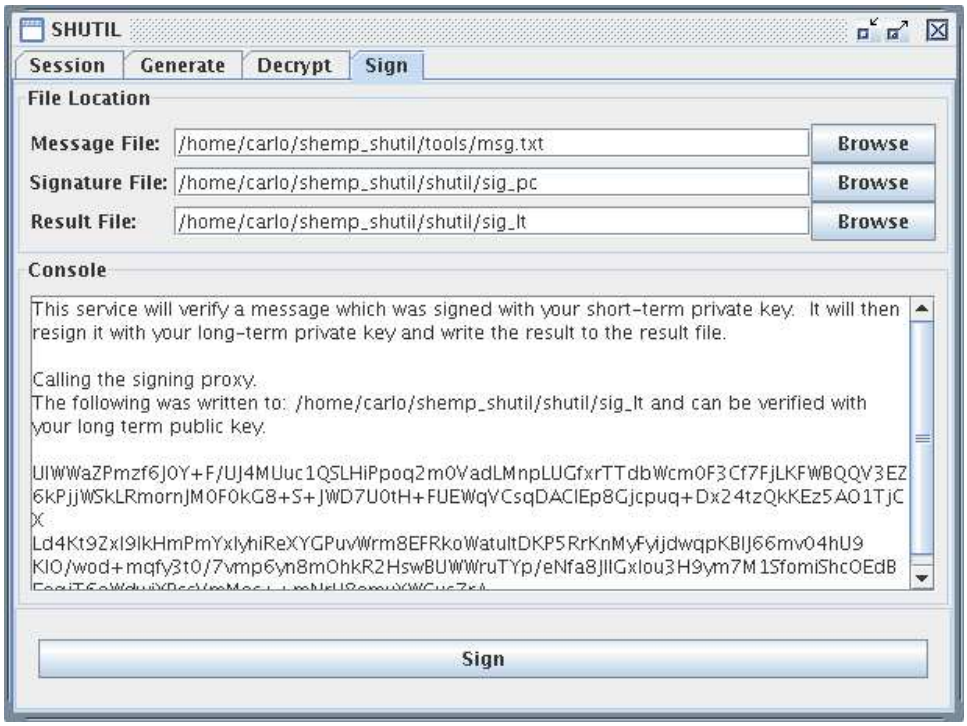

Figure 6.4: Alice successfully signs a message.

Alice's short-term private key. If so, then the signing proxy must check to see if Alice's KUP allows her to perform the "sign" operation under the current environment. As before, the repository locates the RAC for itself, the PAC for the platform which Alice is making the request from (again, identified by the SSL session), and Alice's KUP. The repository verifies all of the signatures, constructs an XACML request with the attributes found in the RAC and PAC, and then passes the request and Alice's KUP to the XACML PDP. The PDP evaluates the "sign" operation in Alice's KUP, and determines whether the signing proxy should perform the operation.

Assuming that Alice's KUP allows the operation, the message is re-signed with Alice's long-term private key, returned to Alice, displayed in the `shutil` console, and written to the result file that Alice specified (see Figure 6.4).[2]

[2]In theory, the repository could also timestamp the signature at this point. We will discuss this issue in detail in Chapter 8.

Assuming everything was successful, Alice can now give Bob the signature which was generated by the signing proxy using her long-term private key. When Bob wants to verify the signature, he uses Alice's long-term public key, just as he would. Bob can now verify the signature even after Alice's PC has expired, and he is not required to know anything about SHEMP whatsoever. Alice can now generate signatures from anywhere in the domain (provided that her KUP allows it), and she does not have to worry about transporting her private key. The performance impacts of this approach will be considered in Chapter 7.

6.1.1 Alternate Designs

We considered two alternative designs which we thought would be useful, but they force Bob to become "SHEMP-aware." Concretely, the implication is that Bob would have to rewrite or extend his applications in order to communicate with *any* SHEMP-user (e.g., Alice in our previous examples). This level of cost to the relying parties would stifle the adoption of SHEMP, as organizations would have to rewrite applications to either parse Proxy Certificates or adhere to a SHEMP-specific message format. Nevertheless, these are potentially useful applications and are worth mentioning, even though we did not implement them.

Closeout Certificates The first alternate design we present involves changing the signing proxy. As before, we begin with the assumption that Alice has (or gets) a Proxy Certificate, and that she wants to sign a message for Bob.

Alice begins by signing the message with her temporary private key, constructing a request for the signing proxy, and sending it. As before, the signing proxy service will perform the policy check by gathering the necessary certificates, verifying their signatures, and asking the PDP if the signing operation is allowed under the current environment.

However, the signing proxy does not actually sign the message with Alice's long-term

private key. Instead, it calculates a hash of the short-term signature, timestamps the hash, and Alice simply gives Bob the short-term signature. When Alice's PC expires or is destroyed, the signing proxy issues a *closeout certificate*, which contains the same public key as Alice's PC as well as the all of the hashes and their timestamps that were generated with the corresponding private key.[3]

Now, when Bob wants to verify Alice's signature, either Alice currently has the PC which contains the public key corresponding to the private key she used to generate the message, or the signing proxy has issued a closeout certificate for that public key which contains all of the timestamped hashes Alice generated with the corresponding private key. In the former case, Bob can trivially verify the message. In the latter case, Bob finds the closeout certificate which contains the digest of the message he is trying to verify, and uses the public key contained in the closeout certificate to verify the message.

The implication of implementing this approach is that Bob now has to do a more complicated search to find the certificate which holds the public key to verify Alice's signature. The timestamps are a help, but require some level of clock synchronization, which is often difficult in practice. Furthermore, any application which Bob uses to verify signatures has to be extended or rewritten to perform the new algorithm.

User-specified Requirements Another alternate approach involves changing the SHEMP message structure altogether so that parties can add requirements which specify how the message should be viewed.

For one example, Bob may want to encrypt a message for Alice, and specify that parts of the message should only be viewed under certain conditions. Concretely, this can be accomplished by allowing Bob to add restrictions to Alice's KUP in real time. This way, the policy check on the repository takes Bob's wishes into account as well as Alice's.

[3]These hashes are likely contained in a Merkle tree or some similar data structure.

```
<message>
   <!-- Ciphertext to be decrypted when Alice has a TPM -->
   <messagePart>
      <data>UlWWaZPmzf6J0Y+F/. . .jcpuq+Dx24tzQkKEz5A<data>
      <attributes>
          <attribute>HasTPM=true</attribute>
      </attributes>
   </messagePart>

   <!-- To be decrypted when Alice is behind a firewall -->
   <messagePart>
      <data>Ld4Kt9ZxI9IkHmPmY. . .xIyhiReXYGPuvWrm8EFR<data>
      <attributes>
          <attribute>BehindFirewall=true</attribute>
      </attributes>
   </messagePart>
</message>
```

Figure 6.5: An example encryption message format.

Another example could involve Alice including the attributes from her RAC and PAC in her signature. This allows Bob to examine the conditions under which the signature was generated, and assuming he knows what Alice's security attributes mean, he can decide for himself whether he believes the signature.

Figure 6.5 shows a SHEMP encryption message format in XML. The format allows Bob to specify what attributes should be added to Alice's KUP in order for her to decrypt certain parts of his message. The message in Figure 6.5 specifies that Alice's decryption proxy should only decrypt the top portion of the message only if Alice is using a platform equipped with a TPM (e.g., Bear/Enforcer), and the bottom portion only if Alice is behind a firewall.

Figure 6.6 depicts a SHEMP signing message format in XML. The attributes included in the message are the attributes included in the RAC and PAC. They tell Bob about the environment under which the signature was generated. In the example of Figure 6.6, both platforms are equipped with TPMs and are behind the firewall.

```
<message>
   <!-- The signature -->
   <data>KlO/wod+mqfy3t0/7. . .vmp6yn8mOhkR2HswBUWW<data>

   <!-- Repository attributes -->
   <attributes>
      <subjectDN>CN=77:6e:. . . ,L=L,ST=ST,C=C</subjectDN>
      <issuerDN>CN=RepAdmin1, . . . ,ST=PA,C=PA</issuerDN>
      <attribute>HasTPM=true</attribute>
      <attribute>BehindFirewall=true</attribute>
   </attributes>

   <!-- Platform attributes -->
   <attributes>
      <subjectDN>CN=90:ed:. . . ,L=L,ST=ST,C=C</subjectDN>
      <issuerDN>CN=PlatAdmin1, . . . ,ST=PA,C=PA</issuerDN>
      <attribute>HasTPM=true</attribute>
      <attribute>BehindFirewall=true</attribute>
   </attributes>
</message>
```

Figure 6.6: An example signing message format.

Clearly, the design goal of keeping Bob unaware of SHEMP breaks down in this case. In fact, not only must Bob be aware of SHEMP, he must be aware of the specific instantiation of SHEMP which Alice is using—i.e., he must know Alice's domain's policy. As before, Bob's applications must be updated to understand the new message format, as well as configured to understand the security policies of Alice's domain (and any other domains Bob deals with). Some of the implementation burden can be eased by standardizing the message format (possibly via defining a SHEMP MIME type), but significantly raises the barrier for SHEMP adoption. Thus, we did not implement it.

6.2 Grid Application Designs

In addition to the applications we implemented (the encryption and signing proxies), we re-designed three applications which are currently used in the Grid community to take advantage of SHEMP. In this section, we give an overview to illustrate uses of SHEMP beyond the PKI primitives of authentication, decryption, and signing. We will revisit these applications in Chapter 7, as they form the basis of a user study we conducted to measure the usability of the SHEMP policy framework.

The three applications all rely on PCs for authentication and authorization. GSSKLOG uses PCs to obtain tokens for the *Andrew Filesystem* (AFS), GridFTP uses a PC to control access to an FTP server, and GridSQL uses PCs to restrict which SQL operations can be performed.

GSSKLOG The GSSKLOG application is used by Grid installations (such as Dartmouth's Green Grid) to allow users to obtain AFS tokens. Our design uses SHEMP to give the GSSKLOG application more information about the requester. The application essentially constructs a policy very similar to Alice's KUP which protects a resource, in this case, the AFS token. The policy can then specify what attributes the client must possess in order to obtain a token, and may further restrict the permission level of the token based on the requester's environment.

For example, assume that Alice is an AFS administrator and requests a token which grants her administrative privileges. The GSSKLOG application may construct a policy which allows Alice to obtain an administrative token from her own desktop, but not from anywhere else. Similarly, the application may wish to restrict token privilege levels based on whether the requester is behind a firewall, on a public terminal, using secure hardware, etc.

GridFTP The GridFTP application currently supports the use of Proxy Certificates for authentication. We extended the system to use the SHEMP system, specifically, the environmental information found in the requester's PC (as above). This information allows the GridFTP application's policy to be more fine-grained and restrict access to specific FTP operations (e.g., `get` and `put`) depending on the requester's environment.

Another potentially interesting use of SHEMP would involve the GridFTP application restricting access to parts of the FTP server's filesystem based on environmental information. Concretely, this could be achieved by having the GridFTP application construct a policy (again, similar to Alice's KUP) which protects subdirectories in the filesystem tree. For example, the policy may specify that certain parts of the system may only be visible to clients who are inside the firewall. Other examples consist of the GridFTP application not making non-anonymized medical data available to clients coming from public terminals, or disallowing the `put` operation from being performed from public terminals.

GridSQL The last application in this category is a MySQL database which uses PCs for authentication. In the Grid middleware arena, there is much discussion of wanting to restrict access to database information. In our design, SHEMP could be used to restrict subsets of data and/or operations (e.g., the `DELETE` operation) to parties operating within specific parameters. For example, a database of medical information way wish to disallow the `JOIN` operation for any client which is not on a "HIPAA compliant" machine, as the operation could likely link fields such as name or other identifying information and diagnosis.

6.3 The Bigger Picture

In addition to the PKI primitives and specific Grid applications discussed in the previous sections, we can envision broader application areas that might benefit from SHEMP. Instead of just enhancing a particular service or application, we explore uses of SHEMP which enhance a suite of related applications or an entire application area.

Computational Grids By far, the largest use of the X.509 Proxy Certificate approach is the Grid community [132]. In fact, the X.509 Proxy Certificate standards [126] were drafted by members of this community, and the major Grid toolkit (i.e., Globus [32]) includes Proxy Certificate support. 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.

Many applications in the Grid community could benefit from SHEMP. The price for adoption is relatively low since the community is used to MyProxy and PCs, and SHEMP adheres to many of the Grid community's programming norms. The benefits of using SHEMP are high: utilization of secure hardware, the inclusion of environmental attributes in the PC, and uses of PCs for decryption and signing.

Mobile Clients Another interesting application area for SHEMP is one in which Alice has a number of different machines (sometimes called a *constellation*) which all have a need to use the key, but vary in levels of security. For example, it is not uncommon for a user to own a desktop (possibly equipped with secure hardware), 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. Alice's KUP could limit the temporary key to performing only certain operations (e.g., encryption) on certain devices.

In this area of power constrained devices, schemes such as SWATT [113] enable a form

of attestation which can be used to identify a platform. SHEMP's flexibility regarding authentication and attestation mechanisms allows it to be extended to new attestation schemes as they become available.

Multi-Level Security and Workflow Much of the security thought in the Orange Book era involved *Multi-Level Security* (MLS). Details can be found in the Orange Book [90], but the general idea is to allow different classes of users to access different sets of information. For example, if Alice possesses a "Secret" security clearance, then she should not be allowed to view information labeled as "Top Secret", as this information exceeds her clearance level. The term "MLS" is used to describe a system which allows multiple access classes and enforces access policies.[4]

While the MLS paradigm is not widely employed in practice, a new breed of applications (often called *workflow applications* [38, 111]) are implementing some of the MLS techniques at the application layer. Such applications often define different business units (or specific roles) as the access classes, and then use a markup language (e.g., XML) to label portions of a document. For example, Alice may work in development, Bob in human resources, and Charlie may be the CEO. Charlie may send a company wide memo which contains payroll information. Since Alice has no need to know this information, Alice's email application should refuse to display that portion of the memo. However, it may be relevant to Bob, and he is "cleared" to see such information, so his email client should display it. As another example, assume that the three parties are jointly working on a spreadsheet. Certain rows and columns should not be accessible to certain parties, depending on the relevance to their department.

SHEMP provides a foundation on which to build such applications. Specifically, since SHEMP PCs contain the user's current environment (i.e., the attributes), applications can

[4]Much of the theoretical foundation can be found in the seminal work of Bell and LaPadula [5].

use environmental attributes in addition to authentication or role information to make access control decisions. The Grid applications of the previous section (i.e., GSSKLOG, GridFTP, and GridSQL) illustrate scenarios where applications make decisions based on this environmental information. The SHEMP policy framework allows applications to have very fine-grained access control policies, to the point of restricting access to programming constructs such as functions, objects, and data structures. Returning to the spreadsheet scenario outline above, if the application were built using SHEMP, then the application could not only hide access to certain rows or columns based on *who* is viewing them, but also *where* they are viewing it from. For example, no one may see salary information from a public terminal.

6.4 Chapter Summary

The last chapters explored the SHEMP system in detail. We began by introducing the tools used to build SHEMP: MyProxy and Proxy Certificates, Secure Hardware, and the XACML policy language. In Chapter 5, we covered the SHEMP design from a conceptual perspective. We introduced the special SHEMP parties: the Repository and Platform Administrators, as well as the system-level components such as the repository, CA, and client platform. We also explored the system from a specification and implementation level, demonstrating how to set SHEMP up and use it. Finally, in this chapter, we covered a range of applications, starting with ones we implemented (the proxies), and ending in ones where we think SHEMP could have something to offer. In the next chapter, we give an analysis of SHEMP, and illustrate how it meets the criteria of Chapter 3.

Chapter 7

Evaluating SHEMP

The goal of SHEMP is to make standard desktops usable PKI clients. To meet this goal, SHEMP must first be a practical system which does not require the entire desktop OS to be re-engineered and/or rewritten. As we demonstrated in the last chapter, SHEMP satisfies this practicality requirement: it is a real system that runs as an application on a standard desktop. Furthermore, we expanded on this practicality requirement by showing how we used SHEMP to construct real-world applications (the decryption and signing proxies) and by discussing how we could use SHEMP to build numerous other applications.

While practicality is a necessary property of a successful solution to the desktop PKI problem, it is not sufficient. In addition to showing that SHEMP is practical and can be used to build real applications, we need to show that SHEMP offers security, mobility, and flexibility—and that the complexity of these features do not overwhelm users (end users as well as administrators and application developers). Most importantly, we need to show that SHEMP can allow relying parties to make reasonable trust judgements.

In Chapter 3, we established criteria that define the notion of what it means for a desktop to be usable as a PKI client. We argue that any solution which claims to be usable as a PKI client should not only be practical, but also do the following:

1. minimize the risk and impact of key disclosure,

2. allow for client mobility,

3. be usable to application developers and users, and

4. enable relying parties to make reasonable trust judgments.

We claim that the SHEMP system as described in Chapter 5 meets all of these criteria, and thus makes desktops usable PKI clients. In this chapter, we offer support for this claim by using analysis and experiments to show how SHEMP meets the first three criteria. In Chapter 8, we illustrate how SHEMP meets the last criterion through the use of formal methods.

Chapter Outline We begin this chapter by analyzing how well SHEMP reduces the risk and impact of private key disclosure in Section 7.1. In Section 7.2, we show how SHEMP gives clients mobility without sacrificing security. Section 7.3 addresses SHEMP's usability by presenting the results of a SHEMP usability study and performance tests. Finally, Section 7.4 summarizes and concludes the chapter.

7.1 Minimizing the Risk and Impact of Key Disclosure

In Chapter 3, we defined the notion of security as minimizing the risk, impact, and opportunity for misuse of a user's private key. This definition is a critical component of making desktops usable PKI clients.

In Chapter 2, we demonstrated how desktops fail to meet this definition of security. Because of a large Trusted Computing Base (TCB), executing just one piece of code with a victim's privileges can lead to a private key disclosure. In the modern computing landscape, there is a significant risk of an attacker executing code on a victim's desktop—such

vulnerabilities are published daily on mailing lists such as BugTraq [112]. The presence of this risk makes it impossible for Bob, upon receiving a request claiming to be from Alice, to know if she is really aware of and intended the request. There is a good chance that someone other than Alice has access to her private key, which makes it impossible for Bob to make reasonable trust judgements about Alice, thus leaving desktops unusable as PKI clients.

We claim that SHEMP meets this definition of security and makes desktops secure. SHEMP reduces the risk of private key disclosure by using a number of methods to shrink the TCB. SHEMP reduces the impact of key disclosure and opportunity for misuse by allowing the TCB size to vary in time.

Since security is impossible to quantify, we offer a security analysis of SHEMP to back our claim. While such an analysis is not as strong as empirical evidence or formal proof, it is commonplace within the secure systems space for two reasons:

- Empirical studies to measure system security typically involve a "penetrate and patch" methodology, where systems are given to "ethical hackers" in order to locate vulnerabilities. Such a methodology cannot show that a system is secure (i.e., has no vulnerabilities), only that vulnerabilities exist.

- Formal proofs often prove that a *model* of the system is correct—not the actual system itself. If the model fails to capture *any* feature or state of the actual system, then proving that the model is correct does not naturally imply a proving the actual system is correct.

As a result, the secure systems community often relies on analysis (possibly in combination with other techniques) when reasoning about the level of security in a system. We do the same when reasoning about SHEMP's security.

Risk Analysis The security community at large has adopted the risk analysis model for evaluating and designing security. Security architects typically begin by auditing their environment, defining critical components, and assessing the likelihood and consequences of compromise given a specific threat model. They can then allocate resources in such a way as to protect high-risk components first. SHEMP was designed with this mode of thinking in mind.

Before we offer a security analysis of SHEMP, we state an important assumption which holds throughout our analysis: the level of security in SHEMP (or any system) cannot be measured with a single bit. It is *not* the goal of our analysis to conclude some meaningless statement such as "SHEMP is secure." Rather, our analysis aims to illustrate how SHEMP can be used to increase security in a wide range of environments with possibly different threat models. We will show how SHEMP creates a framework which makes it possible to build a secure PKI environment (i.e., one which minimizes the risk and impact of key disclosure) under an array of threat models.

7.1.1 Minimizing Risk

SHEMP decreases the risk of private key disclosure in a number of ways. First, SHEMP removes users' keys from the desktop and places them in a credential repository which is administered by a professional. Placing keys in a repository shrinks the TCB. The TCB is expanded to cover the desktop only when needed, and only for a short period of time. Second, by using secure hardware when available, SHEMP can reduce the TCB size even further. Finally, the SHEMP policy mechanism includes environmental information (i.e., repository and platform attributes) in each user's Proxy Certificate (PC), which allows relying parties to decide for themselves whether they should trust the request. Furthermore, user and application policies give entities a way to control the TCB based on the user's

security context.

Getting Keys Off of the Desktop As we explained in Chapter 2, modern desktops have a large TCB which makes them susceptible to attacks such as keyjacking. SHEMP reduces risk of key disclosure by taking private keys off of the end user desktops machines (or devices such as USB tokens) and placing them in a SHEMP credential repository.

Since the TCB is a finite set of software and (possibly hardware) components, we can represent TCBs with set notation as the set TCB. We consider the TCB of the current client-side approach to be the union of the TCBs of all of the n client desktops in the domain. We denote this total TCB as T_{total}, where:

$$T_{total} = \bigcup_{i=1}^{n} TCB_i \ .$$

If *any one* of the n desktops in Alice's domain have the keyjacking malware installed, then anyone who uses that desktop will have their keys stolen or misused. Solutions which encourage mobility (i.e., allowing users to store their private keys on USB dongles) actually make matters worse, as a compromised machine is likely to service a number of users. In this case, all of the users of the compromised machine could have their key stolen or misused. If we assume that c of the desktops are infected with keyjacking malware, then Alice has a c/n chance of having her key stolen or misused. If the desktops are all roughly the same in terms of OS and software, and an attacker can compromise one of them, then it is likely that c can approach n very quickly (e.g., if the keyjacking malware were propagated by a worm or virus), leaving it almost certain that Alice will be keyjacked.

Under the SHEMP approach, there is only one machine which houses users' private

keys: the key repository.[1] Centralization shrinks the total TCB from the n desktops to just one key repository when no one is using the system. When Alice needs to use her key, she requests that the repository extend the TCB to cover her machine for the duration of her session. Concretely, this is accomplished by the repository signing a short-lived Proxy Certificate for a temporary key on Alice's current desktop. The SHEMP TCB at some time t is the repository's TCB plus the TCB of whatever clients are involved in active sessions (i.e., have valid PCs) at time t. If we let $p(t)$ be the number of valid PCs at time t, we can denote SHEMP's total TCB at time t as $T_{shemp}(t)$, where:

$$T_{shemp}(t) = (TCB_{rep}) \cup \left(\bigcup_{i=1}^{p(t)} TCB_i \right) .$$

Assume that organization S uses SHEMP, and that organization O does not. Additionally, assume that they have the same number of machines (denoted n), and that one machine is serving as S's key repository (leaving S with $n-1$ clients, i.e., $p(t) \leq n-1$). The TCB at S is never greater than the TCB at O, i.e., $\forall t : |T_{shemp}(t)| \leq |T_{total}|$ because:

$$\left| (TCB_{rep}) \cup \left(\bigcup_{i=1}^{p(t)} TCB_i \right) \right| \leq \left| \bigcup_{i=1}^{n} TCB_i \right| .$$

To see why this statement is true, assume that every user in S has a valid PC at some time t. In this case $p(t) = n-1$, which yields the same size TCB as O. The implication is that if any client desktop does not have a valid PC, then the SHEMP approach shrinks the TCB.

[1] Actually, the SHEMP design allows for a number of repositories, but we envision a small number of repositories in relation to clients.

From Alice's perspective, the risk of compromise is not noticeably different, however. If we let c of S's machines be compromised, then Alice still has c/n chance of using a compromised machine. However, the impact of using a key is reduced, as the attacker can only keyjack a temporary keypair for a short period of time (this issue will be discussed in detail in Section 7.1.2). As discussed in Chapter 2, the status quo allows an attacker to keyjack Alice's real private key for an indefinite period of time.

From a risk analysis viewpoint, all of the desktops are critical resources in the status quo. If an attacker gets the keyjacking malware installed on any desktop in the domain, he is likely to get the private keys on that desktop. From the organization's perspective, resources must be allocated in such a way that treats all of the desktops as equals. Given a fixed amount of resources, this allocation scheme gives fewer resources to all of the desktops. Such strategies prevent expensive secure hardware (such as the IBM 4758) from being installed on all of the clients.

Under SHEMP, the private keys are stored in one place: the repository. An attacker has a higher reward for a successful attack on the repository, but a lower reward for successful attacks on the client desktops. From the organization's perspective, this consolidation is useful; it allows the organization to allocate resources where they matter most—at the repository. Instead of spending a little on many machines, the organization can spend more on a few critical machines. This approach allows the organization to get economies of scale by using secure hardware at the repository.

SHEMP also minimizes the risk of private key disclosure by placing all of the private keys under the control of a trusted entity: the Repository Administrator. The Repository Administrator will likely be closely related to the organizational unit which issues certificates (i.e., the Certificate Authority). A specialist is more likely to protect the private keys than an individual user is. Additionally, users can lose devices such as USB dongles. Thus, letting a specialist care for the private keys decreases the risk of private key disclosure.

Using Secure Hardware As we have discussed throughout this thesis, secure hardware can potentially shrink the TCB. Highly secure devices such as the IBM 4758 can provide a separate security domain from their host. As discussed in Chapter 4, secure platforms such as our Bear/Enforcer platform can provide some level of protection, and cost significantly less than an IBM 4758. SHEMP can reduce the TCB (and hence, the risk of private key disclosure) further by taking advantage of secure hardware, when and where available.

Since the keys reside in a central location, we envision that the repository will utilize some form of secure hardware. Ideally, the repository application should be running in secure hardware, and the private keys should be stored inside. The organization's threat model should dictate the level of secure hardware that they adopt. For maximum security, the repository should run in a device such as the IBM 4758, and the clients should minimally use something like Bear/Enforcer.

In terms of the TCB equations, the use of secure hardware makes the value of the TCB variable smaller. We denote a TCB with secure hardware as TCB_{shw}, and define the relationship between hardened and non-hardened TCBs as $TCB_{shw} < TCB$. Revisiting the TCB of our organization using SHEMP (S), if we assume that all of the nodes use secure hardware and that the number of PCs are equivalent at time t, we get:

$$\left|(TCB_{rep}) \cup \left(\bigcup_{i=1}^{p(t)} TCB_{shw_i}\right)\right| < \left|(TCB_{rep}) \cup \left(\bigcup_{i=1}^{p(t)} TCB_i\right)\right| .$$

The implication is that the use of secure hardware allows organizations with SHEMP to shrink the TCB even further, thus further decreasing the risk of private key disclosure.

Policy and the TCB Finally, SHEMP also minimizes the risk of key disclosure through the use of the environmental attributes (found in the Repository Attribute Certificate and

Platform Attribute Certificate) and Key Usage Policies (KUPs) found in each Proxy Certificate's PCI extension. SHEMP mandates that all of this information be included.

Our approach gives useful security information to relying parties, allowing them to adjust their trust in the client based on the environment. Relying parties are thus aware when a client generates a temporary key under conditions which are likely to result in key disclosure, and have the possibility to limit the key's use. The usability of the policy statements in the context of building applications will be examined in a usability study described in Section 7.3.

Moreover, the use of policy information can shrink the TCB even further. As we have discussed, Alice's KUP may limit the machines that she can use to perform key operations. We denote the number of machines that Alice can operate from as the number m. We can denote the total TCB from Alice's perspective as T_A where:

$$T_A = (TCB_{rep}) \cup \left(\bigcup_{i=1}^{m} TCB_i \right) .$$

In order for Alice to consider a particular machine in her calculation, her KUP must allow her to operate from that machine. If Alice's KUP restricts her from using *any* machine, then the TCB shrinks from Alice's perspective as $|T_A| \leq |T_{total}|$. To see why this is true, note that the expansion of the statement $|T_A| \leq |T_{total}|$ yields the following:

$$\left| (TCB_{rep}) \cup \left(\bigcup_{i=1}^{m} TCB_i \right) \right| \leq \left| \bigcup_{i=1}^{n} TCB_i \right| .$$

Recall that m is the number of machines that Alice may operate from, and that n is the total number of machines in the organization. Additionally, recall that there are $n - 1$

desktops, resulting in the relationship $m \leq n - 1$. If Alice's KUP restricts her from using any machine in the organization, then $m < n - 1$ and $|T_A| < |T_{total}|$.

It is worth noting that SHEMP's use of policy not only shrinks the TCB when compared to the current client-side approach, but also shrinks the TCB when compared to the MyProxy system. We consider the total MyProxy TCB from Alice's perspective to be union of the repository's TCB and all of the $n - 1$ desktops (recall that there are n machines, and one of them is serving as the repository, which results in $n - 1$ desktops). We denote this TCB TM_A, where:

$$TM_A = TCB_{rep} \cup \left(\bigcup_{i=1}^{n-1} TCB_i \right) .$$

The relationship between Alice's view of the TCB under SHEMP and Alice's view of the TCB under MyProxy is $|T_A| \leq |TM_A|$:

$$\left| (TCB_{rep}) \cup \left(\bigcup_{i=1}^{m} TCB_i \right) \right| \leq \left| (TCB_{rep}) \cup \left(\bigcup_{i=1}^{n-1} TCB_i \right) \right| .$$

In the case where Alice's KUP does not limit her from using any machines, we get $m = n - 1$, and the total TCBs are the same. However, if Alice's KUP restricts her at all, then $m < n - 1$ which shrinks the total TCB under SHEMP. Furthermore, if any of Alice's client machines use secure hardware, then the SHEMP TCB will be smaller than the MyProxy TCB.

7.1.2 Minimizing Impacts

In addition to minimizing the risk of a private key disclosure, SHEMP minimizes the impact of such a disclosure in the event that it does occur. First, a successful keyjacking-style attack only gives the attacker access to a temporary keypair, and only for a limited period of time. Second, SHEMP reduces the impact of a disclosure to the organization by simplifying and shrinking the size of Certificate Revocation Lists (CRLs). Finally, SHEMP makes forensics easier by consolidating (and possibly protecting) the audit trail.

Closing the Window SHEMP minimizes the impact of private key disclosure at the client by only allowing the temporary key to be used for a short time. In Chapter 2, we showed that key disclosure is disastrous within the current client-side infrastructure. In many cases, an attacker is able to obtain the private key himself, or use it to perform arbitrary operations for an indefinite period of time. Under SHEMP, the key issued on the client's desktop is valid for a number of hours (our prototype defaults to two hours). This small time window limits the opportunity for a successful attacker to use the victim's key.

The set of operations that an attacker can perform with a stolen key is possibly further limited by the victim's KUP. A successful attacker may not have access to the encryption or signing proxies (or other resources in the domain) depending on how the victim has set her KUP. Therefore, a restrictive KUP can also limit the impacts of a private key disclosure.

If Alice fails to log off of the SHEMP repository and destroy her temporary private key, then it may persist on the client machine after the Proxy Certificate has expired. In the event that an attacker successfully compromises a machine in such a state, he may be able to wield Alice's (expired) temporary key. If the attacker accesses information on the client which was encrypted with the corresponding public key (Alice's KUP would have had to allow her to decrypt from the client machine), then he can decrypt that information. The attacker cannot generate signatures or use Alice's long-term private key on the repository,

as the decryption and signing proxies would notice that Proxy Certificate is expired.

Revocation SHEMP minimizes the impact to the organization in the case of a key compromise. In many status quo PKIs, compromised keys are revoked by placing their certificate into a CRL or an *Online Certificate Status Protocol* (OCSP) server. Keeping CRLs up to date and distributing them are non-trivial problems in the PKI space, and make for interesting research problems in themselves (e.g., [16, 41, 56, 68, 82]).

Since only the SHEMP repository can use Alice's private key, SHEMP can effectively revoke a user's keypair by changing the authentication information at the repository. Changing Alice's authentication information (e.g., changing her password) results in Alice (or anyone with Alice's login information) being unable to log on to the repository and make requests to use her private key. This approach greatly reduces the size of CRLs, and thus reduces the amount of work for the organization's administrative staff. Such approaches are often given as advantages to the credential repository approach (e.g, MyProxy [64, 88] and SEM [10, 17]).

The Audit Trail Finally, SHEMP minimizes the impact of a private key compromise by consolidating the audit trail used for gathering forensic information. Since all accesses to use Alice's private key (either to generate a PC or to perform some operation) are received by the SHEMP repository, there exists a central log of Alice's private key activity on the repository. In the event that Alice's key is compromised, investigators need only look in one place for information.

Furthermore, since the SHEMP repository software can run inside of secure hardware, SHEMP can secure the logs themselves by keeping them inside of the hardware. The logs could be cryptographically protected to prevent tamper or viewing by unauthorized individuals. In the event of a key compromise, the organization would not only have a

central location for the logs, but can protect them against modification (possibly by the attacker).

7.1.3 Keyjacking Revisited

We continue our security analysis of the SHEMP system by offering a concrete example which illustrates the differences between the current client-side infrastructure, MyProxy, and SHEMP. As our example, we will revisit the keyjacking attack of Chapter 2, and explain how the different approaches effect the risk and impacts of a private key disclosure.

Threat Model Suppose that a malicious party named "Kevin the Keyjacker" would like to steal Alice's private key, and then use it to generate signatures. Recall from Chapter 2 that, in order to get Alice's private key, Kevin has to first get a small executable to run on Alice's machine with her privileges. Furthermore, assume that Kevin is in some remote location, that is, he does not have physical access to Alice's machine. Our assumptions about Kevin essentially define the threat model for this example: a remote attacker who can run arbitrary code on Alice's machines with her privileges.

As we have shown, the current client-side infrastructure cannot protect Alice's key given this threat model. MyProxy is somewhat of an improvement, as it gets the private key off of the client desktops. However, without taking advantage of secure hardware on the client, and without a method for describing Alice's current environment, MyProxy cannot adjust to handle stronger threat models (e.g., one where the attacker has physical access to the client). SHEMP as we described in Chapter 5 can prevent this attack (discussed below), and through the use of secure hardware and its policy language, can compensate for a range of threat models. This ability to compensate for a range of threat models gives SHEMP an advantage over the status quo and MyProxy.

Risk of Private Key Disclosure Once Kevin has gotten his code to execute (e.g., possibly by causing a buffer overflow) on Alice's machine, he can get access to her private key and use it to generate signatures. To Kevin, every machine in Alice's domain has roughly the same chance to lead to a successful attack. Kevin need not target critical servers or infrastructure, attacking any standard desktop will likely result in gaining access to some user's private key (he may get Bob's key instead of Alice's, but he has accessed a user's private key nonetheless).

Comparing MyProxy to the current client-side infrastructure is not an entirely fair comparison, as MyProxy is not a general-purpose PKI solution, but an authentication and authorization system (i.e., MyProxy does not allow users to sign or decrypt messages with their Proxy Certificate). In order to give Alice access to all of her private key operations, the MyProxy repository would have to export Alice's private key to her machine when she makes a request (such operations are permitted in MyProxy). Using this approach, MyProxy still offers an advantage over the current client-side infrastructure, as Alice's private key resides on her desktop for only a short period of time. During that time, however, Alice is susceptible to a keyjacking attack from Kevin.

Kevin's strategy will likely change if Alice uses MyProxy. While Kevin can compromise a desktop, he may now have to wait for some time before he can steal a key (i.e., until Alice logs in to MyProxy and migrates her private key to the desktop). Furthermore, unless he steals her key outright or Alice forgets to log out, he has a limited window where he can use her key. If Kevin steals Alice's password, then he can use her key from anywhere. To Kevin, attacking the MyProxy repository has the highest payout, as he could access to *every* user's key in the organization. Even if the keys are stored a very secure device at the repository (e.g., an IBM 4758 [64]), Kevin can use the keys at will by inserting his malware into the MyProxy repository application itself. Since the hardened MyProxy approach does not place the application in secure hardware, it is susceptible to attack.

Under SHEMP, it is possible for Alice's organization to configure the system in such a way as to minimize the risk of key disclosure even further. Assume that Alice's organization runs one SHEMP repository inside of a very secure device such as the IBM 4758. In contrast to the hardened MyProxy approach mentioned above, assume that the entire SHEMP application is running inside of the device. Furthermore, assume that the client desktops are running on Bear/Enforcer platforms.[2]

Now assume that Alice has obtained a temporary keypair and Proxy Certificate on her client. In this configuration, and assuming that the Platform Administrator has configured the Bear/Enforcer platforms appropriately, Kevin's keyjacking malware will violate the victim application's integrity policy and cause Enforcer to halt the system. Kevin can thus deny service to Alice, but he cannot obtain her private key—not even her temporary one—by attacking her desktop.

As with MyProxy, Kevin's optimal strategy is to access the repository. However, since we are assuming that the current SHEMP configuration places the repository application and keys inside of an IBM 4758, Kevin is unable to get his malware into the application, thus preventing him from accessing the private keys. If we assume a stronger threat model where Kevin has physical access to the SHEMP repository, his tampering with the IBM 4758 would cause it to destroy the private keys, denying service to the entire population, but still refusing to disclose private keys. If we were to assume a weaker threat model (e.g., where the repository runs on a Bear/Enforcer platform), then Kevin can install his malware if he can obtain root privileges on the repository or physically extract secrets from the TPM.

In summary, SHEMP reduces the risks of private key disclosure. The amount of risk reduction depends on the SHEMP configuration and the amount of resources an organization allocates to SHEMP. While it is possible to harden every machine in the domain with

[2]Our prototype uses Bear/Enforcer platforms for the repository and clients.

an IBM 4758, such a solution would be costly. However, the SHEMP system can take advantage of configurations involving just a few highly secure devices to provide a secure environment.

Impacts of Key Disclosure Under the current client-side infrastructure, if Kevin is successful, the impacts are devastating. As discussed in Chapter 2, Kevin can possibly obtain a copy of Alice's key to use at will. Minimally, he can wield Alice's key indefinitely without her knowledge. In order to fully recover from such an event, Alice needs a new keypair.

A key disclosure also burdens Alice's organization. Minimally, the certificate describing Alice's compromised keypair needs to be revoked by placing an entry for that certificate in the latest CRL. Furthermore, if Alice's key disclosure is important enough to investigate (e.g., perhaps she signs electronic payroll documents), then the organization needs to search through logs on multiple machines in order to reconstruct the attack timeline and discover what actually happened.

If Alice is using MyProxy and her key is stolen from her desktop, then she needs a new keypair, as before. The organization would still need to revoke her old certificate, but investigators would likely narrow their search for forensic evidence to a specific time interval and desktop on which Alice used her private key. If Alice just had a temporary keypair and Proxy Certificate on her desktop, then Kevin would just have a few hours to impersonate Alice. In this scenario, he can only use Alice's key for authentication (MyProxy does not let Alice use her private key for decryption or signing) and no certificates need to be revoked. If Kevin captures Alice's repository login information, then he can access her private key at will. However, assuming that Kevin does not export her key to his machine, the organization can effectively stop Kevin from further access by simply changing Alice's login information.

If Alice and her organization are using SHEMP, then her real private key can never be

stolen by a successful keyjacking attack on the desktop. A successful attack possibly gives Kevin access to Alice's temporary keypair (unless the client desktop is configured to halt the system, as described above). In this scenario, Kevin can use the key for a short period of time. Furthermore, depending on Alice's desktop, Kevin may only be able to use the key for a limited set of operations. Ideally, weaker desktops should have more restrictive policies, meaning that successfully attacking a weak client should have a lower payoff for the attacker.

The worst case scenario is where Kevin gets Alice's repository login information. If the organization is using passwords (as our prototype does), then it can change passwords at the repository, and effectively block Kevin from further accessing Alice's key. If the organization has configured SHEMP to use a stronger authentication method such as biometrics or a two-factor scheme, then Kevin may have a harder time getting Alice's authenticator.

In any event, if Kevin gets Alice's temporary keypair, the organization does not need to issue Alice a new keypair, and thus does not need to revoke her certificate. Kevin has a small window of opportunity for key misuse, at best (again, assuming that SHEMP does not halt the client). Even if Kevin is successful in getting Alice's repository login information, the organization still does not need to issue a new keypair to Alice; it can simply change Alice's login information. This eliminates the need for revocation once again. In the event that Alice's organization would like to investigate Alice's key disclosure, it has a centralized record of every access to Alice's private key at the key repository. If the repository is running in secure hardware (such as in our example scenario) then the logs will be protected by secure hardware as well, keeping them safe from tampering.

In any of the scenarios, if Kevin compromises the Certificate Authority's (CA) private key, the organization will essentially have to rebuild the PKI from scratch. All of the user keys will need to be reissued, and all of the user certificates (and any other certificates signed by the CA) will have to be revoked.

However, SHEMP introduces two new parties which insulate users from system complexity: the Repository and Platform Administrators. Since these parties are trusted throughout the domain, they make attractive targets. If Kevin were to compromise one of these private keys, then the certificates signed by that key will have to be revoked. The result is that the platform and repository identity and attribute certificates would have to be reissued. The KUPs can be left alone, however, as they are signed by the CA and effectively consider the Repository and Platform Administrators as roles.

7.1.4 Attacking SHEMP

We conclude our security analysis of SHEMP by offering a strategy of how we would attack SHEMP. In this analysis, we hope to illuminate the pressure points in the SHEMP design and define areas for future improvements. This analysis does not claim to cover every possible avenue of attack on the SHEMP system; certainly, someone will think of an approach we missed.

The Client The first avenue for attacking SHEMP involves the client. Perhaps the most obvious attack involves an attacker installing a keystroke logger on a low-security client machine. If the target organization is using passwords for user authentication (as our prototype does), then an attacker will be able to capture Alice's username and password as she logs on to the repository. With this information, the attacker can then move to high-security machines which are possibly less restricted by Alice's KUP, and essentially use Alice's key at will. While there are numerous approaches to make the task of installing a keystroke logger more difficult for the attacker, the only real solution to this problem is to harden clients in such a way that users have a trusted path to the SHEMP client software.

The second attack comes from an attacker spoofing the SHEMP client software, and presenting this spoofed version to users. When Alice presents her authentication informa-

tion to the spoofed client, the attacker can store a copy for himself. A client platform which enforces an integrity policy can prevent this type of attack. An example of such a platform includes a client running Bear/Enforcer. If the integrity policy is configured to cover the SHEMP client software, then the platform can detect when the attacker installs a spoofed version.

A third attack consists of an attacker keyjacking the temporary private key from the client machine. This scenario is covered in detail in Section 7.1.3. An additional risk under this scenario comes about in the case where Alice does not properly log off of the repository and destroy her temporary private key. A successful keyjacker can use the private key to decrypt messages, even if the corresponding Proxy Certificate has expired. As with the previous attacks, this scenario can be mitigated by using secure hardware on the client to store the key.

The final client attack that we discuss involves an attacker spoofing a high-security client machine. If the target platform is a low-security or untrusted platform, then the end effect is that Alice may trust a low-security machine to perform high security operations. This attack, coupled with one of the previously mentioned client attacks, can lead to a temporary key and/or authentication credential compromise. As we discussed in Chapter 5, the best line of defense against this attack is to identify platforms with the least spoofable identifier available.

The Repository Another class of SHEMP attacks involve the key repository. Since the repository houses the private keys for the entire population, it is likely to be a target for attackers. While some of the client attacks could be profitable, the repository is susceptible to wholesale compromises as well.

The first attack we discuss is the scenario where an attacker attempts to compromise all of the users' private keys. The attack could potentially be performed remotely by accessing

the machine and copying the keystore. Our recommendation is to place the repository and the keys in a secure device, thus preventing remote compromise. However, an attacker with physical access may still be able to compromise the private keys, depending on the security level of the device. The best defense is to store the private keys and application in a device which can withstand local physical attacks, such as an IBM 4758.

The second repository attack consists of an attacker denying service to the keys stored in the repository. The most straightforward method for accomplishing this is to make numerous connection attempts to the repository. The more difficult and more disastrous denial of service attack involves the attacker physically tampering with the device (e.g., the IBM 4758) in an effort to trigger the tamper detection mechanism and force the device to destroy its secrets. This would destroy the private keys for the entire population. Defending against the remote denial of service scenario consists of throttling network traffic upstream, perhaps at the router. Placing the repository in an access controlled physical location can mitigate the risk of the second scenario.

Similar to the client-side attack mentioned above, an attacker that can convince a user that a low-security repository is really a high-security one could replace or modify the repository software in such a way that could lead to users' authentication information being captured. The best defense against this threat is to use the least spoofable identifier possible, possibly relying on secure hardware to authenticate the machine to clients.

Other Entities In addition to attacking the repository and client platforms, attackers may also target the most important keys in the organization: the CA's private key, the Platform Administrator's private key, and the Repository Administrator's private key.

An attacker with the CA's private key can issue and revoke any certificate belonging to the domain. This is a standard concern for PKI designers, and thus the CA's private key is often placed in a secure device of some sort. SHEMP introduces two new targets:

the administrators. If an attacker were to obtain either of these keys, the results would be disastrous. The attacker would be able to add false repositories and/or client platforms to the domain. This would make the task of collecting users' authentication information easy. To protect the administrators' keys, an organization should treat them with the same level of importance as the CA's private key, possibly protecting them with secure hardware of some sort.

7.2 Mobility

In Chapter 3, we established that a successful solution to the desktop PKI problem must allow users to migrate throughout the domain. As user populations become more mobile and begin to use multiple devices, a PKI must allow users to migrate across machines. Moreover, migration must not occur at the expense of security.

In Chapter 2, we discussed how the current client-side infrastructure fails to give users mobility. Currently, users must export their private key, transport it to their destination, and then import their keys at the destination. The intermediate format is often susceptible to attack and users must have exportable keys, meaning that users get mobility at the expense of security [35].

We claim that SHEMP gives users mobility without sacrificing security. We support this claim by analysis and demonstration using the SHEMP prototype. In our prototype testbed, we have three client desktops which are assigned a different set of security attributes. The desktops represent low-, medium-, and high-security machines. We are able to access our private key from each one, subject to the restrictions in our KUP.

The mobility of SHEMP stems from the fact that it is based on the MyProxy design. The use of the credential repository approach allows SHEMP users to access their keys from anywhere, provided that they can access the key repository. MyProxy's mobility is

what led us to use it as a foundation for the SHEMP design in the first place. In all fairness, we can claim that SHEMP is as mobile as MyProxy.

One area where SHEMP excels, however, is in the security properties which are maintained during migration. Again, the current client-side infrastructure makes migration risky by using unsafe transport formats, and by forcing users to have exportable keys. The use of a secure transport format such as Sacred [102, 103, 104] could provide some benefits, but it is not necessarily a part of what we consider the current client-side infrastructure. MyProxy is an improvement in that private keys typically stay on the repository, and only Proxy Certificates are given to the user. However, MyProxy does not consider Alice's environment when deciding whether or not to allow Alice to use her private key. As long as Alice—or anyone else—can provide the correct authentication information to the repository, it will allow her to access her key.

SHEMP takes the MyProxy approach a step further by actually checking the security properties of the current environment, and then consulting Alice's KUP to see if it should grant the request. Concretely, the SHEMP repository uses the platform authentication step to identify the requesting platform. As discussed in Chapter 5, the repository gives the attributes contained in the Platform and Repository Attribute Certificates along with Alice's KUP to a Policy Decision Point (PDP) for evaluation. If the PDP returns "Permit", then the request is granted. SHEMP's use of environmental information in making its access decision gives users the same amount of mobility as the MyProxy approach, while simultaneously providing extra security. Thus, SHEMP meets the mobility requirement of the criteria outlined in Chapter 3.

7.3 Usability

So far, we have shown how SHEMP achieves security and mobility, and as with most systems which try to provide security, the mechanisms which make SHEMP secure can also make it hard to use. One way that SHEMP achieves security and flexibility is through the use of policy. In order to meet the third criterion of Chapter 3 and show that SHEMP is usable to application developers and administrators, we need to show that developers and administrators can understand and construct valid policies to solve real security problems—i.e., the policy mechanism must be a valid medium for developers and users to express their mental models. Furthermore, we need to show that the computational overhead introduced by SHEMP's policy mechanism and use of extra keypairs does not make the system unusable from an end user's perspective.

7.3.1 User Study

To see whether the policy mechanisms were usable, we conducted a user study consisting of eight participants. While our sample size is small, it is highly representative of the types of people who would be tasked with constructing SHEMP policies.

Our user study outlined some real application designs taken from Dartmouth's Grid community (discussed in Chapter 6). Once the applications were designed, we gave the designs to subjects who would likely fill the roles of the Repository Administrator, the Platform Administrator, and the CA. We were interested in evaluating whether the parties could generate a meaningful set of policies which represent a given mental model, how many tries it took them, and their feedback regarding the difficulty of their task. The results indicate that the SHEMP policy mechanisms are usable, but a good policy construction tool is essential.

The Study Upon agreeing to participate in the user study, users were given a copy of the toolkit used to construct policies and a set of instructions for unpacking the toolkit and taking the test. The instructions best describe the parameters of the study, and are included below.

```
Introduction
============

This research project is being conducted by a graduate student (John
Marchesini) as part of a Ph.D. dissertation.  The purpose of this
experiment is to study the usability of the SHEMP system's policy
language and policy building tools from a system administrator's
perspective.

Your participation is voluntary and involves:

1) Installing a stand-alone Java test environment which contains the
   toolkit (or setting up an appointment to use my toolkit),
2) Using the toolkit to generate a SHEMP policy for three hypothetical
   applications, and
3) Completing the survey in this document.

Your information will be collected and maintained confidentially.  No
identifying information will be used in any presentation or paper
written about this project.

Feel free to contact me at:

Office : 063 Sudikoff Laboratory
Phone  : 603.646.9179
email  : carlo@cs.dartmouth.edu

Installing the tools
====================

The first step in this experiment involves installing the toolkit.
You will need to be using a machine with the Linux operating system.
If you do not have access to such a machine (or you do not feel like
installing software), you can contact me and we can set up an
appointment for you to take the test on one of my machines.  Obtain a
copy of the toolkit (policy.tar.gz) from me and unpack it:

tar xzvf policy.tar.gz

This will create a directory named 'policy' which has the following
contents:
```

INSTRUCTIONS	This document.
lib/	Libraries directory.
requests/	Directory containing XACML requests which I will use to test your policies.
src/	Directory containing Java source code for all of the tools used in this experiment.
testftp	A script I will use to test the GridFTP application policy.
testklog	A script I will use to test the GSSKLOG application policy.
testsql	A script I will use to test the GridSQL application policy.
tools/	A directory containing scripts that you will use to build policies for the three applications.

Generating Policies

From www.gridcomputing.com:

"Computational Grids enable the sharing, selection, and aggregation of a wide variety of geographically distributed computational resources (such as supercomputers, compute clusters, storage systems, data sources, instruments, people) and presents them as a single, unified resource for solving large-scale compute and data intensive computing applications (e.g, molecular modeling for drug design, brain activity analysis, and high energy physics). This idea is analogous to electric power network (grid) where power generators are distributed, but the users are able to access electric power without bothering about the source of energy and its location."

For the duration of this experiment, assume that you are an administrator working for Dartmouth's Grid installation (Green Grid). Specifically, you are in charge of administering three Grid applications: GSSKLOG, GridFTP, and GridSQL. Dartmouth's Green Grid uses the SHEMP security infrastructure to protect Grid-accessible resources. Your primary task today is to construct SHEMP security policies for the three applications under your jurisdiction using the tools in the tools/ directory.

Dartmouth has settled on the following set of attributes which may be assigned to all Dartmouth machines:

1) BelongsToDartmouth
2) LastPatchDate
3) BehindFirewall
4) PublicTerminal
5) OSType
6) IsLaptop

Any number of these attributes may be assigned to any machine in the Dartmouth domain. The meaning of the attributes should be straightforward, but the values warrant clarification. Some of the attributes may be set to either "true" or "false", and others have other values such as "windows", "linux", or "mac". The following table summarizes possible values for the attributes in the experiment:

Attribute Name	Possible Values
BelongsToDartmouth	true false
LastPatchDate	DD/MM/YYYY
BehindFirewall	true false
PublicTerminal	true false
OSType	windows linux mac
IsLaptop	true false

The person that assigns attributes to machines is the Platform Administrator, and is known as the "PlatformAdmin." Your policies should only trust attributes issued by the PlatformAdmin.

Each SHEMP-enabled application has a set of operations which should be governed by the policy (specifics are given below). The operation is allowed to execute if and only if all of the conditions are met, where conditions are (attribute, value) pairs. As the policy builder, your task includes assigning zero or more conditions to the operations. (Assigning zero conditions to an operation will allow the operation to always execute.)

In order to test your policy, I will construct a request to execute some specific operation in an application. That request will contain a set of (attribute, value) pairs assigned by the PlatformAdmin which will describe a hypothetical client platform. The request and your policy will then be given to a Policy Decision Point, which will determine whether or not your policy allows the operation to execute.

Dartmouth has decided on the following policies for the applications:

Application Name: GSSKLOG

Operation 0 : getToken
Conditions : The client machine belongs to Dartmouth

```
Application Name: GridFTP
-------------------------

Operation 0 : get
Conditions  : always allow

Operation 1 : put
Conditions  : The client machine belongs to Dartmouth
              The client machine is not a public terminal

Application Name: GridSQL
-------------------------

Operation 0 : SELECT
Conditions  : always allow

Operation 1 : INSERT
Conditions  : The client machine belongs to Dartmouth
              The client machine is behind a firewall

Operation 2 : DELETE
Conditions  : The client machine belongs to Dartmouth
              The client machine is behind a firewall
              The client machine is not a public terminal

Operation 3 : CREATE
Conditions  : The client machine belongs to Dartmouth
              The client machine is behind a firewall
              The client machine is not a public terminal
              The client machine runs the linux OS

To begin the experiment, cd into the tools/ directory and execute the
interactive self-explanatory scripts build_klog, build_ftp, and
build_sql.  The scripts will output three xml files (klog.xml,
ftp.xml, and sql.xml, respectively).  These files contain the
generated policy for the applications.  Feel free to look at these,
but please do not edit them.  You will need to get these files to me
for testing.  If you make a mistake, you can retake the test by simply
reexecuting the script.  Your previous policy will be overwritten.
```

Once users unpacked the toolkit, they began to take the test by executing the appropriate scripts, as indicated in the instructions. The scripts call our Java-based XACML policy generator with the appropriate arguments, and users were then guided through the necessary steps to build policies. A screenshot of the policy generator is shown in Figure 7.1.

After the user has finished constructing the policy with the generation tool, the tool outputs the XACML policy (see Figure 7.2). Once we received all of the user-generated policies, we constructed XACML requests (see Figure 7.3), and gave the requests and policies to an XACML Policy Decision Point (PDP) for evaluation. The PDP decides whether the policy allows the requested action to occur, given the current environment as expressed in the request. Using this technique, we were able to score the test subjects on their ability to construct a policy which matches the model presented in the instructions.

Results Subjects were tasked to construct policies governing a total of seven operations: one operation for `GSSKLOG`, two operations for `GridFTP`, and four operations for `GridSQL`. Although some of the operations had more attributes to consider than others, we decided to tally scores based on the output of the entire operation. If a subject set any or the attributes incorrectly, whether there was one or four attributes for the operation, then the answer was considered wrong. This approach best represents the real world: if any of the attributes are set incorrectly, then Alice may be able to perform an operation she should not be able to.

The overall results were positive. Half of the subjects built perfect policies, and of the remaining half, no one missed more than one operation. Furthermore, every mistake that was made resulted from a typographical error, such as a misspelled word or failure to respect case sensitivity. These results suggest that a policy generation tool which does not allow users to make such mistakes (possibly by doing input validation or presenting users with a graphical menu of options to choose from) would yield better results.

Once users had completed the test, they were asked to complete and return the a small survey (included below). The goals of the survey were to see if there was a correlation between an administrator's experience and their score and get some feedback on the usability of our tool.

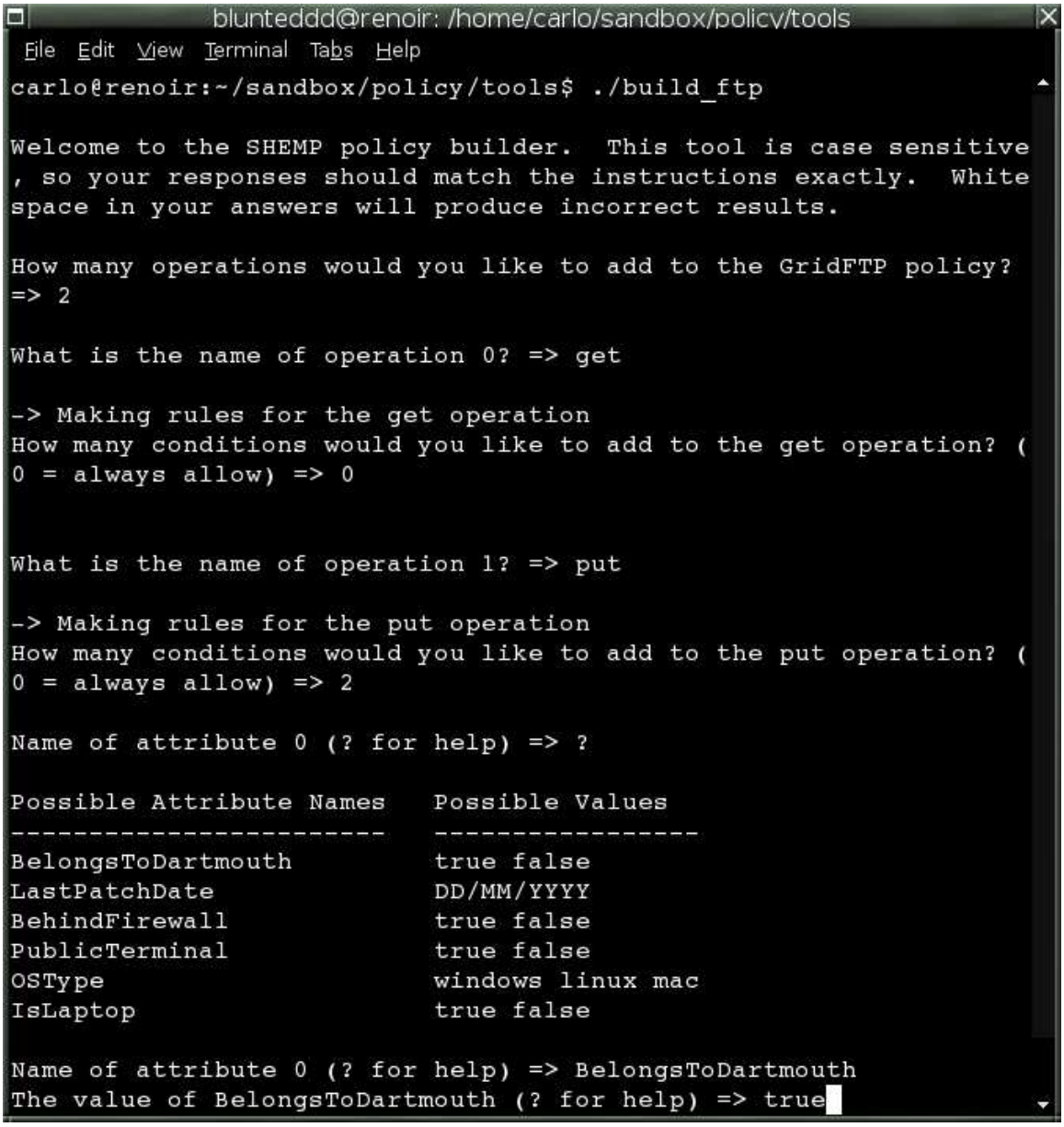

Figure 7.1: Our XACML policy generator.

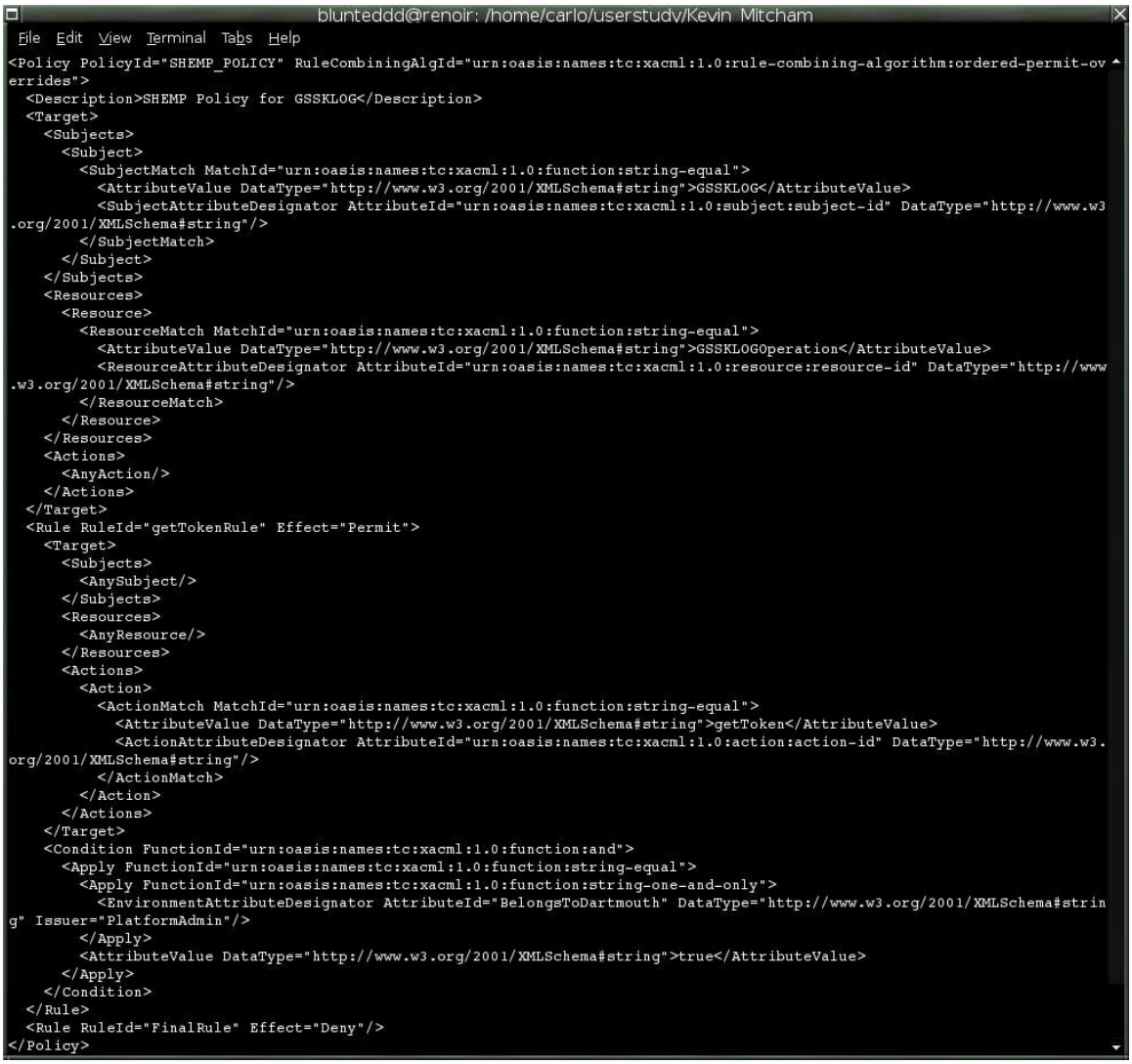

Figure 7.2: An XACML policy.

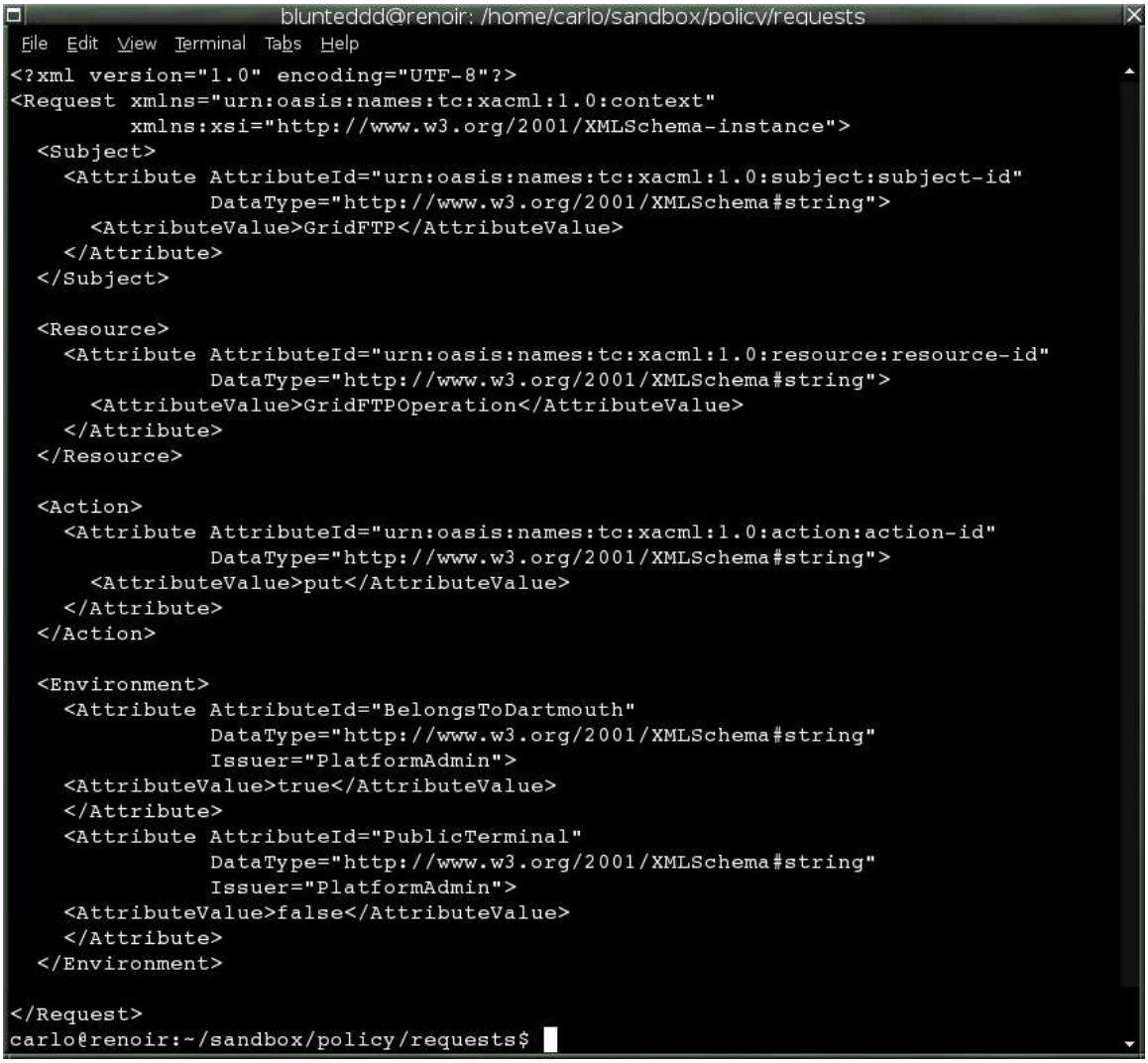

Figure 7.3: An XACML resource request.

The Survey
==========

Once you have successfully generated policies and sent them to me, please take a moment to fill out the following survey. When you have finished, please send me your answers.

1) Name :

2) Email address :

3) How many machines do you currently administer? :

4) What is the largest number of machines you have administered at one time? :

5) Have you ever administered an application for multiple users (e.g., an Apache web server, MySQL database, etc)? (If not, please skip to Question #12.) :

6) Did you install the application(s)? :

7) Did you configure the application(s)? :

8) Did you set the application's security settings? :

9) Was building a SHEMP security policy easier or harder than your previous experiences with other applications? :

10) What features of the SHEMP toolkit made it easier or harder? :

11) What was lacking from the SHEMP toolkit that made it easier or harder? :

12) Was there anything particularly confusing about the toolkit? :

13) If you answered yes to Question #12, please elaborate. :

14) What feature(s) would you add to the toolkit, if you could? :

15) Do you think any current features are unnecessary? :

16) If you answered yes to Question #15, which ones and why? :

17) How many times did you run the build_klog application? :

18) How many times did you run the build_ftp application? :

19) How many times did you run the build_sql application? :

Compiling the survey data led to a few interesting discoveries. First, there was an inverse correlation between the number of machines under the subject's control and the number of mistakes the subject made. The subjects who administered the most machines made the fewest mistakes. Second, of the subjects who had configured other application security policies, all but one of them said SHEMP was easier to configure. The one who said it was harder recommended using a GUI, and giving a users a way to go back. Many thought that the structure of the tool was helpful; they liked the question and answer tone rather than having a random access configuration file to edit. Third, no one reported anything particularly confusing about SHEMP, and everyone mentioned in one way or another that they would like a GUI with the potential to go back to the previous set of options. Finally, the subjects learned to use the tool rather quickly: no one reported running any of the scripts more than three times.

Analysis While our toolkit for testing purposes was a set of standalone applications, it is precisely representative of the logic found in our SHEMP prototype. The SHEMP repository (and the decryption and signing proxies running on the repository) constructs a well-formed XACML request with every private key request (i.e., Proxy Certificate generation, and message decryption and signing). The request contains the attributes from the verified Platform and Repository Attribute Certificates in the `Environment` tag of the request (see Figure 7.3), and the policy is Alice's KUP. The PDP is embedded in the repository, thus allowing requests to be made internally.

Applications can also use this approach to protect their resources. In the example applications described in the user study, these application resources are operations, such as the `GridFTP`'s `get` and `put` operations. However, applications could also use the policy to protect critical data (such as a private key) or other abstractions, such as parts of the `GridFTP` server's filesystem.

In summary, the user study suggests that the SHEMP policy mechanisms allow administrators and application developers to construct policies which match a given mental model of the system. The study also suggests that good tools can improve usability even further. A good GUI with easy navigability would further hide the complexity of using XACML, and improve the odds that policy builders construct accurate policies.

7.3.2 Performance Analysis

In order to show that the overhead introduced by SHEMP does not make the system unusable to end users, we instrumented the SHEMP prototype so that it would accurately report performance measurements. Performance is not the most interesting aspect of SHEMP, but since a third party is contacted for all private key operations, we expected a slowdown and wanted some quantification. If SHEMP keeps users waiting for long periods of time to perform key operations, then users are likely to find faster solutions, even at the expense of security.

Once we instrumented the code, we used our prototype testbed to measure the overhead of Proxy Certificate generation and the decryption and signing proxies. As a baseline, we compared SHEMP to a simple Java application which we call the SHEMP `CryptoAccessory`. The `CryptoAccessory` performs the standard cryptographic operations using a locally-stored keypair, and without third-party involvement. This baseline tool performs the suite of public key cryptographic operations (encryption, decryption, signing, and verification), and is included in the SHEMP toolkit.

The Testbed As discussed in Chapter 5, our testbed consists of four machines, two running the Bear/Enforcer platform, one running a Redhat Linux distribution, and the other running Debian. For our performance tests, we only used three of the four machines, and placed them in different network configurations to capture the effect of network delay in

our test. While this makes our results look worse, it is a more accurate picture of how SHEMP performs in real environments.

For the first configuration, we put the SHEMP client and the repository on the same machine, thus eliminating network delay altogether. In the second configuration, we placed the client and repository on the same Ethernet segment, so as to simulate a Local Area Network. For the final configuration, we put the client and repository on different networks by putting the client on our campus-wide wireless network.

Results Once our testbed was configured properly, and set to report timing measurements, we began a series of experiments. We measured the slowdown for the three operations (Generate Proxy Certificate, Decrypt, and Sign) on the three network configurations (Local, Same Segment, Different Network). We calculated each of the nine data points by taking an average of ten runs with our `CryptoAccessory` and SHEMP, and then calculating the slowdown introduced by the SHEMP overhead.

We first calculated a baseline by using our `CryptoAccessory` to perform ten runs of each operation in each configuration. The operations consisted of generating an RSA keypair, using it to decrypt a message, and then using it to sign a message. We then ran ten of the SHEMP versions of the operations, and used our SHEMP client to do the following:

- generate an RSA keypair, and then ask the repository to sign a Proxy Certificate which contains the newly generated public key;
- ask the decryption proxy to decrypt a message which was encrypted with our long-term public key and re-encrypt it with our temporary public key, and we then decrypted the message with our temporary private key;
- sign a message with our temporary private key, and then ask the signing proxy to verify the signature and sign the message with our long-term private key.

Operation	Local	Same Segment	Different Network	Average
Gen. PC	3.69%	1.94%	4.33%	3.32%
Decrypt	54.95%	42.64%	54.42%	50.67%
Sign	40.22%	49.04%	57.51%	48.92%

Table 7.1: Slowdown of SHEMP compared to local private key operations.

Note that all of the SHEMP operations also perform a policy check on the repository. The repository (and decryption and signing proxies) locate all of the appropriate policies and attribute certificates, verify their signatures, and pass the information to the PDP for evaluation. Further details of these operations can be found in Chapter 6. The results of our experiment are given in Table 7.1.

Analysis The column labelled "Average" is an average over the results of the different configurations. The results indicate that only 3.32% of the time spent generating a Proxy Certificate is used by SHEMP. This extra time that SHEMP introduces is used to transport the unsigned Proxy Certificate over the network, verify the current environment's attribute certificates, perform a policy check against Alice's KUP, sign the Proxy Certificate, and return the signed PC to Alice. The other 96.68% of the time is spent generating the temporary keypair on the client.

The performance results for the proxies is less impressive, indicating that roughly half of the time spent performing the operation is introduced by SHEMP. In these cases, SHEMP uses the time to transport the messages over the network, perform a policy check, perform a public key operation (either to verify the message or to encrypt the message with Alice's temporary public key), and perform a private key operation (either to sign a message with Alice's long-term private key or decrypt a message which was encrypted with her long-term public key). From a user's perspective, using SHEMP doubles the time it takes to perform a private key operation.

However, this is not as bad as it appears. First, the extra time needed for SHEMP may not be noticeable to humans. For example, over the average of the ten decryption operations performed in the "Different Network" configuration, SHEMP takes the time of the operation from 222.3 milliseconds to 487.8 milliseconds. It is likely that human perception can not detect the slowdown. If the network is lagging, then the time of the operation is likely to grow even more, but then any application using the network will feel a loss of performance as well. Second, it is possible to reduce the overhead by using cryptographic acceleration hardware. Our prototype repository used the default Java cryptographic provider to perform the operations. If we run the repository in an IBM 4758, we could exploit the cryptographic acceleration subsystem to improve performance.

Our performance analysis indicates that the overhead introduced by SHEMP does not make the system unusable. While the performance hit is statistically significant, it can be improved via specialized hardware, and users are unlikely to notice the slowdown anyway.

7.4 Chapter Summary

In this chapter, we have shown how SHEMP meets the first three criteria of Chapter 3. We showed how SHEMP enhances security by shrinking the TCB, and by allowing organizations to configure SHEMP to meet their threat model. We offered an analysis of how SHEMP reduces the risk and impacts of a private key disclosure, and how SHEMP can prevent keyjacking-style attacks.

Using SHEMP allows an organization to adjust for the threat model which it deems relevant. Additionally, SHEMP reduces the risk of key disclosure by taking advantage of secure hardware on the clients, and by disallowing the key from ever leaving the repository. If SHEMP is configured in such a way that allows a user's temporary keypair to be disclosed, then the attacker can use the key for a short period of time and for a possibly

limited set of operations. The organization rarely needs to revoke certificates, and has a centralized and possibly protected set of logs if it decides to conduct an investigation.

We then illustrated how SHEMP gives users mobility without decreasing security. Finally, we presented the results of two studies we conducted which show that SHEMP is usable from the perspective of administrators, application developers, and end users.

As mentioned in Chapter 3, all of the above properties are necessary, but if the system cannot enable relying parties to make reasonable trust judgments, then SHEMP does not work. To clarify, we are not only concerned with issues regarding UI development or the standard concerns of the Human-Computer Interaction Security community. Rather, we must show that SHEMP allows relying parties to conclude what they ought to conclude, and disallow them from concluding things they should not. Thus, the ability to enable such trust judgments should be viewed as a correctness condition for SHEMP. Since this is the most important part of showing that SHEMP makes desktops usable for PKI, we have devoted the entire next chapter to discussing it.

Chapter 8

Making Trust Judgements

PKI systems are often complex distributed systems that are responsible for giving users enough information to make reasonable trust judgments about one another. Since the currencies of PKI are trust and certificates, users who make trust decisions (often called *relying parties*) must do so using only some initial trust beliefs about the PKI and some pile of certificates (and/or other assertions) they received from the PKI.

While there are a number of metrics we can use to reason about PKIs—such as their usability, efficiency, or expressivity—one measure stands out as the most important: correctness. We say a PKI is *correct* if it allows Alice to conclude about Bob what she should, and disallows her from concluding things she should not. If a relying party Alice is given a pile of certificates about Bob from a PKI, can she make reasonable trust judgments about Bob? Does she have any reason to believe that Bob is who he says he is? Does Bob really have the private key matching the public key in his certificate? Does Bob have the properties and attributes that his certificate(s) claims he does? Can Alice make the types of trust judgments she wants to?

In order to design systems which allow relying parties to answer such questions and make reasonable trust judgments, PKI designers need tools which can accurately evaluate

the correctness of their designs and clearly illustrate what types of trust judgments their systems enable. Since such decisions are complex and the cost of a mistake is high (i.e., Alice might trust Bob when she should not), the tools best suited to the job are formal methods. The literature contains a number of approaches for applying formal methods to the PKI problem (e.g., [42, 58, 61, 76, 116]). The modeling work of Ueli Maurer [76] stands out, as it is simple and flexible.

We wish to stress that we are concerned with a superset of the problems found in *Trust Management* (TM) systems, such as KeyNote [7, 8], PolicyMaker [65, 66], and SDSI/SPKI [22]. TM systems are primarily concerned with allowing an relying party (called an *authorizer* in the TM context) to answer a the *proof-of-compliance* question: "Do Alice's credentials allow her to access my resource?" TM systems are primarily concerned with distributed authorization, and as we will discuss in Section 8.3, our approach is applicable to a wide range of systems and applications.

As discussed throughout this thesis, we are primarily concerned with designing, building, and deploying systems that allow relying parties to make reasonable trust judgments. We have applied Maurer's calculus to model some of the systems we have seen in the wild as well as systems we have built in our lab. The real world is messy; repeatedly, we find that the calculus cannot model some of the concepts we see in practice.

- Usually, what matters about a public key is not some innate "authenticity" of it, but whether the keyholder has the properties to which the certificate attests. A calculus needs to address this.

- Certificates carry more than names; they carry extensions, use policies, attributes, etc. Some types of certificates (e.g., X.509 Attribute Certificates [30]) bind a key to a set of properties, and other types (e.g., SDSI/SPKI [22]) do not require names at all. In many real-world PKI applications, a globally unique name is not even the relevant

parameter [19, 21]. A calculus needs to address this.

- Certificates and beliefs expire and/or get revoked. Some systems use short certificate lifespans as a security advantage (e.g., SHEMP via short-lived Proxy Certificates [126, 133]). Systems which use multiple certificates to describe an entity can have lifespan mismatches. For example, an Attribute Certificate that contains the courses Alice is enrolled in this term may expire well before her Identity Certificate. A calculus needs to address this.

- Some systems allow users to delegate some or all of their authority to other users. For example, the Greenpass [34] system uses delegation to give guests access to the campus-wide wireless network. Another example involves our department delegating authority to a third party to process graduate school applications online. (Letter-writers cannot easily determine the validity of this delegation from server-side SSL.) As a final example, SHEMP relies on delegation to expand the size of the TCB, as needed. A calculus needs to address this.

- Some systems use a combination of multiple certificate types. SHEMP, as well as the Grid community's MyProxy [88] system, uses X.509 certificates in conjunction with short-lived Proxy Certificates [126, 133] for authentication and dynamic delegation. Greenpass uses an X.509 certificate in conjunction with a SDSI/SPKI certificate to express delegation. A calculus needs to address this.

- Many federated PKI systems (such as the Federal Bridge Certification Authority and the Higher Education Bridge Certification Authority) involve multiple entities issuing multiple statements about the trustworthiness of multiple users. The result is that relying parties may have multiple, possibly conflicting, statements about an entity. A calculus needs to address this.

Rather than start with a calculus and attempt to make all of the PKIs we see fit into the calculus, we start with the things we have seen, and rework Maurer's calculus to allow us to reason about all of them. Ultimately, our new calculus will provide a tool for evaluating how well SHEMP allows relying parties to make reasonable trust judgements.

Chapter Outline We begin by reviewing Maurer's calculus in Section 8.1. Then, we extend the calculus in Section 8.2 in order to make it usable for evaluating real-world PKIs. Section 8.3 uses the extended calculus to reason about a number of real-world PKI scenarios. In Section 8.4, we apply the extended calculus to SHEMP in order to show that SHEMP meets the last criterion in Chapter 3—i.e., SHEMP allows relying parties to make reasonable trust judgments. Finally, Section 8.5 summarizes and concludes this chapter.

8.1 Maurer's Calculus

In 1996, Ueli Maurer published a paper entitled "Modelling a Public-Key Infrastructure" [76] which presented two methods of modeling a PKI: one using a deterministic model and the other using a probabilistic one. We are only interested in his deterministic model (outlined in Section 3 of his paper), and our calculus is based on the deterministic model. Maurer's probabilistic model assigns confidence values to statements and allows a relying party to deduce the likelihood that a particular statement is true. The probabilistic model assumes that there is some function for assigning probabilities to statements in such a way that the probabilities match the user's beliefs. With our emphasis on creating a model which is suited for the real world, we dismissed this approach on the grounds that such a probability assigning function would be nearly impossible to create. As with Maurer's deterministic model, our model can be viewed as a probabiltistic one with probabilities set to zero and one.

This section gives a brief overview of Maurer's deterministic model and demonstrates how the model is used by working two examples. All of the information in this section (including the examples) can be found in Section 3 of Maurer's paper.

8.1.1 Maurer's Deterministic Model

According to Maurer, in any PKI, a relying party Alice can use a certificate issued by Certification Authority (CA) X for a user Bob if and only if the following conditions are met:

1. Alice knows the public key for X and believes that it is *authentic*.

2. Alice *trusts* X to be honest and to correctly authenticate the owner of a public key before signing it.

Informally, for Alice to trust any certificates issued by X, she must have a copy of X's public key, believe that it really does belong to X, and she must believe that X's certificate issuance policies are sufficient.

Maurer's deterministic model is a special type of logic (i.e., a calculus) which can be used to model a PKI and determine whether Alice can deduce the above two conditions. The calculus contains four types of propositions (called *statements* in this context) and two inference rules for deriving statements from sets of statements. The axioms are sets of statements which Alice believes to be true, and the initial set of Alice's axioms is called Alice's *initial view*. Alice can use her initial view and the inference rules to derive new statements, and these new statements along with Alice's initial view are defined as Alice's *derived view*. Maurer defines a *valid statement* as one that is contained in Alice's derived view (i.e., it is derivable from her initial view), and defines an *invalid statement* as one which cannot be derived from Alice's initial view.

Maurer introduces two concepts which we found unnatural, and thus worth clarifying. First is the concept of a *recommendation*, which is Maurer's instrument for transferring trust in a PKI. Recommendations are similar to certificates, except that they grant the power to issue certificates and/or further recommendations, and may contain private information. For example, if entity X has issued a recommendation to entity Y, then X is stating that it believes Y is trustworthy enough to issue certificates and further recommendations. Second, Maurer uses a *trust level parameter* to limit the length of certificate chains. For instance, if Alice trusts X at level 3, then she will accept certificate chains with a maximum length of 3. If Alice trusts X at level 1, then she only trusts X to issue certificates directly to users (i.e., certificate chains of length 1).

In addition to the calculus, Maurer also presents a graphical notation which is useful for representing a PKI. The graphical notation coincides with the statement definition, so that once a given PKI is represented with the graphical notation, it is easy to determine which statements can be made about the PKI.

Maurer's deterministic model is made up of the following two definitions:

Definition 1. *Statements* are one of the following forms:

- *Authenticity of public keys.* $Aut_{A,X}$ denotes Alice's belief that a particular public key P_X is authentic (i.e., belongs to entity X) and is represented graphically as an edge from A to X: $A \longrightarrow X$.

- *Trust.* $Trust_{A,X,1}$ denotes Alice's belief that a particular entity X is trustworthy for issuing certificates. Similarly, her belief that X is trustworthy for issuing recommendations of level $i-1$ is denoted by $Trust_{A,X,i}$.

  The symbol is a dashed edge from A to X labeled with i: $A \overset{i}{\dashrightarrow} X$.

- *Certificates.* $\mathit{Cert}_{X,Y}$ denotes the fact that Alice holds a certificate for Y's public key (allegedly) issued and signed by entity X. The symbol is an edge from X to Y: $X \longrightarrow Y$.

- *Recommendations.* $Rec_{X,Y,i}$ denotes the fact that Alice holds a recommendation of level i for entity Y (allegedly) issued and signed by entity X.

  The symbol is a dashed edge from X to Y labeled with i: $X \overset{i}{\dashrightarrow} Y$.

Alice's *initial view*, denoted View_A, is a set of statements. In order for Alice to use Bob's certificate, she must be able to derive the statement $\mathit{Aut}_{A,B}$, meaning that Alice believes the public key contained in Bob's certificate (i.e., P_B) actually belongs to Bob.

Definition 2. A statement is *valid* if and only if it is either contained in View_A or if it can be derived from View_A by applications of the following two inference rules:

$$\forall X, Y \quad : \quad \mathit{Aut}_{A,X},\ \mathit{Trust}_{A,X,1},\ \mathit{Cert}_{X,Y} \ \vdash\ \mathit{Aut}_{A,Y} \qquad (1)$$

$$\forall X, Y, i \geq 1 \quad : \quad \mathit{Aut}_{A,X},\ \mathit{Trust}_{A,X,i+1},\ Rec_{X,Y,i} \ \vdash\ \mathit{Trust}_{A,Y,i} \quad (2)$$

For a finite set S of statements, $\overline{S}$ denotes the closure of S under applications of the inference rules (1) and (2), i.e., the set of statements derivable from S. Alice's *derived view* is the set $\overline{\mathit{View}_A}$ of statements derivable from her initial view View_A. A statement s is valid if and only if $s \in \overline{\mathit{View}_A}$, and *invalid* otherwise.

Maurer's model assumes that trust and recommendations of level i imply trust and recommendations of lower levels, i.e.,

$$\forall X, Y, 1 \leq k < i \quad : \quad Trust_{A,X,i} \ \vdash\ Trust_{A,X,k} \quad (3)$$

and

$$\forall X, Y, 1 \leq k < i \quad : \quad Rec_{X,Y,i} \ \vdash\ Rec_{X,Y,k} \quad (4)$$

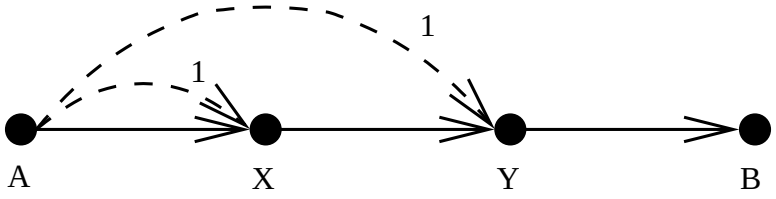

Figure 8.1: A simple PKI.

8.1.2 Examples

Working through some simple examples will make the definitions concrete and demonstrate how they work together to solve a problem. The examples come from Section 3 in Maurer's paper (there are two additional examples in Maurer's paper).

Example 1. Consider the PKI depicted in Figure 8.1. The figure indicates that Alice (A) believes her copy of X's public key is authentic (depicted by the solid edge from A to X). She trusts X and Y to issue certificates (the dashed edges), and her view contains two certificates: one from X to Y, and one from Y to B. In order for Alice to be able to use Bob's (B) certificate, she needs to derive the statement $Aut_{A,B}$. Alice's initial view is the following set of statements:

$$View_A = \{Aut_{A,X},\ Trust_{A,X,1},\ Trust_{A,Y,1},\ Cert_{X,Y},\ Cert_{Y,B}\}$$

and the statement $Aut_{A,B}$ can be derived by two applications of rule (1):

$$\begin{aligned} Aut_{A,X},\ Trust_{A,X,1},\ Cert_{X,Y} \quad &\vdash \quad Aut_{A,Y} \\ Aut_{A,Y},\ Trust_{A,Y,1},\ Cert_{Y,B} \quad &\vdash \quad Aut_{A,B}\ . \end{aligned}$$

Thus, Alice's derived view is given by:

$$\overline{View_A} = View_A \cup \{Aut_{A,Y},\ Aut_{A,B}\}\ .$$

Alice can use Bob's certificate because she believes that it is authentic, i.e., the statement $Aut_{A,B}$ is valid since $Aut_{A,B} \in \overline{View_A}$.

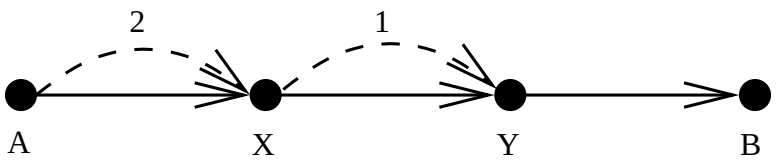

Figure 8.2: Another simple PKI.

Example 2. Now consider the PKI of Figure 8.2. The PKI is similar to that of Figure 8.1, except that Alice does not trust Y directly. She also trusts X of level 2, meaning that she trusts X to issue recommendations as well as certificates. As before, in order for Alice to be able to use Bob's (B) certificate, she needs to derive the statement $Aut_{A,B}$. In this scenario, Alice's initial view is the following set of statements:

$$View_A = \{Aut_{A,X},\ Trust_{A,X,2},\ Rec_{X,Y,1},\ Cert_{X,Y},\ Cert_{Y,B}\}\ .$$

Since Alice does not trust Y, she must derive it using rule (2):

$$Aut_{A,X},\ Trust_{A,X,2},\ Rec_{X,Y,1} \quad \vdash \quad Trust_{A,Y,1}\ .$$

Once she trusts Y, she can use rule (1) as before to establish the authenticity of Bob's certificate:

$$\begin{aligned} Aut_{A,X},\ Trust_{A,X,1},\ Cert_{X,Y} \quad &\vdash \quad Aut_{A,Y} \\ Aut_{A,Y},\ Trust_{A,Y,1},\ Cert_{Y,B} \quad &\vdash \quad Aut_{A,B}\ . \end{aligned}$$

Thus, Alice's derived view is given by:

$$\overline{View_A} = View_A \cup \{Trust_{A,Y,1},\ Aut_{A,Y},\ Aut_{A,B}\}\ .$$

As before, Alice can use Bob's certificate because she believes that it is authentic, i.e., the statement $Aut_{A,B}$ is valid since $Aut_{A,B} \in \overline{View_A}$.

8.1.3 Where Maurer's Model Breaks Down

Maurer's deterministic model is appealing because it is simple and flexible. However, when we apply the model to the types of systems we deal with in practice, we discover the limits of its applicability.

Authenticity Maurer's "Authenticity of public keys" is the wrong concept. In practice, we find that a relying party does not care about some innate "authenticity" of a public key. Rather, a relying party cares about the authenticity of the binding between the public key and the information in the certificate. If Alice is reasoning about Bob's certificate, she wants to determine if the entity holding the corresponding private key (i.e., Bob) is the same entity described by the information in the certificate. Often times, the portion of the certificate information that defines the subject's name is not what Alice cares about. She may want to determine the authenticity of other certificate information, such as key usage policies, constraints, or other extensions. Sometimes, the subject's name is of no interest to Alice at all; she may only care about the subject's role or some other attributes expressed in the certificate. Maurer's calculus just binds a key to some entity Bob, but to quote Joan Feigenbaum, Alice must still answer the question "OK, so who is Bob?".

Time In real-world PKIs, certificates expire, beliefs expire, and certificates get revoked. Without any concept of time, Maurer's model makes it impossible for relying parties to take such events into consideration when making trust decisions.

Delegation Sometimes, Bob would like to give another party the right to claim some of the attributes in his certificate. For instance, Bob may want to let Charlie claim to be Bob, so that Charlie can act as Bob. Diane may want to issue a certificate to Frank which indicates he is one of her teaching assistants. A University CA may want to let the Art

Department certify Art students. Maurer's recommendations are not enough.

Verification Maurer claims that certificates and recommendations are *allegedly* issued by an entity, and then in a footnote states: "We use the word 'alleged' because without verification, there exists no evidence that the certificate was indeed issued by the claimed entity" [76]. That verification is outside the scope of the calculus. However, verification—including the various contending approaches to checking revocation and expiration—is an important and messy part of real world PKI [16, 41, 56, 68, 82]. To illustrate the point, assume that a relying party Alice has the following initial view:

$$\mathit{View}_A = \{\mathit{Aut}_{A,X},\ \mathit{Trust}_{A,X,1},\ \mathit{Cert}_{X,Y}\}\ .$$

Now, assume that $\mathit{Cert}_{X,Y}$ is invalid for some reason. Perhaps X's signature cannot be verified, or the certificate has been revoked or has expired, or the certificate format forbids the certificate (e.g., it is an X.509 Attribute Certificate that was issued by a party which is not an Attribute Authority). Under Maurer's calculus, if Alice applies rule (1), she can derive the authenticity of Y's public key:

$$\mathit{Aut}_{A,X},\ \mathit{Trust}_{A,X,1},\ \mathit{Cert}_{X,Y} \quad \vdash \quad \mathit{Aut}_{A,Y}$$

and Alice can use the invalid certificate because $\mathit{Aut}_{A,Y} \in \overline{\mathit{View}_A}$ even though she should not.

8.2 A Model for the Real World

The shortcomings of Maurer's calculus, together with our desire to build real PKI systems and our need to reason about our designs, led us to rework Maurer's calculus so that it may be applicable to systems such as SHEMP. Our revised model is rooted in Maurer's

deterministic model, but extends it in order to deal with the complexity of real-world PKIs. From a high level, our extensions involve several elements:

1. We generalize Maurer's *Authenticity of public keys* to capture the notion of the authenticity of the binding between a public key and the certificate information.

2. We add the concept of time to Maurer's calculus so that we can model expiration and revocation.

3. We replace Maurer's *Recommendation* with a *Trust Transfer* which allows an entity A to give entity B the right to claim some or all of A's certificate information. This replacement allows us to remove the non-intuitive trust level parameter from the calculus, and to explicitly handle the various forms of trust transfer that occur in real-world PKI.

4. We introduce the notion of *validity templates* which are used to capture format-specific definitions of a statement's validity.

5. We redefine the inference rules to utilize these extensions.

(We also change the notation to use postfix instead of subscripts for the arguments, to improve readability.)

8.2.1 Our Model

Before we formally define our model, we introduce three concepts used in the model.

We use two concepts to make Maurer's deterministic model time-aware: lifespan and activity. The *lifespan* of a statement s is the time interval from t_j to t_k on which s can be used in trust calculations. We denote lifespans as the interval $\mathcal{I}$ where $\mathcal{I}$ is the time interval

$[t_j, t_k]$.[1] If at time t, $t > t_k$, we say that s has *expired*, and is no longer usable in trust calculations. We say statement s is *active* at time t if and only if $t \in \mathcal{I}$ where $\mathcal{I}$ is the lifespan of s. (In theory, we could add two levels of time: the time period during which the assertion is true, and the time period during which a party may believe and use this assertion. However, we found that the scenarios we considered, the simpler approach of one level sufficed.)

We use the concept of a *domain* to indicate the set of properties that a certificate issuing entity may assign to its subjects. Intuitively, the domain of an entity is what it is allowed to vouch for. For example, the Dartmouth College CA can bind names, Dartmouth-specific attributes, and other extensions to public keys. Thus, the Dartmouth CA's domain (denoted as the set $\mathcal{D}$) is the set of names, Dartmouth-specific attributes, and extensions it can bind to public keys. The Dartmouth CA can not bind Department of Defense (DoD)-specific attributes to public keys because it is not authorized to vouch for the DoD—i.e., the DoD-specific attributes are not in $\mathcal{D}$. It is also worth noting that if Alice trusts the Dartmouth College CA to vouch for domain $\mathcal{D}$, then Alice trusts the CA to vouch for a subset of $\mathcal{D}$. Additionally, if Alice trusts the CA to vouch for domains $\mathcal{D}_0$ and $\mathcal{D}_1$, then Alice trusts the CA to vouch for the union of the domains $\mathcal{D}_0 \cup \mathcal{D}_1$.

With these three concepts, we can formally define our model with the following two definitions.

Definition 3. In our model, statements and their representations are one of the following forms:

- **Authenticity of binding.** *Aut*$(A, X, \mathcal{P}, \mathcal{I})$ denotes A's belief that, during the interval $\mathcal{I}$, entity X (i.e., the entity holding the private key K_X) has the properties defined by the set $\mathcal{P}$.

[1] Open intervals can also be used, e.g., $(t_j, t_k]$, (t_j, t_k), $[t_j, t_k)$.

The symbol is an edge from A to X labeled with $\mathcal{P}, \mathcal{I}$: $A \xrightarrow{\mathcal{P},\mathcal{I}} X$.

- **Trust.** *Trust*$(A, X, \mathcal{D}, \mathcal{I})$ denotes A's belief that, during the interval $\mathcal{I}$, entity X is trustworthy for issuing certificates and trust transfers over domain $\mathcal{D}$.

  The symbol is a dashed edge from A to X labeled $\mathcal{D}, \mathcal{I}$: $A \overset{\mathcal{D},\mathcal{I}}{\dashrightarrow} X$.

- **Certificates.** *Cert*$(X, Y, \mathcal{P}, \mathcal{I})$ denotes the fact that X has issued a certificate to Y which, during the interval $\mathcal{I}$, binds Y's public key to the set of properties $\mathcal{P}$.

  The symbol is an edge from X to Y labeled with $\mathcal{P}, \mathcal{I}$: $X \xrightarrow{\mathcal{P},\mathcal{I}} Y$.

- **Trust Transfers.** *Tran*$(X, Y, \mathcal{P}, \mathcal{I})$ denotes that A holds a trust transfer issued by X which, during the interval $\mathcal{I}$, binds Y's public key to the set of properties $\mathcal{P}$.

  The symbol is a dashed edge from X to Y labeled with $\mathcal{P}, \mathcal{I}$: $X \overset{\mathcal{P},\mathcal{I}}{\dashrightarrow} Y$.

- **Certificate Validity Templates.** *Valid*$\langle A, C, t\rangle$ denotes A's belief that certificate C is valid at time t according to the definition of validity appropriate for C's format. Minimally, the issuer's signature over C must be verified and C must be active.

- **Transfer Validity Templates.** *Valid*$\langle A, T, t\rangle$ denotes A's belief that trust transfer T is valid at time t according to the definition of validity appropriate for T's format. Minimally, the issuer's signature over T must be verified and T must be active.

We introduce the notion of *templates* because different PKI approaches have different and non-trivial ways of expressing validity of certificates and transfers. For example, validity for X.509 identity certificates may be determined by expiration dates and the absence of the certificate on a currently valid Certificate Revocation List (CRL); validity of transfer in an X.509 identity certificate may be determined by `basicConstraints` and usage bits in a certificate held by the source party. (The statements will be discussed in depth in Section 8.2.2).

Alice's initial view is denoted View_A, as before. Under the model, if Alice wishes to verify that Bob had some property p at time t, she must be able to derive the statement $\mathit{Aut}(A, B, \mathcal{P}, \mathcal{I})$ where $t \in \mathcal{I}$ and $p \in \mathcal{P}$. In many cases, the evaluation time t is the current time, meaning that Alice wants to verify that Bob currently has some property p. It should be noted, however, that the model still functions if the evaluation time t is some time in the past or the future. Such scenarios will be examined more closely in Section 8.3.

Definition 4. In our model, a statement is valid if and only if it is either contained in View_A or if it can be derived from View_A by applications of the following two inference rules:

$$\forall X, Y, t \in \{\mathcal{I}_0 \cap \mathcal{I}_1\}, \mathcal{Q} \subseteq \mathcal{D} : \tag{5}$$
$$\mathit{Aut}(A, X, \mathcal{P}, \mathcal{I}_0),\ \mathit{Trust}(A, X, \mathcal{D}, \mathcal{I}_1),\ \mathit{Valid}\langle A, \mathit{Cert}(X, Y, \mathcal{Q}, \mathcal{I}_2), t\rangle\ \vdash\ \mathit{Aut}(A, Y, \mathcal{Q}, \mathcal{I}_2)$$

$$\forall X, Y, t \in \{\mathcal{I}_0 \cap \mathcal{I}_1\}, \mathcal{Q} \subseteq \mathcal{D} : \tag{6}$$
$$\mathit{Aut}(A, X, \mathcal{P}, \mathcal{I}_0),\ \mathit{Trust}(A, X, \mathcal{D}, \mathcal{I}_1),\ \mathit{Valid}\langle A, \mathit{Tran}(X, Y, \mathcal{Q}, \mathcal{I}_2), t\rangle \vdash \mathit{Trust}(A, Y, \mathcal{Q}, \mathcal{I}_2)$$

As with Maurer's deterministic model, for a finite set S of statements, $\overline{S}$ denotes the closure of S under applications of the inference rules (5) and (6), i.e., the set of statements derivable from S. The *evaluation time* t is the time that Alice is attempting to reason about. Alice's *derived view at evaluation time* t is the set of statements derivable from her initial view at evaluation time t. Alice's derived view is defined by the function $\overline{\mathit{View}_A(t)}$ where $\overline{\mathit{View}_A} : t \longrightarrow \overline{S}$. Under the model, a statement s is *valid at evaluation time* t if and only if $s \in \overline{\mathit{View}_A(t)}$, and invalid otherwise.

8.2.2 Semantic Sugar

The statements of our model (Definition 3) are intended to be more general than the statements of Maurer's deterministic model. This section explains the intuition for the definitions of the model, and highlights some of the semantic difference between our model and Maurer's.

Authenticity of binding Maurer's notion of "authenticity" establishes that entity X holds the private key corresponding to the public key in X's certificate. We extend authenticity to establish that some entity X not only holds the private key corresponding to the public key in X's certificate, but also has the other properties $\mathcal{P}$ contained in the certificate. As noted earlier, these properties may include attributes, roles, key attributes, or any other certificate extensions such as Proxy Certificate Information extensions, extended key usage, etc.

Trust Maurer's definition of trust limits trust vertically (i.e., how deep trust may propagate) via the level parameter. Our definition of trust limits trust horizontally (i.e., how wide the trust may span) via the domain concept. A trusted entity should only be allowed to vouch (either via a certificates or trust transfer statement) for properties that it is authorized to speak for. Again, the Dartmouth CA should not be allowed to assign DoD-specific attributes to members of the Dartmouth community. Similarly, Alice should not be able to delegate to Bob some property she does not possess or is not allowed to delegate. Entities may be allowed to vouch for a specific domain for a number of reasons.

- In many cases, the assignment of a domain to a trusted entity is done out-of-band of the PKI, and the users trust the entity a priori. Such a scenario is often the case with an organization's CA—users almost always trust their CA to vouch for the organization's population. In our calculus, this fact is represented by the inclusion of the CA's authenticity and trust statements in every user's initial view, i.e.,

$$\forall u \in U : \textit{View}_u = \{\textit{Aut}(u, CA, \mathcal{P}, \mathcal{I}),\ \textit{Trust}(u, CA, \mathcal{D}, \mathcal{I})\}\ .$$

where U is the set of users in the organization, CA is the organization's CA, and $\mathcal{D}$ contains the ability to vouch for all $u \in U$. This is not a requirement of the calculus, but often occurs in practice.

- In other cases, the assignment of a domain to a trusted entity is done implicitly. Delegation scenarios are an example of this type of binding (such a scenario will be explored further in Section 8.3). Typically, if Alice trusts Bob to delegate some of his privileges to another entity, Alice would require Bob to have had the privilege in the first place. In the model, this is represented by Bob's trust statement having a subset of the properties in his authenticity statement, i.e. for $\mathcal{Q} \subseteq \mathcal{P}$:

$$\textit{View}_A = \{\textit{Aut}(A, B, \mathcal{P}, \mathcal{I}),\ \textit{Trust}(A, B, \mathcal{Q}, \mathcal{I})\}\ .$$

Certificates In Maurer's deterministic model, certificates bind a name to public key. Our notion of certificate not only binds a name to the public key, but also the rest of the certificate information. Real certificates are more complex than a name and public key. They contain extensions, attributes, roles, usage policies, validity intervals, etc. and relying parties often need this information to make reasonable trust decisions.

Trust Transfers Maurer's recommendation is used to explicitly transfer trust in a PKI. It is much like a certificate, although it can be transferred further, and may contain private information. Our notion of a trust transfer is similar, although more generally applicable. Our trust transfer statement can be used to model different types of transactions such as when a CA names certifies a subordinate CA, or when Alice delegates some or all of her properties to Bob. Moreover, a trust transfer may be an explicit statement (such as a certifi-

cate) or an implicit statement (e.g., by activating a certificate extension such as the X.509 `basicConstraints` extension).

Validity Templates As discussed in Section 8.1.3, Maurer's deterministic model does not include checking for certificate validity as part of the calculus. However, it would be impossible to allow for such checking for every type of certificate format, and still have a calculus which is usable. Every time a new certificate format were released, the calculus would have to be expanded in order to reason about the new format. The way our model deals with this is through the use of validity templates: a meta-statement whose validity checking algorithm depends on the argument type. Templates allow us to reason about different certificate formats without having to handle every format's specifics. As stated in Definition 3, for the *Valid*$\langle\rangle$ template to be true, a certificate or transfer should minimally have a verifiable signature and be active. Additionally, the *Valid*$\langle\rangle$ template should evaluate the format-specific validity rules.

For example, assume that *Valid*$\langle A, C, t\rangle$ is being evaluated, and C is an X.509 Identity Certificate. In order for *Valid*$\langle A, C, t\rangle$ to be true, the template instantiation should check that C's signature verifies, that C has not expired, that C has not been revoked (e.g., by having in one's belief set a properly signed, active copy of the Certificate Revocation List (CRL) to which C points), that C's key attributes allow the requested operation, that the certificate chain length has not been exceeded, etc. If C were an X.509 Attribute Certificate, *Valid*$\langle A, C, t\rangle$ may also check that C was signed by an attribute authority.

Evaluating trust transfer statements is very similar. For example, if *Valid*$\langle A, T, t\rangle$ is being evaluated, and T is a trust transfer expressed implicitly in an X.509 certificate (i.e., by the `basicConstraints` extension being set to true), then the template instantiation should check that T's signature verifies, that T has not expired, that T has not been revoked, that the `basicConstraints` extension is set to true, and that the value of the

`pathLenConstraint` extension has not been exceeded.

Inference Rules The first inference rule of the new model (rule (5)) is similar to the first rule of Maurer's deterministic model (rule (1)). They are both used to derive authenticity statements. Rule (5) states how Alice may deduce the authenticity of the binding expressed in Y's certificate:

- She believes entity X is bound to the set of properties $\mathcal{P}$ for some time interval $\mathcal{I}_0$.
- She trusts X to issue certificates for the domain $\mathcal{D}$ during the time interval $\mathcal{I}_1$.
- She has a copy of a valid certificate issued from X to Y which binds Y's public key to a set of properties $\mathcal{Q}$ for some time interval $\mathcal{I}_2$.
- She believes X is allowed to speak for the set of properties in the certificate (i.e., $\mathcal{Q}$ is in X's domain).
- Her beliefs about X are active (i.e., Alice's evaluation time $t \in \{\mathcal{I}_0 \cap \mathcal{I}_1\}$).

The second inference rule of the new model (rule (6)) is similar to the second rule of the deterministic model (rule (2)). Both rules are used to transfer trust between entities. Rule (6) states how Alice may deduce the trustworthiness of Y:

- She believes entity X is bound to the set of properties $\mathcal{P}$ for some time interval $\mathcal{I}_0$.
- She trusts X to issue trust transfer statements for domain $\mathcal{D}$ during time interval $\mathcal{I}_1$.
- She has a valid trust transfer statement from X to Y which binds Y's public key to a set of properties $\mathcal{Q}$ for some time interval $\mathcal{I}_2$.
- She believes that X is allowed to speak for the set of properties in the certificate (i.e., $\mathcal{Q}$ is a subset of X's domain).

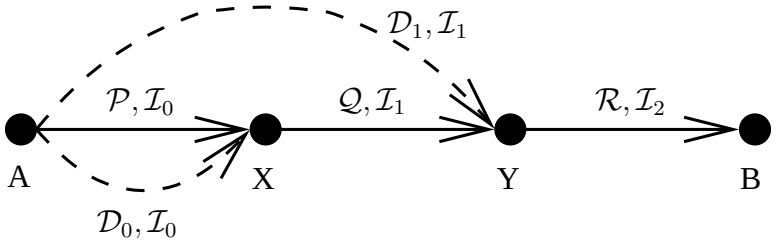

Figure 8.3: Statement graph for a simple PKI.

- Her beliefs about X are active (i.e., evaluation time $t \in \{\mathcal{I}_0 \cap \mathcal{I}_1\}$).

Note that the level parameter of Maurer's deterministic model has been omitted in our model. The use of validity templates allows relying parties to directly check the properties in certificates and trust transfer statements for things like certificate chain length, delegation depth, `pathLenConstraint`, etc. This way, relying parties may draw such conclusions using the context of the certificate format rather than an artificial level parameter. However, our replacement makes it difficult to map all of Maurer's model into ours. We find this to be acceptable since the level parameter is an artificial construct anyhow, and is not an inherent part of PKI in general.

8.2.3 Examples

Working some examples will illustrate the basic concepts of our model. The examples are based on the ones from Section 8.1.2 (and Section 3 in Maurer's paper), although we extend them to illustrate the new features of our model.

Example 3. Consider the PKI depicted in Figure 8.3. The figure indicates that Alice (A) believes that X's public key is bound to the set of properties $\mathcal{P}$ during the interval $\mathcal{I}_0$ (depicted by the solid edge from A to X). She trusts X and Y to issue certificates (the dashed edges) over domains $\mathcal{D}_0$ and $\mathcal{D}_1$ respectively. (For simplicity, we set the lifespans of the trust statements to match the authenticity and certificate statements, but this is not

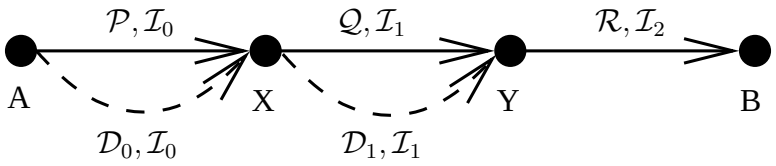

Figure 8.4: Statement graph for another simple PKI.

necessary.) Her view also contains two certificates: one from X to Y which binds Y's public key to the set of properties $\mathcal{Q}$ during the interval $\mathcal{I}_1$, and one from Y to B which binds B's public key to the set of properties $\mathcal{R}$ during the interval $\mathcal{I}_2$. In order for Alice to be able to use Bob's (B) certificate (either the public key or the properties in $\mathcal{R}$) at evaluation time t, she needs to derive the statement $\mathit{Aut}(A, B, \mathcal{R}, \mathcal{I}_2)$. Alice's initial view is the following set of statements:

$$\mathit{View}_A = \left\{ \begin{array}{lll} \mathit{Aut}(A, X, \mathcal{P}, \mathcal{I}_0), & \mathit{Trust}(A, X, \mathcal{D}_0, \mathcal{I}_0), & \mathit{Trust}(A, Y, \mathcal{D}_1, \mathcal{I}_1), \\ \mathit{Cert}(X, Y, \mathcal{Q}, \mathcal{I}_1), & \mathit{Cert}(Y, B, \mathcal{R}, \mathcal{I}_2) & \end{array} \right\} .$$

Suppose that evaluation time $t \in \{\mathcal{I}_0 \cap \mathcal{I}_1\}$, $\mathcal{Q} \subseteq \mathcal{D}_0$, and $\mathcal{R} \subseteq \mathcal{D}_1$. The statement $\mathit{Aut}(A, B, \mathcal{R}, \mathcal{I}_2)$ can be derived by two applications of rule (5):

$$\mathit{Aut}(A, X, \mathcal{P}, \mathcal{I}_0),\ \mathit{Trust}(A, X, \mathcal{D}_0, \mathcal{I}_0),\ \mathit{Valid}\langle A, \mathit{Cert}(X, Y, \mathcal{Q}, \mathcal{I}_1), t\rangle \vdash \mathit{Aut}(A, Y, \mathcal{Q}, \mathcal{I}_1)$$
$$\mathit{Aut}(A, Y, \mathcal{Q}, \mathcal{I}_1),\ \mathit{Trust}(A, Y, \mathcal{D}_1, \mathcal{I}_1),\ \mathit{Valid}\langle A, \mathit{Cert}(Y, B, \mathcal{R}, \mathcal{I}_2), t\rangle \vdash \mathit{Aut}(A, B, \mathcal{R}, \mathcal{I}_2).$$

Thus, Alice's derived view at evaluation time t is given by:

$$\overline{\mathit{View}_A(t)} = \mathit{View}_A \cup \{\mathit{Aut}(A, Y, \mathcal{Q}, \mathcal{I}_1),\ \mathit{Aut}(A, B, \mathcal{R}, \mathcal{I}_2)\} .$$

Alice believes that the binding between Bob's public key and the set of properties $\mathcal{R}$ in his certificate is authentic during the time interval $\mathcal{I}_2$. Alice may stop believing this fact when Bob's certificate expires or gets revoked. (Such events give rise to *nonmonotonicity* [61], which will be discussed in Section 8.5.)

Example 4. Consider the PKI depicted in Figure 8.4. The figure indicates that Alice (A) believes that X's public key is bound to the set of properties $\mathcal{P}$ during the interval $\mathcal{I}_0$ (depicted by the solid edge from A to X). She trusts X to issue certificates over domain $\mathcal{D}_0$ (the dashed edge). (For simplicity, we set the lifespans of the trust statements to match the authenticity and certificate statements, but this is not necessary.) Her view contains a trust transfer from X to Y (the dashed edge), and two certificates (solid edges): one from X to Y which binds Y's public key to the set of properties $\mathcal{Q}$ during the interval $\mathcal{I}_1$, and one from Y to B which binds B's public key to the set of properties $\mathcal{R}$ during the interval $\mathcal{I}_2$. The trust transfer from X to Y could be an explicit statement issued by X indicating that it trusts Y to issue certificates; in practice, it is more likely to be expressed implicitly in the certificate issued from X to Y (e.g., by X setting the `basicConstraints` field of Y's X.509 certificate or the "delegation" flag of Y's SDSI/SPKI certificate).

In order for Alice to be able to believe Bob's (B) certificate (either the public key or the properties in $\mathcal{R}$) at evaluation time t, she needs to derive the statement $\mathit{Aut}(A, B, \mathcal{R}, \mathcal{I}_2)$. In this scenario, Alice's initial view is the following set of statements:

$$\mathit{View}_A = \left\{ \begin{array}{lll} \mathit{Aut}(A, X, \mathcal{P}, \mathcal{I}_0), & \mathit{Trust}(A, X, \mathcal{D}_0, \mathcal{I}_0), & \mathit{Tran}(X, Y, \mathcal{D}_1, \mathcal{I}_1), \\ \mathit{Cert}(X, Y, \mathcal{Q}, \mathcal{I}_1), & \mathit{Cert}(Y, B, \mathcal{R}, \mathcal{I}_2) & \end{array} \right\} .$$

Since Alice does not trust Y to issue certificates directly, she must derive her trust in Y using rule (6). Suppose that evaluation time $t \in \mathcal{I}_0$ and $\mathcal{D}_1 \subseteq \mathcal{D}_0$, we have:

$$\mathit{Aut}(A, X, \mathcal{P}, \mathcal{I}_0),\ \mathit{Trust}(A, X, \mathcal{D}_0, \mathcal{I}_0),\ \mathit{Valid}\langle A, \mathit{Tran}(X, Y, \mathcal{D}_1, \mathcal{I}_1), t\rangle \vdash$$
$$\mathit{Trust}(A, Y, \mathcal{D}_1, \mathcal{I}_1) .$$

Once Alice trusts Y, she can use rule (5) to establish the authenticity of the binding expressed in Bob's certificate at evaluation time t assuming $t \in \mathcal{I}_1$, $\mathcal{Q} \subseteq \mathcal{D}_0$, and $\mathcal{R} \subseteq \mathcal{D}_1$ (in addition to our previous supposition that $t \in \mathcal{I}_0$, and $\mathcal{D}_1 \subseteq \mathcal{D}_0$):

$$\textit{Aut}(A, X, \mathcal{P}, \mathcal{I}_0),\ \textit{Trust}(A, X, \mathcal{D}_0, \mathcal{I}_0),\ \textit{Valid}\langle A, \textit{Cert}(X, Y, \mathcal{Q}, \mathcal{I}_1), t\rangle \vdash \textit{Aut}(A, Y, \mathcal{Q}, \mathcal{I}_1)$$

$$\textit{Aut}(A, Y, \mathcal{Q}, \mathcal{I}_1),\ \textit{Trust}(A, Y, \mathcal{D}_1, \mathcal{I}_1),\ \textit{Valid}\langle A, \textit{Cert}(Y, B, \mathcal{R}, \mathcal{I}_2), t\rangle \vdash \textit{Aut}(A, B, \mathcal{R}, \mathcal{I}_2)\ .$$

Thus, Alice's derived view at evaluation time t is given by:

$$\overline{\textit{View}_A(t)} = \textit{View}_A \cup \{\textit{Trust}(A, Y, \mathcal{D}_1, \mathcal{I}_1),\ \textit{Aut}(A, Y, \mathcal{Q}, \mathcal{I}_1),\ \textit{Aut}(A, B, \mathcal{R}, \mathcal{I}_2)\}\ .$$

Alice believes that the binding between Bob's public key and his certificate properties is authentic during the time interval $\mathcal{I}_2$. Alice may stop believing this fact when Bob's certificate expires or gets revoked.

8.3 Using Our New Model

The motivation for developing this new model was to give PKI designers a tool which can be used to reason about a wide range of PKI systems. To this end, the appropriate measure of success is the model's ability to capture a number of different real-world scenarios. In this section, we apply the model to an array of situations in order to illustrate its applicability.

Modeling Multiple Certificate Families The new model's certificate statement binds an entity's public key to some set of properties $\mathcal{P}$ for some lifespan $\mathcal{I}$. The power of the new model stems from the fact that it is agnostic with respect to the semantics of the properties in $\mathcal{P}$, and yet still builds a calculus which allows relying parties to reason about these sets. The semantics of the property sets vary across certificate formats.

- In the standard X.509 Identity Certificate [41], the property sets may include the subject's Distinguished Name, Alternative names, name constraints, her key attributes,

information about where to retrieve CRLs, and any number of domain-specific policies. The property set may also include information as to whether the subject is allowed to sign other certificates (i.e., via the `basicConstraints` field).

- X.509 Attribute Certificates (ACs) [30] contain a very different set of properties than X.509 Identity Certificates. ACs typically use domain-specific properties which are used by relying parties to make authorization decisions. Some common examples of attributes include: identity, group membership, role, clearance level, etc. Other differences include the fact that an AC's subject may delegate to another party the right to claim some of the delegator's attributes, and that ACs may not be used to form certificate chains.

- The X.509-based Proxy Certificate (PC) [126] is similar to an X.509 Identity Certificate, except that PCs have a Proxy Certificate Information (PCI) extension and are signed by standard X.509 Identity Certificates. The PC standard allows any type of policy statement expressed in any language (such as eXtensible Access Control Markup Language) to be placed in the PCI. Thus, the set of properties for a PC could contain a large family of policy statements.

- The SDSI/SPKI certificate format [22] takes an entirely different approach to certificates. The set of properties placed in a SDSI/SPKI certificate does not contain a global name for the subject, as SDSI/SPKI uses the public key as the subject's unique identifier. (If there is any name at all, it would be part of a linked local namespace.) Further, a SDSI/SPKI certificate contains attributes much like X.509 attributes, except they are expressed as S-expressions as opposed to ASN.1. In contrast to X.509 ACs, SDSI/SPKI certificates are allowed to be chained.

Indeed, the power of the new model is its ability to reason about all of these diverse certificate formats and semantics using only one calculus.

Modeling Revocation Validity templates play an important role in our model: they allow users to reason about different *types* of signed statements. As an example, consider the case when Alice needs to make a trust decision about Bob. The instantiation of the validity template used to check Bob's certificate may require that Alice check a CRL to ensure that Bob's certificate is not included in the list of revoked certificates.

Thus, Alice's first step is to make a trust decision about a signed CRL. CRLs contain a list of revoked certificates, a lifespan (noted by the `thisUpdate` and `nextUpdate` fields), and are signed by the organization's CA. Formally, we can represent a CRL as a kind of certificate which is issued by a CA X and contains a list of revoked certificates $\mathcal{L}$, and a lifespan $\mathcal{I}$: $\mathit{Cert}(X, \emptyset, \mathcal{L}, \mathcal{I})$. Since CRLs do not contain a public key, we use the empty set notation to indicate the absence of a key. In order for Alice to use the CRL at evaluation time t, she needs to deduce that it is authentic, i.e., $\mathit{Aut}(A, \emptyset, \mathcal{L}, \mathcal{I}) \in \overline{\mathit{View}_A(t)}$.

Assuming that Alice believes that the binding expressed in CA X's certificate is authentic, and that she trusts CA X to issue certificates over domain $\mathcal{D}$, her initial view would be:

$$\mathit{View}_A = \{\mathit{Aut}(A, X, \mathcal{P}, \mathcal{I}_0),\ \mathit{Trust}(A, X, \mathcal{D}, \mathcal{I}_0),\ \mathit{Cert}(X, \emptyset, \mathcal{L}, \mathcal{I}_1)\}\ .$$

If X is authorized to vouch for the revocation status of all the certificates in $\mathcal{L}$ (i.e., $\mathcal{L} \subseteq \mathcal{D}$), and all of the statements are active (i.e., $t \in \mathcal{I}_0$), then Alice can deduce the CRL's authenticity by applying rule (5):

$$\mathit{Aut}(A, X, \mathcal{P}, \mathcal{I}_0),\ \mathit{Trust}(A, X, \mathcal{D}, \mathcal{I}_0),\ \mathit{Valid}\langle A, \mathit{Cert}(X, \emptyset, \mathcal{L}, \mathcal{I}_1), t\rangle \vdash \mathit{Aut}(A, \emptyset, \mathcal{L}, \mathcal{I}_1).$$

For the CRL, the instantiation of the validity template $\mathit{Valid}\langle A, \mathit{Cert}(X, \emptyset, \mathcal{L}, \mathcal{I}_1), t\rangle$ must check that $t \in \mathcal{I}_1$, and that X's signature is verifiable. If the conditions are met, we

have $\mathit{Aut}(A, \emptyset, \mathcal{L}, \mathcal{I}_1) \in \overline{\mathit{View}_A(t)}$, which indicates Alice's belief that $\mathcal{L}$ accurately represents the list of revoked certificates during the interval $\mathcal{I}_1$.

Once Alice believes the CRL, she must make a trust decision about Bob's certificate: $\mathit{Cert}(X, B, \mathcal{Q}, \mathcal{I}_2)$. Assuming that CA X can vouch for Bob's certificate information (i.e., $\mathcal{Q} \subseteq \mathcal{D}$), and all of the statements are active (i.e., $t \in \mathcal{I}_0$), then Alice can deduce Bob's authenticity by applying rule (5):

$$\mathit{Aut}(A, X, \mathcal{P}, \mathcal{I}_0),\ \mathit{Trust}(A, X, \mathcal{D}, \mathcal{I}_0),\ \mathit{Valid}\langle A, \mathit{Cert}(X, B, \mathcal{Q}, \mathcal{I}_2), t\rangle \vdash \mathit{Aut}(A, B, \mathcal{Q}, \mathcal{I}_2).$$

In this case, the instantiation of the validity template $\mathit{Valid}\langle A, \mathit{Cert}(X, B, \mathcal{Q}, \mathcal{I}_2), t\rangle$ is being used to establish the validity of a certificate, not a CRL. As before, the template instantiation must check that $t \in \mathcal{I}_2$, and that X's signature over Bob's certificate verifies. However, the instantiation should also check that Bob's certificate has not been revoked, i.e., $\mathit{Cert}(X, B, \mathcal{Q}, \mathcal{I}_2) \notin \mathcal{L}$ as well as any other certificate information which is relevant to the requested operation.

Authorization-based Scenarios and Trust Management In many modern distributed systems, access to some resource is granted based on authorization rather that authentication. Systems such as PERMIS [15] use the attributes contained in ACs to determine whether an entity should have access to a resource (other Trust Management systems such as KeyNote [7, 8] and PolicyMaker [65, 66] have their own certificate formats for expressing credentials). This approach simplifies the management of Access Control Lists (ACLs) at the resource. For example, if Bob wants to access Alice's file, he presents his AC to Alice. Alice first decides if the AC is authentic, and if so, she examines Bob's attributes to check if he should have access (e.g., if the file is accessible to the group "developers", then Bob's attributes must state that he is a member of the group).

Maurer's deterministic model cannot handle this scenario, primarily because it cannot handle ACs. Under the deterministic model, if Alice were to deduce the authenticity of Bob's public key, she still has learned nothing about Bob (i.e., his attributes). All she has established is that the entity named Bob really has the private key corresponding to the public key found in the certificate.

There are a number of Trust Management (TM) languages which do handle this scenario, such as *Delegation Logic* [60] and others [62]. These TM languages can not only tell Alice that Bob has a certain set of credentials, but can also evaluate Bob's credentials and Alice's policy to determine whether Alice should allow the file access. While TM languages are typically framework-specific (i.e., KeyNote, PolicyMaker, and SDSI/SPKI have their own policy languages), there have been efforts to generalize across languages [131]. Since our model is aimed at reasoning about PKI systems, and not TM systems, such policy evaluation is outside the scope of our model's abilities. However, our model can be used to model these different certificate and credential formats (e.g., ACs, credentials, SDSI/SPKI certificates), as well as reason about the authenticity of the core trust statements.

Under our model, Bob would first present his AC to Alice (e.g., $\mathit{Cert}(X, B, \mathcal{P}, \mathcal{I})$). Assume that Alice can then derive the authenticity of the binding between Bob's public key and the properties in the certificate—i.e., $\mathit{Aut}(A, B, \mathcal{P}, \mathcal{I}) \in \overline{\mathit{View}_A(t)}$. Since Bob's certificate is an AC, Alice needs to determine if an attribute placing Bob in the "developers" group is in the set $\mathcal{P}$. If so, then Bob is allowed to access the file.

Delegation Some systems allow users to delegate some or all of their properties to another entity. Maurer's deterministic model allows users to issue recommendations and certificates to other entities, but this is insufficient to capture the notion of delegation. Maurer's model allows Alice to vouch for Bob, but she is limited to vouching for Bob's identity.

In our model, Alice can give some or all of her properties to Bob (possibly including

identity), provided she has the properties in the first place (i.e., she can only give Bob the properties in her domain). In the calculus, Alice would issue a certificate to Bob (i.e., $\mathit{Cert}(A, B, \mathcal{P}, \mathcal{I})$).

If a relying party Charlie has established that the binding between Alice's public key and her properties is authentic, trusts Alice to delegate, and receives a delegation from Alice to Bob, then his view will be:

$$\mathit{View}_C = \{\mathit{Aut}(C, A, \mathcal{P}, \mathcal{I}),\ \mathit{Trust}(C, A, \mathcal{P}, \mathcal{I}),\ \mathit{Cert}(A, B, \mathcal{P}, \mathcal{I})\}\ .$$

He can then derive the authenticity of the binding between Bob's public key and the delegated properties (i.e., the statement $\mathit{Aut}(C, B, \mathcal{P}, \mathcal{I})$) by applying rule (5):

$$\mathit{Aut}(C, A, \mathcal{P}, \mathcal{I}),\ \mathit{Trust}(C, A, \mathcal{P}, \mathcal{I}),\ \mathit{Valid}\langle C, \mathit{Cert}(A, B, \mathcal{P}, \mathcal{I}), t\rangle \ \vdash\ \mathit{Aut}(C, B, \mathcal{P}, \mathcal{I})\ .$$

Thus, we have $\mathit{Aut}(C, B, \mathcal{P}, \mathcal{I}) \in \overline{\mathit{View}_C(t)}$.

Modeling MyProxy The Grid community's MyProxy credential repository [88] uses a chain of certificates for authentication. When Bob (or some process to which Bob delegates) wants to access a resource on the Grid, he generates a temporary keypair, logs on to the MyProxy server, and requests that a Proxy Certificate (PC) [126, 133] be generated which contains the public portion of the temporary keypair and some subset of Bob's privileges. The new PC is then signed with the private portion of the keypair described by Bob's long term X.509 Identity Certificate, thus forming a chain of certificates.

The situation is depicted in Figure 8.5. Entity X is the CA which issued Bob's X.509 Identity Certificate, and T is the entity which will own the temporary keypair (possibly Bob or some other delegated entity or process). Initially, Alice believes that the binding between X's public key and properties is authentic during $\mathcal{I}_0$, and she trusts X to issue

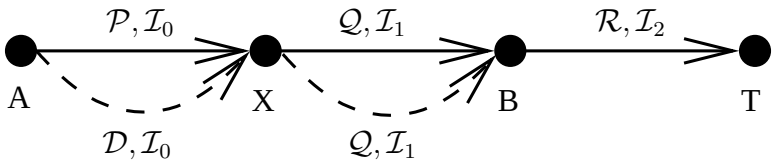

Figure 8.5: The statement graph for the MyProxy system.

certificates and trust transfers for the domain $\mathcal{D}$. X has issued Bob a certificate binding his public key to the set of properties $\mathcal{Q}$ during $\mathcal{I}_1$. X has also issued a trust transfer to Bob, so that he may use his private key to sign his PC. The trust transfer is not a separate certificate in this scenario; it is an implicit statement which X makes by setting the `basicConstraints` field in Bob's X.509 Identity Certificate which allows him to sign certificates. Finally, Bob has issued a certificate (the PC) to the entity possessing the temporary keypair T for some subset $\mathcal{R}$ of his properties. The PC is valid over the interval $\mathcal{I}_2$, which in practice, is on the order of hours. In order for Alice to accept Bob's PC, she must derive the statement $\mathit{Aut}(A, T, \mathcal{R}, \mathcal{I}_2)$. This scenario can be reduced to an instance of Example 4 in Section 8.2.3. Modeling SHEMP will be discussed in Section 8.4.1.

Discovering Requirements: Greenpass The Greenpass [34] system uses delegation to give guests inside access to a campus-wide wireless network. Further, it relies on an X.509 certificate in conjunction with SDSI/SPKI certificates to express delegation. To gain some insight as to why the designers chose this hybrid approach, we can model the problem with the calculus.

Let us assume that a relying party Alice is a member of Dartmouth College, which we denote C. Let us also assume that another member of C, named Bob, has invited his colleague George from the University of Wisconsin (denoted W) to come for a visit. Bob would like to give George some guest access to the network, so that he can access some resources protected by Alice. In order for Alice to grant access to George, she must make a trust decision about George. Since there is no trust relationship between C and W (i.e., they

are not cross-certified or participating in the Higher Education Bridge CA), Alice cannot simply reason about George based on statements made by George's CA. Since George is Bob's guest, Bob is in a position to vouch for George.

Initially, Alice's view consists of her authenticity and trust beliefs about her CA, a certificate issued by her CA to Bob, and a certificate issued by George's CA to George:

$$\mathit{View}_A = \{\mathit{Aut}(A, C, \mathcal{P}, \mathcal{I}_0),\ \mathit{Trust}(A, C, \mathcal{D}, \mathcal{I}_0),\ \mathit{Cert}(C, B, \mathcal{Q}, \mathcal{I}_1), \mathit{Cert}(W, G, \mathcal{R}, \mathcal{I}_2)\}\ .$$

Since Bob has a certificate issued by a CA which Alice trusts, she can deduce the authenticity of Bob's certificate information (assuming that $t \in \mathcal{I}_0$, $\mathcal{Q} \subseteq \mathcal{D}$, and Bob's certificate is valid), i.e.,

$$\mathit{Aut}(A, C, \mathcal{P}, \mathcal{I}_0),\ \mathit{Trust}(A, C, \mathcal{D}, \mathcal{I}_0),\ \mathit{Valid}\langle A, \mathit{Cert}(C, B, \mathcal{Q}, \mathcal{I}_1), t\rangle \vdash \mathit{Aut}(A, B, \mathcal{Q}, \mathcal{I}_1)\ .$$

Now, in order for Alice to grant George access to her resources, she needs to believe the binding between George's public key and the properties in his certificate, and then that the properties grant him authorization. However, since Alice does not trust W (and has no reason to), she has no reason to trust any of the properties about George expressed in his certificate (namely, in the set of properties $\mathcal{R}$).

Since George is Bob's guest, Bob is in a position to delegate some of his privileges to George. In order for Alice to believe this delegation, Alice first needs to believe that Bob is in a position to delegate, and she then needs to believe that Bob actually delegated to George. The first condition requires the CA to transfer trust to Bob (i.e., $\mathit{Tran}(C, B, \mathcal{Q}, \mathcal{I}_1) \in \mathit{View}_A$).[2] The second condition requires that Bob issue a certificate

[2] In the Greenpass prototype, this trust transfer is expressed as a SDSI/SPKI certifi cate issued to Bob's public key and allowing him to delegate. It could also have been implicit, by setting the `basicConstraints` fi eld of Bob's X.509 certifi cate, but this would require reissuing Bob's certifi cate.

which delegates some of his properties to George (i.e., $\mathit{Cert}(B, G, \mathcal{S}, \mathcal{I}_3) \in \mathit{View}_A$ where $\mathcal{S} \subseteq \mathcal{Q}$).

Assuming all of the preconditions are met, and the certificates and trust transfer are valid, Alice can deduce Bob's trustworthiness, and the authenticity of the certificate issued from Bob to George, i.e.,

$$\mathit{Aut}(A, C, \mathcal{P}, \mathcal{I}_0),\ \mathit{Trust}(A, C, \mathcal{D}, \mathcal{I}_0),\ \mathit{Valid}\langle A, \mathit{Tran}(C, B, \mathcal{Q}, \mathcal{I}_1), t\rangle \vdash \mathit{Trust}(A, B, \mathcal{Q}, \mathcal{I}_1)$$

$$\mathit{Aut}(A, B, \mathcal{Q}, \mathcal{I}_1),\ \mathit{Trust}(A, B, \mathcal{Q}, \mathcal{I}_1),\ \mathit{Valid}\langle A, \mathit{Cert}(B, G, \mathcal{S}, \mathcal{I}_3), t\rangle \vdash \mathit{Aut}(A, G, \mathcal{S}, \mathcal{I}_3).$$

Thus Alice can reason about George because $\mathit{Aut}(A, G, \mathcal{S}, \mathcal{I}_3) \in \overline{\mathit{View}_A(t)}$.

The last question that the system designer is faced with is: "what type of certificate format should be used for the certificate issued from Bob to George?" The first consideration is that if George already has a public key, the system should reuse it. The second consideration is that Alice is not concerned with George's identity, but rather his authorization. Finally, we need to reason about what type of certificate format would allow this type of delegation scenario, and still make $\mathit{Valid}\langle A, C, t\rangle$ evaluate to true. Proxy Certificates would not allow George to have the public key of his regular certificate (i.e., $\mathit{Cert}(W, G, \mathcal{R}, \mathcal{I}_2)$) also used in his Proxy Certificate, resulting in $\mathit{Valid}\langle A, C, t\rangle$ never being true. An X.509 Attribute Certificate would not make $\mathit{Valid}\langle A, C, t\rangle$ true unless Bob was an Attribute Authority proper. This leaves us with the choice to use SDSI/SPKI certificates, which is what Greenpass implemented.

Time Travel There may be times when a relying party would like to reason about an event that has passed or one that has not happened yet (because some statements are not yet active). Maurer's model obviously breaks down due to the lack of the concept of time in the calculus. In the new model, we can reason about such events by manipulating the evaluation time t.

For example, assume that a relying party Alice is trying to verify a signature that Bob generated on April 21, 1984. (Note that Alice would need a mechanism such as a times-tamping service [36] in order to know that the signature existed on April 21, 1984). Further, assume that Alice believed that the CA X had an authentic binding between its public key and certificate information, and that it trusted the CA during that time period. Last, Alice would have to possess a certificate for Bob's which was valid during that period.

More formally, let $\mathcal{I}_0$ be the time period from January 1, 1984 to December 31, 1984. Let $\mathcal{I}_1$ be the time period from April 1, 1984 to April 30, 1984. Finally, let $\mathcal{Q} \subseteq \mathcal{D}$. Alice's initial view is given by:

$$\mathit{View}_A = \{\mathit{Aut}(A, X, \mathcal{P}, \mathcal{I}_0),\ \mathit{Trust}(A, X, \mathcal{D}, \mathcal{I}_0),\ \mathit{Cert}(X, B, \mathcal{Q}, \mathcal{I}_1)\}\ .$$

Now, at evaluation time t where $t \in \{\mathcal{I}_0 \cap \mathcal{I}_1\}$ (i.e., t is some time in April, 1984), Alice can use rule (5) to derive the authenticity of the binding between Bob's public key and his certificate information:

$$\mathit{Aut}(A, X, \mathcal{P}, \mathcal{I}_0),\ \mathit{Trust}(A, X, \mathcal{P}, \mathcal{I}_0),\ \mathit{Valid}\langle A, \mathit{Cert}(X, B, \mathcal{Q}, \mathcal{I}_1), t\rangle \vdash \mathit{Aut}(A, B, \mathcal{Q}, \mathcal{I}_1).$$

Thus, Alice can use Bob's public key to verify the signature (she could also use any other of Bob's properties in the set $\mathcal{Q}$) because $\mathit{Aut}(A, B, \mathcal{Q}, \mathcal{I}_1) \in \overline{\mathit{View}_A(t)}$ when t is some time in April, 1984. If she were to try and derive the same statement in May of 1984, she would fail because Bob's certificate expired after April, 1984. Since the evaluation time t would be in May, 1984, we have $t \notin \{\mathcal{I}_0 \cap \mathcal{I}_1\}$, causing the preconditions for rule (5) to be unsatisfied and the validity template instantiation to fail. Thus, $\mathit{Aut}(A, B, \mathcal{Q}, \mathcal{I}_1) \notin \overline{\mathit{View}_A(t)}$.

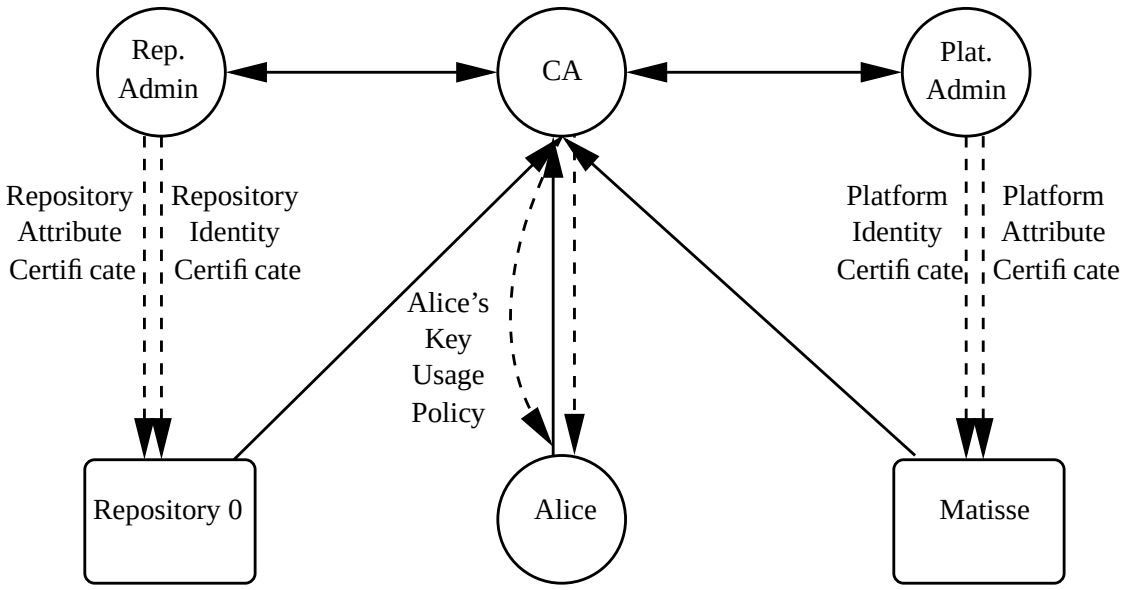

Figure 8.6: The entities, trust relationships, and certifi cates in SHEMP.

8.4 SHEMP Correctness Proof

In order for a PKI system such as SHEMP to be considered *correct*, it must allow relying parties to make reasonable trust judgments about targets. Minimally, a relying party Bob must be able to establish the authenticity of the binding expressed by Alice's certificate. In SHEMP, entities use short-lived PCs in conjunction with standard X.509 identity certificates. Thus, in order for Bob to reason about the SHEMP-user Alice, he must establish the authenticity of Alice's PC.

8.4.1 Modeling SHEMP

As described in Chapter 5, the process of setting up SHEMP results in a number of entities, trust relationships, and certificates which are depicted in Figure 8.6.[3]

Using our directed graph notation, it is possible to generate the statement graph of the SHEMP system. SHEMP's trust and authenticity relationships are shown in Figure 8.7,

[3]Figure 8.6 is identical to Figure 5.4 and is placed here for easy reference.

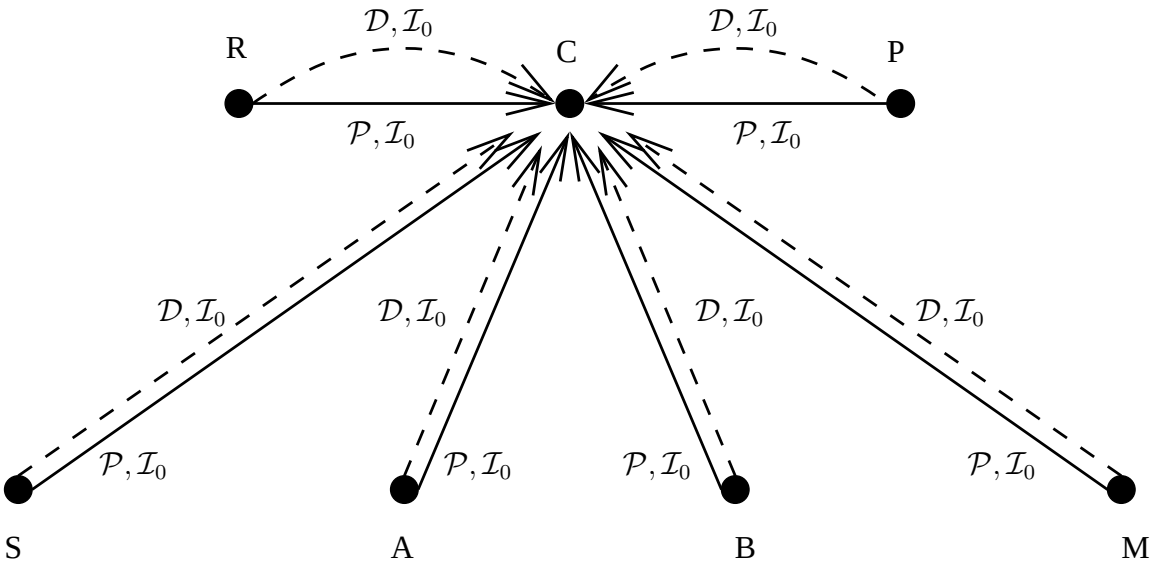

Figure 8.7: Statement graph for SHEMP entities, trust relationships, and authenticity beliefs.

and the trust transfers and certificates are shown in Figure 8.8.

Figure 8.7 depicts the Repository Administrator R, the CA C, the Platform Administrator P, the repository S, users Alice A and Bob B, and Alice's machine named Matisse M. The solid edges in Figure 8.7 indicate that all entities believe that the binding between the CA's public key and the CA's properties (the set $\mathcal{P}$) is authentic during the interval $\mathcal{I}_0$. The dashed edges represent the fact that they all trust the CA to issue certificates and trust transfers over domain $\mathcal{D}$ during the interval $\mathcal{I}_0$.

Figure 8.8 depicts the entities, certificates, and trust transfers in SHEMP. The solid edges represent the certificates—both identity certificates and attribute certificates. Since the different certificates are binding different properties to the same entity, the properties are represented as different sets. For example, the Repository Administrator R issues the repository identity certificate which assigns the set of properties $\mathcal{W}$ to the repository S. The Repository Administrator R also issues the Repository Attribute Certificate (RAC) which binds security attributes of S (denoted as the set $\mathcal{W}'$) to S's public key.[4] Alice's Key Usage

[4]In the SHEMP prototype, the identity certificate and attribute certificate also have the same identifier.

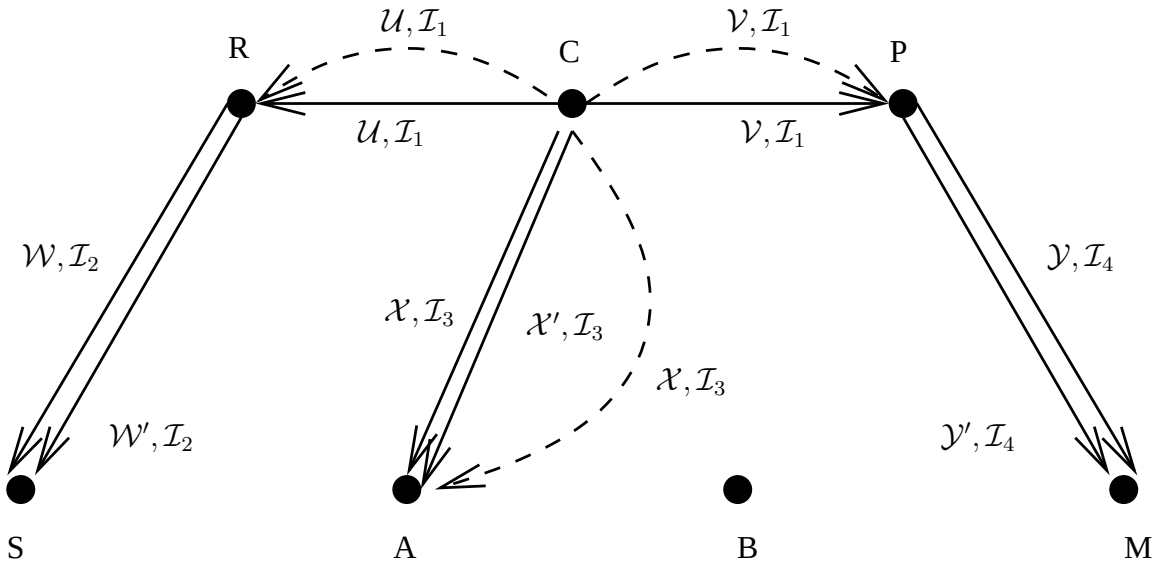

Figure 8.8: The statement graph for the SHEMP entities, certificates, and trust transfers.

Policy (KUP) is represented as an attribute certificate issued by the CA to Alice. For the remainder of this section, we denote the sets of properties contained in attribute certificates with the prime. For instance, *Cert*$(P, M, \mathcal{Y}, \mathcal{I}_4)$ is M's identity certificate, whereas *Cert*$(P, M, \mathcal{Y}', \mathcal{I}_4)$ is M's attribute certificate.

It should also be noted that Bob is not certified, as he is a relying party in this scenario. In general, relying parties may be certified, but are not required to be under SHEMP. However, Bob must trust the CA and believe that the CA's certificate is authentic in order to believe any signed statements from the CA (as shown in Figure 8.7).

The trust transfer statements are represented by the dashed edges in Figure 8.8. For simplicity, we assign the same lifespan to trust transfer and certificates issued to the same entity.

8.4.2 Proving SHEMP is Correct

As described in Section 8.1.3, Maurer's calculus allows relying parties to conclude false statements, and thus cannot be used to reason about SHEMP. For SHEMP to be considered

correct, we must show that it allows the relying party Bob to conclude about Alice what he should, and disallows him from concluding things he should not. Specifically, Bob (B) must be able to establish the authenticity of Alice's (A) Proxy Certificate, and should not be able to establish the authenticity of her certificate if any relevant statements have expired or been revoked. More formally, at evaluation time t, *Aut*$(B, T, \mathcal{Q}, \mathcal{I}_5) \in \overline{View_B(t)}$ where T is the temporary keypair described by Alice's PC (i.e., P_T is the public key contained in A's PC), $\mathcal{Q}$ is the set of properties described by the Alice's PC (which includes the PCI extension information about her current environment and KUP), and $\mathcal{I}_5$ is the PC's lifespan (on the order of hours).

Definition 5. We introduce one last definition which is specific to SHEMP (i.e., not part of our calculus itself) before we state and prove the theorem. A *public statement* is a statement that is accessible to all entities in the system (i.e., a statement that exists in every entity's view). Public statements include certificates (both identity and attribute) and trust transfer statements. We define the set $View_p$ to be the set of all public statements in the system. Figure 8.8 depicts the set of public statements under SHEMP:

$$\textit{View}_p = \left\{ \begin{array}{lll} \textit{Cert}(C, R, \mathcal{U}, \mathcal{I}_1), & \textit{Cert}(C, P, \mathcal{V}, \mathcal{I}_1), & \textit{Cert}(C, A, \mathcal{X}, \mathcal{I}_3), \\ \textit{Cert}(C, A, \mathcal{X}', \mathcal{I}_3), & \textit{Cert}(R, S, \mathcal{W}, \mathcal{I}_2), & \textit{Cert}(R, S, \mathcal{W}', \mathcal{I}_2) \\ \textit{Cert}(P, M, \mathcal{Y}, \mathcal{I}_4), & \textit{Cert}(P, M, \mathcal{Y}', \mathcal{I}_4), & \textit{Tran}(C, R, \mathcal{U}, \mathcal{I}_1), \\ \textit{Tran}(C, P, \mathcal{V}, \mathcal{I}_1), & \textit{Tran}(C, A, \mathcal{X}, \mathcal{I}_3), & \end{array} \right\}.$$

Theorem (Correctness) Let SHEMP-user A be a target, B be a relying party, T be the key described by A's PC, $\mathcal{Q}$ be the set of properties described in A's PC, and $\mathcal{X}$ be the set of A's properties where $\mathcal{Q} \subseteq \mathcal{X}$. At evaluation time t, *Aut*$(B, T, \mathcal{Q}, \mathcal{I}_5) \in \overline{View_B(t)}$ if and only if every statement in $View_p$ is active.

Proof *Part I.* Every statement in $View_p$ is active $\Rightarrow$ *Aut*$(B, T, \mathcal{Q}, \mathcal{I}_5) \in \overline{View_B(t)}$.

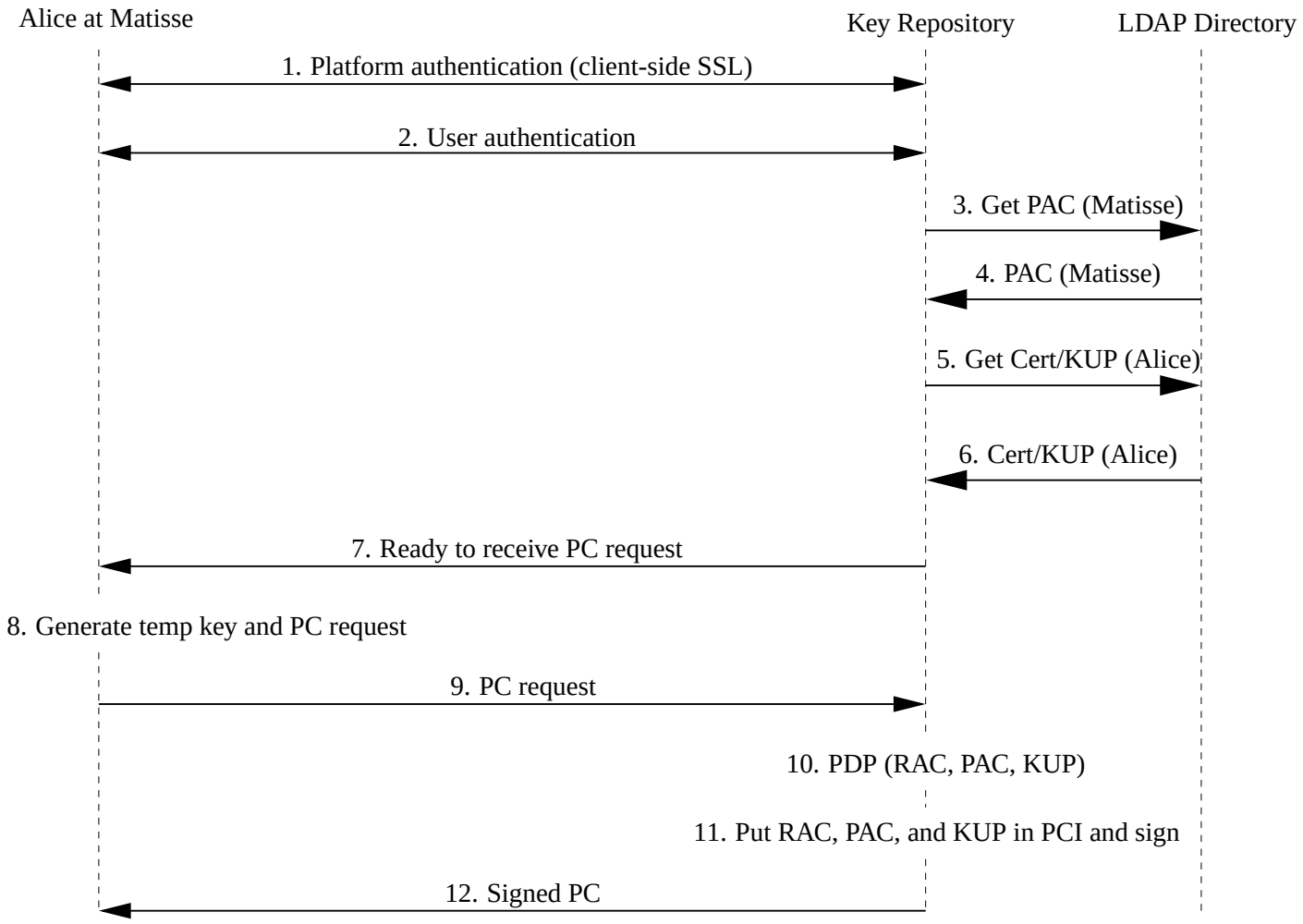

Figure 8.9: The basic protocol for generating a Proxy Certifi cate under SHEMP.

Suppose that every statement in $View_p$ is active. As discussed in Chapter 5 and illustrated in Figure 8.9[5], the repository S will sign A's PC request if:

1. The requesting platform M and the repository mutually authenticate, i.e., they establish the authenticity of each other's public key,
2. A successfully authenticates to S,
3. The attribute certificates are verified, i.e., the repository believes all of the attributes are authentic, and
4. A's policy along with the attributes allow the operation.

More formally, S will use A's private key to sign her PC request and place *Cert*$(A, T, \mathcal{Q}, \mathcal{I}_5)$ in $View_p$ if:

1. *Platform Authentication.* *Aut*$(S, M, \mathcal{Y}, \mathcal{I}_4) \in \overline{View_S(t)}$ and *Aut*$(M, S, \mathcal{W}, \mathcal{I}_2) \in \overline{View_M(t)}$.
2. *User Authentication.* A successfully authenticates to S,
3. *Attribute Verification.* *Aut*$(S, M, \mathcal{Y}', \mathcal{I}_4) \in \overline{View_S(t)}$, *Aut*$(S, S, \mathcal{W}', \mathcal{I}_2) \in \overline{View_S(t)}$, and *Aut*$(S, A, \mathcal{X}', \mathcal{I}_3) \in \overline{View_S(t)}$.
4. *Policy Check.* A's policy along with the attributes allow the operation.

If any of the steps fail, then the repository will not sign *Cert*$(A, T, \mathcal{Q}, \mathcal{I}_5)$, and the validity template *Valid*$\langle B,$ *Cert*$(A, T, \mathcal{Q}, \mathcal{I}_5), t\rangle$ will always evaluate to false. This would prevent the PC from being used in *any* applications of rule (5) or (6), thus making in impossible for any relying party to deduce the authenticity of the PC.

[5]Figure 8.9 is identical to Figure 5.5, and is placed here for easy reference

Next, we examine each of the conditions in detail.

Platform Authentication. Using the SHEMP model depicted in Figure 8.7 and Figure 8.8, S's initial view is the set of statements contained in $View_p$, plus its own initial beliefs (i.e., $View_S = View_p \cup \{\textit{Aut}(S, C, \mathcal{P}, \mathcal{I}_0), \textit{Trust}(S, C, \mathcal{D}, \mathcal{I}_0)\}$). Since we are supposing that all of the statements in $View_p$ are active, we have evaluation time $t \in \{\mathcal{I}_0 \cap \mathcal{I}_1 \cap \mathcal{I}_2 \cap \mathcal{I}_3 \cap \mathcal{I}_4\}$. Recall that the CA's domain is $\mathcal{D}$, and assume that $\mathcal{V} \subseteq \mathcal{D}$ and $\mathcal{Y} \subseteq \mathcal{V}$. The statement $\textit{Aut}(S, M, \mathcal{Y}, \mathcal{I}_4)$ can be derived by applying the rules (5) and (6) as follows:

$$\textit{Aut}(S, C, \mathcal{P}, \mathcal{I}_0),\ \textit{Trust}(S, C, \mathcal{D}, \mathcal{I}_0),\ \textit{Valid}\langle S, \textit{Cert}(C, P, \mathcal{V}, \mathcal{I}_1), t\rangle \vdash \textit{Aut}(S, P, \mathcal{V}, \mathcal{I}_1)$$

$$\textit{Aut}(S, C, \mathcal{P}, \mathcal{I}_0),\ \textit{Trust}(S, C, \mathcal{D}, \mathcal{I}_0),\ \textit{Valid}\langle S, \textit{Tran}(C, P, \mathcal{V}, \mathcal{I}_1), t\rangle \vdash \textit{Trust}(S, P, \mathcal{V}, \mathcal{I}_1)$$

$$\textit{Aut}(S, P, \mathcal{V}, \mathcal{I}_1),\ \textit{Trust}(S, P, \mathcal{V}, \mathcal{I}_1),\ \textit{Valid}\langle S, \textit{Cert}(P, M, \mathcal{Y}, \mathcal{I}_4), t\rangle \vdash \textit{Aut}(S, M, \mathcal{Y}, \mathcal{I}_4)\ .$$

If the validity template instantiations for X.509 certificates and trust transfers evaluate to true (i.e., the certificates are properly signed, have the `basicConstraints` fields set, etc.), then $\textit{Aut}(S, M, \mathcal{Y}, \mathcal{I}_4) \in \overline{View_S(t)}$. At this point, the repository S has authenticated the client platform M.

Similarly, M can authenticate S by first noting that $View_M = View_p \cup \{\textit{Aut}(M, C, \mathcal{P}, \mathcal{I}_0), \textit{Trust}(M, C, \mathcal{D}, \mathcal{I}_0)\}$. Since we are supposing that all of the statements in $View_p$ are active, we again have $t \in \{\mathcal{I}_0 \cap \mathcal{I}_1 \cap \mathcal{I}_2 \cap \mathcal{I}_3 \cap \mathcal{I}_4\}$. Further, assume that $\mathcal{U} \subseteq \mathcal{D}$ and $\mathcal{W} \subseteq \mathcal{U}$. The statement $\textit{Aut}(M, S, \mathcal{W}, \mathcal{I}_2)$ can be derived by applying the inference rules (5) and (6) as follows:

$$\textit{Aut}(M, C, \mathcal{P}, \mathcal{I}_0),\ \textit{Trust}(M, C, \mathcal{D}, \mathcal{I}_0),\ \textit{Valid}\langle M, \textit{Cert}(C, R, \mathcal{U}, \mathcal{I}_1), t\rangle \vdash \textit{Aut}(M, R, \mathcal{U}, \mathcal{I}_1)$$

$$\textit{Aut}(M, C, \mathcal{P}, \mathcal{I}_0),\ \textit{Trust}(M, C, \mathcal{D}, \mathcal{I}_0),\ \textit{Valid}\langle M, \textit{Tran}(C, R, \mathcal{U}, \mathcal{I}_1).t\rangle \vdash \textit{Trust}(M, R, \mathcal{U}, \mathcal{I}_1)$$

$$\textit{Aut}(M, R, \mathcal{U}, \mathcal{I}_1),\ \textit{Trust}(M, R, \mathcal{U}, \mathcal{I}_1),\ \textit{Valid}\langle M, \textit{Cert}(R, S, \mathcal{W}, \mathcal{I}_2), t\rangle \vdash \textit{Aut}(M, S, \mathcal{W}, \mathcal{I}_2)\ .$$

Again, if the validity template instantiations for X.509 certificates and trust transfers evaluate to true, then $\textit{Aut}(M, S, \mathcal{W}, \mathcal{I}_2) \in \overline{View_M(t)}$, meaning that the client platform M

has authenticated the repository S. Now that the platforms have mutually authenticated, the protocol of Figure 8.9 proceeds to step 2.

User Authentication. Since SHEMP is agnostic with respect to user authentication mechanisms, user authentication is not modeled formally. However, this step must succeed in order for the protocol to advance. As discussed in Chapter 5, the current prototype continues after A has presented a valid password.

Attribute Verification. Steps 3-9 of the protocol essentially gather the necessary statements needed to make the policy check. As the attributes certificates are gathered, the repository S establishes their authenticity. As previously noted, $View_S = View_p \cup \{\textit{Aut}(S, C, \mathcal{P}, \mathcal{I}_0), \textit{Trust}(S, C, \mathcal{D}, \mathcal{I}_0)\}$, and we are supposing that all statements in $View_p$ are active. Further, let $\mathcal{X}' \subseteq \mathcal{D}$, $\mathcal{V} \subseteq \mathcal{D}$, $\mathcal{Y}' \subseteq \mathcal{V}$, $\mathcal{U} \subseteq \mathcal{D}$, and $\mathcal{W}' \subseteq \mathcal{U}$. At evaluation time t, the authenticity of the attributes can be derived by using the inference rules as follows:

$$\textit{Aut}(S, C, \mathcal{P}, \mathcal{I}_0),\ \textit{Trust}(S, C, \mathcal{D}, \mathcal{I}_0),\ \textit{Valid}\langle S, \textit{Cert}(C, A, \mathcal{X}', \mathcal{I}_3), t\rangle \vdash \textit{Aut}(S, A, \mathcal{X}', \mathcal{I}_3)$$

$$\textit{Aut}(S, C, \mathcal{P}, \mathcal{I}_0),\ \textit{Trust}(S, C, \mathcal{D}, \mathcal{I}_0),\ \textit{Valid}\langle S, \textit{Cert}(C, P, \mathcal{V}, \mathcal{I}_1), t\rangle \vdash \textit{Aut}(S, P, \mathcal{V}, \mathcal{I}_1)$$

$$\textit{Aut}(S, C, \mathcal{P}, \mathcal{I}_0),\ \textit{Trust}(S, C, \mathcal{D}, \mathcal{I}_0),\ \textit{Valid}\langle S, \textit{Tran}(C, P, \mathcal{V}, \mathcal{I}_1), t\rangle \vdash \textit{Trust}(S, P, \mathcal{V}, \mathcal{I}_1)$$

$$\textit{Aut}(S, P, \mathcal{V}, \mathcal{I}_1),\ \textit{Trust}(S, P, \mathcal{V}, \mathcal{I}_1),\ \textit{Valid}\langle S, \textit{Cert}(P, M, \mathcal{Y}', \mathcal{I}_4), t\rangle \vdash \textit{Aut}(S, M, \mathcal{Y}', \mathcal{I}_4)$$

$$\textit{Aut}(S, C, \mathcal{P}, \mathcal{I}_0),\ \textit{Trust}(S, C, \mathcal{D}, \mathcal{I}_0),\ \textit{Valid}\langle S, \textit{Cert}(C, R, \mathcal{U}, \mathcal{I}_1), t\rangle \vdash \textit{Aut}(S, R, \mathcal{U}, \mathcal{I}_1)$$

$$\textit{Aut}(S, C, \mathcal{P}, \mathcal{I}_0),\ \textit{Trust}(S, C, \mathcal{D}, \mathcal{I}_0),\ \textit{Valid}\langle S, \textit{Tran}(C, R, \mathcal{U}, \mathcal{I}_1), t\rangle \vdash \textit{Trust}(S, R, \mathcal{U}, \mathcal{I}_1)$$

$$\textit{Aut}(S, R, \mathcal{U}, \mathcal{I}_1),\ \textit{Trust}(S, R, \mathcal{U}, \mathcal{I}_1),\ \textit{Valid}\langle S, \textit{Cert}(R, S, \mathcal{W}', \mathcal{I}_2), t\rangle \vdash \textit{Aut}(S, S, \mathcal{W}', \mathcal{I}_2)\ .$$

If the validity templates for the X.509 certificates and trust transfers all evaluate to true, then $\{\textit{Aut}(S, A, \mathcal{X}', \mathcal{I}_3),\ \textit{Aut}(S, M, \mathcal{Y}', \mathcal{I}_4),\ \textit{Aut}(S, S, \mathcal{W}', \mathcal{I}_2)\} \in \overline{View_S(t)}$, meaning that the repository S is convinced of the authenticity of all of the relevant attribute certificates.

Policy Check. If S is successful in verifying all of the attributes, the protocol advances to step 10. Since the policy and attributes themselves (i.e., not the certificate, but the attributes in the certificate) are in XACML and XML, they are evaluated out of band by an

XACML Policy Decision Point (PDP) and are not modeled formally. As with the user authentication step, the policy check must be successful in order for the protocol to advance. In the SHEMP prototype, the PDP must return "Permit" in order for the protocol to advance. (Section 8.4.3 explores this topic in more detail.)

Issuing a PC. If the protocol reaches step 11, then S uses A's private key to sign A's PC request. In our model, this is represented by S generating the statement $\mathit{Cert}(A, T, \mathcal{Q}, \mathcal{I}_5)$ and forming the set $View_p\prime$ where:

$$View_p\prime = View_p \cup \{\mathit{Cert}(A, T, \mathcal{Q}, \mathcal{I}_5)\} .$$

Last, for a relying party B to be able to use the PC, B must establish its authenticity at some evaluation time t. If the protocol succeeded, then $View_B = View_p\prime \cup \{\mathit{Aut}(B, C, \mathcal{P}, \mathcal{I}_0), \mathit{Trust}(B, C, \mathcal{D}, \mathcal{I}_0)\}$. By the supposition, we have $t \in \{\mathcal{I}_0 \cap \mathcal{I}_1 \cap \mathcal{I}_2 \cap \mathcal{I}_3 \cap \mathcal{I}_4\}$. Additionally, assume that the PC is active (i.e., $t \in \mathcal{I}_5$), $\mathcal{X} \subseteq \mathcal{D}$, and $\mathcal{Q} \subseteq \mathcal{X}$. B can derive the authenticity of A's PC by applying the inference rules as follows:

$$\mathit{Aut}(B, C, \mathcal{P}, \mathcal{I}_0),\ \mathit{Trust}(B, C, \mathcal{D}, \mathcal{I}_0),\ \mathit{Valid}\langle B, \mathit{Cert}(C, A, \mathcal{X}, \mathcal{I}_3), t\rangle \vdash \mathit{Aut}(B, A, \mathcal{X}, \mathcal{I}_3)$$

$$\mathit{Aut}(B, C, \mathcal{P}, \mathcal{I}_0),\ \mathit{Trust}(B, C, \mathcal{D}, \mathcal{I}_0),\ \mathit{Valid}\langle B, \mathit{Tran}(C, A, \mathcal{X}, \mathcal{I}_3), t\rangle \vdash \mathit{Trust}(B, A, \mathcal{X}, \mathcal{I}_3)$$

$$\mathit{Aut}(B, A, \mathcal{X}, \mathcal{I}_3),\ \mathit{Trust}(B, A, \mathcal{X}, \mathcal{I}_3),\ \mathit{Valid}\langle B, \mathit{Cert}(A, T, \mathcal{Q}, \mathcal{I}_5), t\rangle \vdash \mathit{Aut}(B, T, \mathcal{Q}, \mathcal{I}_5) .$$

If the validity templates for the X.509 certificate, trust transfer, and SHEMP PC evaluate to true (we will explore the validity template instantiation for SHEMP PC's in Section 8.4.3), then $\mathit{Aut}(B, T, \mathcal{Q}, \mathcal{I}_5) \in \overline{View_B(t)}$. □

Part II. $\mathit{Aut}(B, T, \mathcal{Q}, \mathcal{I}_5) \in \overline{View_B(t)} \Rightarrow$ every statement in $View_p$ is active.

We show this by contradiction. Assume that $\mathit{Aut}(B, T, \mathcal{Q}, \mathcal{I}_5) \in \overline{View_B(t)}$ and there is some statement $s \in View_p$ which is not active. At least one of the following is true:

- *Case 1:* s is one of the recommendations from C to one of the administrators (i.e., R and P). It would thus be impossible to apply inference rule (6) (see Section 8.2),

leaving it impossible for platforms to be able to authenticate one another. This would stop the protocol from advancing to step 2, and hence make it impossible for S to sign A's PC. Therefore, ***Aut***$(B, T, \mathcal{Q}, \mathcal{I}_5) \notin \overline{View_B(t)}$, which is a contradiction.

- *Case 2:* s is one of the identity certificates issued by an administrator (either R or P) to a machine (either S or M). It is possible to apply all of the inference rules in this case, but the resulting statement of authenticity would be inactive. For instance, assume that one of the identity certificates has a lifespan $\mathcal{I}$ and $t \notin \mathcal{I}$. It would be possible to derive a statement of authenticity using the inference rules, but since the resulting statement of authenticity has the same lifespan as the certificate ($\mathcal{I}$ in this case), then the statement of authenticity is not active by definition.

  As in the Case 1 above, this case makes it impossible for platforms to authenticate one another, and stops the protocol before proceeding to step 2. Therefore, ***Aut***$(B, T, \mathcal{Q}, \mathcal{I}_5) \notin \overline{View_B(t)}$, which is a contradiction.

- *Case 3:* s is one of the identity certificates issued by the CA C to one of the administrators. As discussed in Case 2 above, the statement of authenticity of the administrator's key would be inactive. Again, platform authentication could not occur, meaning that the protocol would not proceed to step 2. Thus, ***Aut***$(B, T, \mathcal{Q}, \mathcal{I}_5) \notin \overline{View_B(t)}$, which is a contradiction.

- *Case 4:* s is one of the attribute certificates. As with identity certificates, this case makes the resulting statement of authenticity inactive. If S cannot establish the authenticity of all of the attribute certificates, the policy check will fail in step 10, halting the protocol. Thus, ***Aut***$(B, T, \mathcal{Q}, \mathcal{I}_5) \notin \overline{View_B(t)}$, which is a contradiction.

- *Case 5:* s is the statement ***Tran***$(C, A, \mathcal{X}, \mathcal{I}_3)$. If A's recommendation is inactive, then the relying party B will be unable to apply inference rule (6), resulting in a failure to

establish trust in A. Thus, ***Aut***$(B, T, \mathcal{Q}, \mathcal{I}_5) \notin \overline{View_B(t)}$, which is a contradiction.

- *Case 6:* s is A's identity certificate. In this case, the relying party B is unable to establish the authenticity of A. As before, the resulting statement of A's authenticity will be inactive. The last application of inference rule (5) cannot be made, and thus, ***Aut***$(B, T, \mathcal{Q}, \mathcal{I}_5) \notin \overline{View_B(t)}$, which is a contradiction.

Therefore, every statement in $View_p$ must be active. □

8.4.3 A Detailed Example: The SHEMP Signing Proxy

The SHEMP correctness theorem states that a relying party B will be able to reason about A's PC if and only if all of the public statements in the system are active and valid (i.e., the validity templates all evaluate to true). In ***Part I*** of the theorem, we show how a relying party B can establish the authenticity of A's PC. In this section, we elaborate and illustrate how B can use A's PC to reason about A and her environment in order to make an access control decision.

Recall that our SHEMP prototype includes two applications: the decryption and signing proxies. In this section, we will explore the signing proxy in detail, and use our calculus to illustrate how it makes the decision to sign a message on A's behalf.[6]

In this scenario, let the relying party B be the signing proxy service running on the repository, and T be the temporary keypair described by A's PC. As discussed in Chapter 6, applications protect a resource with a policy. In this scenario, the resource is A's long-term private key and the policy is A's Key Usage Policy (KUP). When A makes a request to B to sign a message with A's private key, A sends the message, the signature which was generated with the private portion of T, and A's PC. In order for B to sign the message

[6]We could have chosen the decryption proxy for this purpose; the decision to use the signing proxy was completely arbitrary.

with A's long-term key, B must be able to derive $\mathit{Aut}(B, T, \mathcal{Q}, \mathcal{I}_5) \in \overline{View_B(t)}$.

Using the model of SHEMP presented in Figure 8.7 and Figure 8.8, we let B's initial view be the set $View_p\prime$ (see Definition 5) along with his initial beliefs, i.e., $View_B = View_p\prime \cup \{\mathit{Aut}(B, C, \mathcal{P}, \mathcal{I}_0), \mathit{Trust}(B, C, \mathcal{D}, \mathcal{I}_0)\}$.

The first step is for B to convince himself of the authenticity of A's identity certificate. Assuming that all of the statements are active at evaluation time t (i.e., $t \in \{\mathcal{I}_0 \cap \mathcal{I}_3\}$), and that the CA C is authorized to speak for A's attributes (i.e., $\mathcal{X} \subseteq \mathcal{D}$), then this is accomplished by applying rule (5) as follows:

$$\mathit{Aut}(B, C, \mathcal{P}, \mathcal{I}_0),\ \mathit{Trust}(B, C, \mathcal{D}, \mathcal{I}_0),\ \mathit{Valid}\langle B, \mathit{Cert}(C, A, \mathcal{X}, \mathcal{I}_3), t\rangle \vdash \mathit{Aut}(B, A, \mathcal{X}, \mathcal{I}_3)$$

As discussed in Section 8.2, a number of conditions must be met in order for A's X.509 identity certificate make the validity template instantiation for X.509 certificates evaluate to true:

1. the certificate's signature must verify,
2. the certificate must not have expired (we are assuming that $t \in \mathcal{I}_3$),
3. the certificate must not have been revoked,
4. the certificate chain length must not have been exceeded,
5. domain-specific constraints and extensions must be present, etc.

If the conditions are met, then B can deduce the authenticity of A's identity certificate. B's next task is to decide whether it can trust statements signed by A's long-term key (such as her Proxy Certificate). Again assuming that all of the statements are active at evaluation time t (i.e., $t \in \{\mathcal{I}_0 \cap \mathcal{I}_3\}$), and that the CA C is authorized to speak for A's attributes (i.e., $\mathcal{X} \subseteq \mathcal{D}$), then this is accomplished by applying rule (6) as follows:

$$\textit{Aut}(B, C, \mathcal{P}, \mathcal{I}_0),\ \textit{Trust}(B, C, \mathcal{D}, \mathcal{I}_0),\ \textit{Valid}\langle B, \textit{Tran}(C, A, \mathcal{X}, \mathcal{I}_3), t\rangle \vdash \textit{Trust}(B, A, \mathcal{X}, \mathcal{I}_3)$$

Since A has an X.509 identity certificate, the trust transfer statement is an implicit statement made be the CA by setting the `basicConstraints` extension. In order for the X.509 trust transfer validity template instantiation to evaluate to true, it must verify that the `basicConstraints` extension in A's identity certificate is set.

Once B has established the authenticity of A's identity certificate and trusts her to sign certificates with her long-term private key, he can attempt to establish the authenticity of her PC. Once again assuming that all of the statements are active at evaluation time t (i.e., $t \in \{\mathcal{I}_0 \cap \mathcal{I}_3 \cap \mathcal{I}_5\}$), and that A is authorized to speak for her PC's attributes (i.e., $\mathcal{Q} \subseteq \mathcal{X}$), then this is accomplished by applying rule (5) as follows:

$$\textit{Aut}(B, A, \mathcal{X}, \mathcal{I}_3),\ \textit{Trust}(B, A, \mathcal{X}, \mathcal{I}_3),\ \textit{Valid}\langle B, \textit{Cert}(A, T, \mathcal{Q}, \mathcal{I}_5), t\rangle \vdash \textit{Aut}(B, T, \mathcal{Q}, \mathcal{I}_5)\ .$$

Since $\textit{Cert}(A, T, \mathcal{Q}, \mathcal{I}_5)$ is A's PC, B has to determine whether the validity template instantiation for a SHEMP PC evaluates to true. Concretely, the instantiation checks the following:

1. the PC is properly signed by A's long-term private key,
2. the PC has not expired,
3. the PC must be a properly formatted X.509 PC with the Proxy Certificate Information (PCI) extension set (the X.509 PC standard mandates this) and adhering to the PC naming convention described in the X.509 PC standard, and
4. $\mathcal{Q} \subseteq \mathcal{X}$ must be true (we assumed it was, but in general, the template instantiation needs to check). As discussed in the "Delegation" example from Section 8.3, A must

only assign properties to the PC which it is allowed to assign. In this scenario, the set of properties $\mathcal{X}$ is acting as A's domain.

If the conditions are met, then B believes that the PC binds the temporary keypair T to the set of properties $\mathcal{Q}$ during the interval $\mathcal{I}_5$. B can now make an access control decision based on A's current environment and her KUP.

Let O be the operation that the PC is being used for (message signing in this example) and N be the policy protecting the requested resource (N is A's KUP in this example). Recall from Figure 8.8 that $\mathcal{Y}'$ is the set of platform attributes and $\mathcal{W}'$ is the set of repository attributes. Last, recall from Chapter 5 that a signed PC contains N, $\mathcal{Y}'$, and $\mathcal{W}'$ in the PCI extension (i.e., $\{N, \mathcal{Y}', \mathcal{W}'\} \in \mathcal{Q}$). We can model the Policy Decision Point (PDP) as a function which takes an operation and attributes as inputs, and returns a boolean:

$$PDP : O \times N \times \mathcal{Y}' \times \mathcal{W}' \longrightarrow \{0, 1\} .$$

Finally, B can use the PDP function with the information in the PC to decide whether he should sign the message with A's long term key by calculating $PDP(O, N, \mathcal{Y}', \mathcal{W}')$. If the PDP function returns true, then B signs the message with A's long term private key and returns the message to A.

8.5 Chapter Summary

In this chapter, we introduced Maurer's framework for reasoning about PKI systems, and showed how the model is inadequate for dealing with real-world PKIs. Drawing on our own experience with such systems, we introduced a model based on Maurer's which we used to reason about a number of systems, including SHEMP. We used the model to show that SHEMP allows relying parties to make reasonable trust judgments, and thus satisfies the final criterion of Chapter 3.

While our new model makes it possible to reason about a number of different types of PKIs and has been useful in practice, it is not perfect. There are a number of interesting potential future directions for this research. Since they are not relevant to SHEMP, we give them only brief mention in this chapter.

First, our model does not describe how well the properties in the certificates match the real world properties of certificate's subject. A similar issue arises in the field of program verification. One might determine how well the program fits the specification. However, this does not answer the question "Is my specification any good?" Such approaches yield a program which is at most as correct as the specification. In determining authenticity of a binding between a set of properties and a public key, the relying party trusts the attributes at most as much as it trusts the issuer. If an issuer is careless (or malicious), and binds false properties to Bob's public key then, under the new model (and in the real world), Alice will accept false properties about Bob. As an alternative view, we might consider whether the building blocks of a particular certificate scheme are in fact sufficiently *articulate* for relying parties to make the correct decision, or consider the size of the fraction of the space where relying parties make the wrong decisions. Further investigation into this issue, perhaps including automated formal methods, is an area for future work.

Second, the inclusion of time in the new model makes it *nonmonotonic*: true statements can become false over time. Nonmonotonicity can have a fatal side effect: a relying party may deduce authenticity when it should not. Some statement may have expired or been revoked, and the relying party has not received the revocation information yet. Li and Feigenbaum [61] introduce a concept of "fresh time" which could be used either in the certificate's properties, or possibly as an explicit parameter to make the system monotonic. Using fresh times in our model is another area for future work.

Last, certification and trust transfer statements in our new model are similar to Jon Howell's "speaks-for-regarding" operator [42]. However, our statements go beyond How-

ell's because they are applicable to a number of certificate formats (not just SDSI/SPKI), and they allow cases where transfers of trust are expressed implicitly (e.g., via the X.509 `basicConstraints`). If a relying party Alice receives multiple certificates about Bob, and she successfully deduces their authenticity (i.e., the authenticity of the bindings they contain), then Alice may hold multiple sets of properties assigned to Bob. What kind of set operations should we allow on these sets of properties? Howell disallows the relying party to use the union operation, but allows intersection. Considering the universe of allowable set operations is another area for future work.

Chapter 9

Summary and Conclusion

This research began when we discovered numerous problems with the current client-side approach to deploying PKI. As part of Dartmouth's PKI lab and participants in a large campus-wide PKI roll-out, we felt it was necessary to ask the fundamental question: "Does it work?" Before we spend hundreds of man-hours and thousands of dollars, do we have any reason to believe that once we are finished, Bob will be able to conclude that Alice is aware of and intended requests sent in her name?

As we illustrated in Chapter 2, the answer is "no." The problem is that common desktops are inherently unable to protect users' private keys. Fundamentally, the TCB on standard desktops 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. In addition to security concerns, the current client-side approach has mobility issues as well, often making users sacrifice security in order to gain mobility. Moreover, the current client-side paradigm makes it impossible for users, administrators, and application developers to render accurate mental models of the system. Taken together, these shortcomings in security, mobility, and usability make it impossible for relying parties to make reasonable trust decisions.

Starting with the problems of the current approach, we derived criteria for making desk-

tops usable PKI clients. As we established in Chapter 3, any solution which claims to make desktops usable for PKI must address security, mobility, and usability. Above all, the solution must allow relying parties to draw the trust conclusions they should, and disallow them from drawing ones they should not. There are numerous solutions which address some of the criteria, but a successful solution must address all of them.

Starting with the approach employed by the Grid community (i.e., the MyProxy online credential repository), we designed a system which makes desktops usable PKI clients: SHEMP. In Chapter 5, we discussed the design and implementation of the SHEMP system. In Chapter 6, we presented two concrete applications that we built with SHEMP: the decryption and signing proxies. These applications demonstrate that SHEMP can be used to build a real-world PKI. The proxies give organizations and application developers the full set of PKI primitives: authentication, decryption, and signing. Finally, we presented a number of application designs which take advantage of SHEMP's features to consider the user's environment when making decisions.

SHEMP meets the criteria we established in Chapter 3. In Chapter 7, we offered a security analysis of SHEMP, illustrated how it minimizes the risks and impacts of a private key disclosure, and how it can defend against the keyjacking attacks of Chapter 2. We discussed how SHEMP maintains security while providing mobility through the use of environmental attributes and Key Usage Policies. Finally, we showed that SHEMP is usable by presenting the results of our usability study and performance analysis. The results indicate the SHEMP's policy framework can be used to accurately capture a mental model of the system given the right tools, and that SHEMP's overhead is imperceivable by humans.

Finally, SHEMP allows relying parties to make reasonable trust judgements. In Chapter 8, we introduced a tool for evaluating PKIs: Maurer's calculus. We showed how and where Maurer's approach (specifically, his deterministic model) fails to capture the messiness of the real world. We introduced our new model which is rooted in Maurer's, and

overcomes Maurer's limitations. We demonstrated how our model can be used to reason about a number of real-world systems. We concluded by applying our model to SHEMP in order to show that SHEMP allows relying parties to make reasonable trust judgements.

Our contributions include the discovery of the problem, the design and implementation of the SHEMP system, and the extended version of Maurer's calculus. In addition to making desktops usable for PKI, our contributions to the PKI designer's toolbox allow designers to test for insecurities, construct PKI systems which fit their specific threat model, and formally reason about PKI systems in general. Individually, these are all useful capabilities, even outside of the context of this thesis.

9.1 Below the Surface

The keyjacking results of Chapter 2 suggest that the reason current desktops are not usable as PKI clients is that system designers are forced to operate under the assumption that the entire desktop is trusted. Even if system designers are careful, the construction of modern systems makes this assumption necessary. As we have discussed in depth, the TCB for modern desktop PKI clients includes the entire desktop: the OS and numerous applications, some of which are inseparable from the OS.

The experiments in Chapter 2 show that when just one malicious executable is introduced into the TCB (i.e., runs on the desktop), the security of the entire system can be undermined. Additionally, the experiments show that it is difficult 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, allowing it to vary with time, 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), and giving relying parties relevant information about the current environment.

SHEMP shrinks the TCB by getting private keys off of the desktop altogether, and placing them in a safe place: the repository. By taking advantage of secure hardware, the repository can shrink the TCB further by protecting the private keys as well as the repository application software itself. Since the repository application is orders of magnitude smaller than a general purpose OS, SHEMP greatly shrinks the number of lines of code in the TCB. Furthermore, SHEMP can shrink the TCB on each client by using secure hardware at the client, if available. Finally, although humans do not directly contribute to the TCB in the standard sense, placing the keys under the jurisdiction of one entity (the Repository Administrator) 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.

When users need to access to their private key, they log on to the repository, generate a temporary keypair, and have the repository use their long-term private key to sign a Proxy Certificate containing the public portion of their temporary keypair. The desktop must now be trusted as long as the Proxy Certificate is valid, albeit to a lesser extent, since the keypair on the desktop is not as important as the long-term private key. Nevertheless, the net effect of issuing a Proxy Certificate is an expansion of the TCB. Rather than picturing the TCB as a static security perimeter around specific parts of the system, we view it as a dynamic entity which has a small size in the steady state, and expands only when needed. The short lifespans of the Proxy Certificates keep pressure on the TCB to shrink, and as we discussed in Chapter 7, the total TCB for an organization using SHEMP is never larger than the total TCB for the same organization using the current client-side PKI approach.

Rather than consider the TCB some abstract entity which only security analysts are interested in, SHEMP attempts to make the TCB concrete and accessible to parties which rely on it most. The attribute certificates assigned to the client desktops and the repository contain attributes which are essentially domain-specific statements expressing the state of the TCB which the user is operating in. Using these attributes, a relying party can take into

account the state of the TCB when making decisions, and decide for themselves how much trust should be placed in the requester. Additionally, the SHEMP policy language allows users, administrators, and application developers to govern their resources (such as private keys or critical application operations) in such a way that takes the TCB into account.

Under the surface, SHEMP is really about managing the TCB for a PKI system. Many of these core concepts involving the TCB's size are not novel, dating back to the Orange Book days. Furthermore, some of the benefits to applications discussed in Chapter 6, such as Multi-Level Security, also date back to the Orange Book. SHEMP applies new technologies such as Proxy Certificates to old techniques. One novel technique employed by SHEMP is viewing the TCB as a dynamic entity rather that a static "security perimeter." By allowing the TCB to vary with time, SHEMP can maintain a very small TCB until it is necessary to expand it.

9.2 Future Directions

We close this thesis by discussing some potential avenues for further research. While there are a number of potential applications and environments where SHEMP could prove to be useful, we are focused more on open-ended research questions than on concrete development tasks. In this section, we discuss some possible approaches for expanding the SHEMP framework itself, enabling new programming paradigms, and applying our calculus to a number of other areas.

Shrinking SHEMP A new project in our lab involves using SHEMP as a personal key repository instead of a repository for an entire user population. Under this type of scenario, some device of Alice's (e.g., her smart phone or PDA) would run the SHEMP repository and house her private key. As Alice approaches a client machine in her domain, her

personal repository and the potential client connect and the client gets a Proxy Certificate signed by the private key in Alice's personal repository. The new Proxy Certificate contains the environmental information, as before.

Using the SHEMP architecture in such a manner shrinks the PKI TCB even further, as the repository now houses only one user's key. This approach allows Alice to use PKI applications on a number of different types of client machines without having to port those applications to her personal repository's (possibly constrained) platform. As with the standard SHEMP approach discussed throughout this thesis, this approach allows Alice to operate from clients with different levels of trustworthiness.

One argument against SHEMP is that the more clients have secure hardware, the less need there is for a system like SHEMP. The argument is that, if all clients have secure hardware, then the current client-side PKI vision may not be so bad. There are at least two responses to this argument. First, secure clients (e.g., clients using Intel's LaGrande Technology [120]) will likely still run big, bloated applications, and these applications will expand the TCB. While hardened clients are an improvement, they will still have security issues. Second, the small version of SHEMP presented in this section fits nicely into an environment where clients are hardened with secure hardware. Alice can use applications on multiple machines without having to export and re-import her key or port applications to run on her personal repository. Furthermore, the SHEMP approach keeps the PKI TCB small, and allows it to vary with time.

Repository Networks Under the current model of SHEMP, when a user Alice needs to use her private key, she begins by using the SHEMP client software to log on to the repository which holds her key. If Alice's key ever needs to move to a different repository, then Alice will have to export and re-import her key. As we have discussed throughout this thesis, migrating private keys in such a manner presents an opportunity for private key

disclosure.

If there were a method to safely migrate users' private keys between repositories, SHEMP would become a more flexible platform. A SHEMP repository may want to allow key migration in a number of scenarios. For example, a busy repository may wish to temporarily offload some of its clients to an underutilized repository. As another example, there may be some number of organizations that participate in a federation, and would like allow their users to move freely between the organizations and still have access to their private keys. Finally, Alice may want to ask *any* repository for her key, and have it automatically arrive there.

One possible solution may involve treating repositories as nodes in a peer-to-peer network. In other work, we have explored putting the Gnutella peer-to-peer protocol inside of an IBM 4758 [70], as well as building peer-to-peer networks of Semi-Trusted Mediators (SEMs) [129].[1] From a high level, the peer-to-peer protocol can be used to locate keys in different repositories, the secure hardware authentication mechanisms (such as attestation and outbound authentication) can be used to mutually authenticate the repositories, and a secure transport protocol such as Sacred [102, 103, 104] could be used to safely transport the private key between repositories. SHEMP's current policy framework currently accounts for differing repository security levels, and is thus already robust enough to handle key migration scenarios. Connecting repositories in this way could lead to some interesting future work.

Security-context Aware Computing As a second future direction for SHEMP, we consider the scenario where a user Alice is using application X, and X relies on some mobile code components such as *Common Object Request Broker Architecture* (CORBA) objects to perform some operations. We assume that X is written to take advantage of the SHEMP

[1] Recently, we ported the SEM prototype to run on our Bear/Enforcer platform. That work has been submitted for publication.

system, namely its policy mechanism. Concrete examples of such applications were given in Chapter 6, but to briefly recount, the X gathers attributes about Alice's current operating environment and presents them, along with X's policy, to a Policy Decision Point (PDP) for evaluation.

In this scenario, the CORBA objects which actually implement the operations for X may have their own policy. In order to make an security decision, application X must collect the environmental attributes along with X's own policy, as before. However, it must also gather the policies from the CORBA objects for evaluation to determine whether the operation is allowed to occur in the given environment. SHEMP's current PDP is already equipped to take multiple policies into account when making policy decisions.

Such an approach allows application developers to construct systems by using composition, while at the same time giving module developers have a method to control the set of operations visible to the client application. This allows both application and module developers to take Alice's security context into account when making decisions.

Attestation Protocol Verification As a final potential future direction, we consider an application not of SHEMP directly, but of the calculus discussed in Chapter 8. The SHEMP toolkit includes possibly using secure hardware, not only for protecting code and data, but also for giving relying parties a reason to believe that code and data have not been tampered with. As discussed in Chapter 4, modern secure devices use a feature known as *attestation* for these purposes (the IBM 4758 refers to this as *outbound authentication*).

While such protocols have a great deal of utility, they are often quite complex, involving multiple parties using multiple keypairs to make signed statements about the trustworthiness of multiple entities. Reasoning about such complexity is often best left to formal methods. This concept is not novel; Sean Smith has applied a version of Maurer's calculus to the outbound authentication protocol of the IBM 4758 [116]. Applying the calculus of Chap-

ter 8 could serve to illustrate issues with the calculus as well as provide formal verification about attestation protocols which have not been scrutinized with formal techniques—such as the TCPA/TCG attestation protocols.

9.3 Conclusion

The goal of this thesis was to make desktops usable for PKI. In the course of this research, we made the following contributions:

- We discovered architectural flaws within Microsoft's Cryptographic API, and developed a suite of attacks to exploit those vulnerabilities. We concluded that modern desktops have a large TCB, and as a result, are unsuitable for use as PKI clients.
- We established criteria for making desktops usable as PKI clients, and designed and implemented a system which meets that criteria: SHEMP.
- We took a new approach to reasoning about the TCB. We adopted the view that the TCB should be treated as a dynamic entity, rather than some static security perimeter.
- We implemented applications which use Proxy Certificates for decryption and signing. Previous uses of Proxy Certificates were limited to authentication and dynamic delegation.
- We developed a formal framework for reasoning about PKI systems in general, and illustrated how it can be used to reason about systems like SHEMP. We then used it to prove that SHEMP is correct.

Using our new tools and the knowledge they provided, we were able to design, implement, and reason about a system which makes desktops usable as PKI clients.

Appendix A

SHEMP Reference Manual

This appendix contains instructions for setting up the SHEMP system. All of the information found in this appendix can be found in the documentation included with the SHEMP source code.

A.1 SHEMP Overview

A.1.1 Quick Start

Use the shadmin/shadmin.pl script to set up the installation (e.g., mint/import certs, generate user accounts, etc.). Then go the appropriate directory—repository or shutil, and execute the script found in that directory.

A.1.2 SHEMP Background

The SHEMP system is the central component of my PhD dissertation. Very briefly, it is an attempt to allow users to use their private keys without having to store them on the desktop. SHEMP achieves this through the use of short-lived keypairs and Proxy Certificates.

SHEMP borrows concepts from many places. Many of the ideas found in SHEMP come from the Grid computing community. While it is not necessary, some familiarity with Proxy Certificates and the MyProxy credential repository would be useful. Second, SHEMP makes use of XACML, and in particular, the SunXACML project's implementation. Last, the SHEMP prototype for my dissertation was built on the Bear/Enforcer platform, and uses the TCPA/TCG hardware to protect secrets.

The project is written mostly in Java, and uses the Ant build system. If you just want to run the code, you need the Java 1.5 JRE and the openssl command line tool. If you plan to build the code yourself, you additionally need the JDK 1.5 or greater and the Ant build system. Lastly, the project uses the Java CoG grid toolkit, which is redistributed in the SHEMP packages.

A.1.3 The SHEMP distribution

The are essentially three flavors of SHEMP distribution to date: a client-only package, a repository-only package, and a package that contains both. The core SHEMP distribution contains the following:

```
certs/          This directory is initially left empty.  As the system
                is installed, certificates get placed in this
                directory.

doc/            Contains documentation for SHEMP.   You will find big
                picture documentation here, such as the SHEMP design
                and programmer's guide.  You will also find Javadoc
                documentation in the 'apidoc' subdirectory.

lib/            The jars that make SHEMP work.  In all distributions,
                you will find:

                cog-jglobus.jar   -- the CoG toolkit
                common.jar        -- SHEMP common classes
                jce-jdk13-117.jar -- the BouncyCastle JCE
                log4j-1.2.7.jar   -- Apache logging classes
```

```
                sunxacml.jar      -- Sun's XACML classes
                tools.jar         -- SHEMP tools classes

                If you have the repository package, you will have:

                repository.jar    -- the SHEMP repository classes

                If you have the client package, you will have:

                shapi.jar         -- the SHEMP API classes
                shutil.jar        -- a sample SHEMP application
shadmin/        This folder contains the SHEMP administration tool
                Perl script: shadmin.pl.  The tool is useful for
                minting certificates, and adding new machines/accounts
                to the system.  It also handles things like building
                the users' policy statements.

src/            Contains all of the source necessary to build SHEMP
                and an Ant buildfile.

tools/          Contains a number of command-line utilities developed
                in the course of making SHEMP.


The repository install contains the above directories, plus:

repository/     This directory holds the repository start script and
                a number of repository-specific subdirectories.


The client install contains the core directories, plus:

shutil/         This directory hold a sample SHEMP application called
                the SHEMP Utility.  It demonstrates how to use the
                SHEMP API (the Shapi class) to build an application
                that uses a SHEMP repository.
```

A.2 The SHEMP Repository

In SHEMP, the repository is where all of the magic happens. Before you run the script in this directory, you need to be sure that the repository is set up. The first thing to do is use the shadmin.pl tool to establish a Repository Administrator. Once an admin is recognized, the admin needs to issue a repository identity certificate and RAC (all handled by the 'add

repository' command in shadmin.pl). Last, the admin needs to add users to the system by using the 'add user' command in shadmin.pl. This command will generate a keypair for the user on the repository, generate a Certificate Signing Request for that keypair, and add a new user account on the repository for the user (the SHA1 password hashes for the user passwords are in ./etc/passwd).

Once the certificates and user accounts are set up, the Repository Admin can tweak some parameters in the ./conf/shemp.conf file. This file contains some sensitive information (like the keystore password), so it should be guarded with whatever is available: file perms, Bear/Enforcer security policy, etc.

Once everything is configured, execute the ./repository command, and the repository will start and bind to the specified port.

The ./keys directory contains two Java keystores: 'keystore' for the machine keystore and 'userstore' for the user keys. You can view these with the 'keytool', which is a part of the Java (JRE and JDK) distributions from Sun.

The ./utils directory contains a number of TPM utilities by IBM (as well as one extension by me: readpubek). These could be used to put the PUBEK hash in the repository's identity certificate, for example. Use you imagination.

A.3 The SHEMP Utililty (Shutil)

The Shutil is a small program to demonstrate how to build SHEMP applications by using the SHEMP API. Before you can run the Shutil, a Platform Administrator needs to set up the platform (via the 'add platform' command in the shadmin.pl tool). The setup involves establishing an identity certificate for the platform as well as the Platform Attribute Certificate which lists the platform's attributes.

Once added, the Platform Admin also needs to set up the scripts in the ./utils directory.

createTempKey.sh is executed when a user requests a Proxy Cert, the first step of which is to generate a temporary keypair on the platform. createTempKey.sh has the responsibility of creating that keypair. BE CAREFUL—harmful commands in that file will be executed. File system polymorphism is pretty cheesy, but the only way to generate TPM keys without writing a Java JNI driver was to use this script. destroyTempKey.sh should clean up the temporary key and Proxy Cert. This script is called when a user logs off. In the ./utils directory, you will also find tpmtools.tar.gz which contains a number of useful TPM tools from IBM (and one from me).

The last part of configuration involves tweaking the ./conf/shutil.conf file to suit your needs. This file should be protected by the best available methods (file perms, Bear/Enforcer security policy, etc.) as it contains sensitive information.

Once configured, run the ./shutil (or ./shutilGUI) script and you're off.

The ./keys directory contains the Java 'keystore' which holds the platform's keys—used in the platform authentication (i.e., client-side SSL) step.

A.4 The SHEMP Administration Tool (Shadmin)

This SHEMP Admin tool (shadmin.pl) is by far the most useful utility in the SHEMP distribution. It glues a number of programs together (openssl, keytool, acbuilder, and kup-builder) so that administrators (Certification Authorities (CAs), Platform Admins (PAs), and Repository Admins (RAs)) can get SHEMP up and running without having to deal with all of the tools individually. SHEMP uses quite a few certificates, so a centralized utility to mint and manage them is crucial. You can edit some of the global variables at the top of the script to suit your needs.

The general sequence of steps to set up SHEMP is as follows:

1. Set up a CA. shadmin.pl can use openssl to mint a self-signed cert. It creates a

subdirectory called 'ca' which contains the private key. Alternatively, you could use a real CA, in which case, you need to obtain the CA's cert (in PEM format) and place it in the 'certs' directory.

2. Set up a repository. This involves a number of sub-steps:

   - Generate a keypair for the RA, and get the RA's cert signed. shadmin.pl creates a directory called 'requests' which is used to store the Certificate Signing Requests (CSRs). The private key for the RA is in the 'radmin' subdirectory. The CSR needs to be given to the CA, or if using a self-signed CA cert, the CA menu contains a 'Certify Admin' option which will sign the RAs CSR, and put the signed cert in the 'certs' directory.
   - Add a repository. This step generates a keypair and CSR for the repository, as well as the Repository Attribute Certficate (RAC). The name for the repository should be the 'least spoofable' name possible (e.g., MAC address, TPM's public key hash, etc.). shadmin.pl will have the RA sign the repository's CSR and RAC, and will place the signed certificates in the 'certs' directory.
   - Add a user. shadmin.pl will generate a keypair and CSR for the new user, and place the CSR in the 'requests' subdirectory. The CA needs to sign this CSR, as well as generate a singed Key Usage Policy for the user. If you are using a self-signed CA cert, shadmin.pl will sign the CSR with the CA's private key, call the kupbuilder to set up the KUP for the user, sign the KUP with the CA's private key, and place the signed certificates in the 'certs' directory.
   - Import the signed user cert into the repository.

3. Set up a platform. This is very similar to the first two steps outlined above:

   - Generate a keypair for the PA, and get the PA's cert signed. The PA's CSR

gets written to the 'requests' subdirectory, and the private key for the PA is in the 'padmin' subdirectory. The CSR needs to be given to the CA, or if using a self-signed CA cert, the CA menu contains a 'Certify Admin' option which will sign the PAs CSR, and put the signed cert in the 'certs' directory.

- Add a platform. This step generates a keypair and CSR for the platform, as well as the Platform Attribute Certificate (PAC). The name for the platform should be the 'least spoofable' name possible (e.g., MAC address, TPM's public key hash, etc.). shadmin.pl will have the PA sign the platform's CSR and PAC, and will place the signed certificates in the 'certs' directory.

A.5 SHEMP Tools

In the 'tools' directory, you will find a number of tools which were developed during the course of working on SHEMP. All of the certificates are assumed to be in PEM format.

A.5.1 acbuilder

This is a command line utility used to make the attribute certificates (ACs) which describe platforms and repositories (i.e., the PACs and RACs). These ACs need to have the same subject DN as the machine's identity certificate. The program is called by the shadmin.pl tool each time a repository or platform is added. The program prompts the user for ¡attribute,value¿ pairs to add to the AC, and generates an AC which follows the attrcert.dtd DTD in the shadmin directory.

```
Usage: acbuilder <subject_cert_file> <issuer_cert_file> <output_file>

<subject_cert_file>     The name of a file which contains an X.509
                        identity certificate for the subject (which
```

```
                        should be either a platform or repository).
                        The DN is peeled out of the subject_cert_file
                        and placed in the AC.

<issuer_cert_file>      The name of a file which contains an X.509
                        identity certificate for the issuer (which
                        should be either a Platform Administrator or
                        a Repository Administrator.  The DN is peeled
                        out and placed in the AC.  The shadmin.pl
                        tool will sign the <envelope> element of the
                        AC with the private key of the issuer, and
                        place the signature in the <envelopeSignature>
                        element of the AC.

<output_file>           File to write the unsigned AC to.
```

A.5.2 cyptoaccessory

This is a little Swiss-army RSA crypto utility implemented in Java. It implements signing, verification, encryption, and decryption using the Java keystore and the RSA algorithm. In addition to being useful for testing, the time taken to perform the crypto operations serve as a baseline for SHEMP's performance testing. It is menu-driven; there are no command-line arguments.

A.5.3 extractkey

This is a very small program to take an X.509 certificate, pull the public key out, and write the base64-encoded key to a file. In SHEMP, this program's functionality is used to send a public key to the repository for inclusion in a Proxy Certificate.

```
Usage: extractkey <certfile> <keyfile>

<certfile>              Name of a file containing an X.509 certificate
                        in PEM format.

<keyfile>               Name of a file to write the base64-encoded
                        public key.
```

A.5.4 kupbuilder

This program is used to manufacture Key Usage Policies (KUPs) for SHEMP users. As with the acbuilder tool, this program is called by the shadmin.pl program directly. The core of the KUP is an XACML policy with the user as a Target, and up to three rules: generate (which governs Proxy Certificate generation), encrypt (which governs the use of the SHEMP encryption proxy service), and sign (which governs the SHEMP signing proxy service).

For each of the three operations (generate, encrypt, and sign), the program asks if it should construct a rule for the action. If so, it prompts for a number of ¡attribute,value,issuer_cert¿ 3-tuples. The issuer_cert should belong to either the Platform Administrator or Repository Administrator; DN is pulled out of the issuer_cert, and placed in the policy.

In English, the XACML policy states that, in order for the 'action' to be allowed, the entity in the 'issuer_cert' must say (in an AC) that the 'attribute' has the 'value'. In order for an action to be allowed, all of the 3-tuples must be satisfied.

Once, the XACML policy is created, it is placed in the KUP ¡envelope¿, the Certification Authority signs the ¡envelope¿, and the signature is placed in the ¡envelopeSignature¿ element. The result follows the kuptemplate.dtd DTD in the shadmin directory.

```
Usage: kupbuilder <subject_cert_file> <issuer_cert_file> <output_file>

<subject_cert_file>     The name of a file which contains an X.509
                        identity certificate for the subject (which
                        should be either a platform or repository).
                        The DN is peeled out of the subject_cert_file
                        and placed in the KUP.

<issuer_cert_file>      The name of a file which contains an X.509
                        identity certificate for the issuer (which
                        should be the Certification Authority).  The
                        DN is peeled out and placed in the KUP.  The
```

	shadmin.pl tool will sign the <envelope> element of the KUP with the private key of the issuer, and place the signature in the <envelopeSignature> element of the KUP.
<output_file>	File to write the unsigned KUP to.

Bibliography

[1] R. Anderson. Why Cryptosystems Fail. *Communications of the ACM*, pages 32–40, November 1994.

[2] T. W. Arnold and L. P. Van Doorn. The IBM PCIXCC: A New Cryptographic Coprocessor for the IBM eServer. *IBM Journal of Research and Development*, 48(3/4), 2004.

[3] G. Ateniese, K. Fu, M. Green, and S. Hohenberger. Improved Proxy Re-Encryption Schemes with Applications to Secure Distributed Storage. In *Network and Distributed System Security Symposium*. The Internet Society, 2005.

[4] S. Beattie, A. Black, C. Cowan, C. Pu, and L. Yang. CryptoMark: Locking the Stable door ahead of the Trojan Horse, 2000.

[5] D. Bell and L. LaPadula. Secure Computer Systems: Unified Exposition and Multics Interpretation. Technical Report ESD-TR-75-306, The MITRE Corporation, March 1976.

[6] Berkeley DB. `http://www.sleepycat.com`.

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

[8] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The Role of Trust Management in Distributed Systems. *Secure Internet Programming*, 1603:185–210, 1999. Lecture Notes in Computer Science.

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

[10] 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.

[11] D. Bryson. The Linux Crypto API - A User's Perspective. `http://www.kerneli.org/howto/index.php`, May 2002.

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

[13] Buyer's Guide - Web Portal Security Solution. `http://www.entrust.com/resources/pdf/buyers_guide.pdf`, November 2001.

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

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

[16] D. Cooper. A Model of Certificate Revocation. In *15th Annual Computer Security Applications Conference (ACSAC '99)*, pages 256–264, Phoenix, Arizona, USA, December 1999. IEEE Computer Society.

[17] X. Ding, G. Tsudik, and D. Mazzocchi. Experimenting with Server-Aided Signatures. In *Network and Distributed System Security Symposium*, 2002.

[18] 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.

[19] C. Ellison. The nature of a usable pki. *Computer Networks*, pages 823–830, 1999.

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

[21] C. Ellison. Improvements on Conventional PKI Wisdom. In *Proceedings of the 1st Annual PKI Research Workshop*, April 2002.

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

[23] 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.

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

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

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

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

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

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

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

[31] E. Felten, D. Balfanz, D. Dean, and D. Wallach. Web Spoofing: An Internet Con Game. In *20th National Information Systems Security Conference*, 1997.

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

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

[34] N. Goffee, S. Kim, S.W. Smith, P. Taylor, M. Zhao, and J. Marchesini. Greenpass: Decentralized, PKI-based Authorization for Wireless LANs. In *3rd Annual PKI Research and Development Workshop*. NIST, April 2004.

[35] 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`.

[36] S. Haber and W.S. Stornetta. How to Time-Stamp a Digital Document. *Lecture Notes in Computer Science*, 537, 1991.

[37] J. Hamilton. *CGI Programming 101*. CGI101.com, 1999.

[38] J. Heimberg. Use of Middleware. `www.securecourse.com/WP-SC-Middleware-Case.pdf`, 2002.

[39] S. Henson. Netscape Certificate Database Info. `http://www.drh-consultancy.demon.co.uk/cert7.html`.

[40] S. Henson. Netscape Key Database Format. `http://www.drh-consultancy.demon.co.uk/key3.html`.

[41] R. Housley, W. Polk, W. Ford, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. IETF RFC 3280, April 2002.

[42] J. Howell and D. Kotz. A formal semantics for SPKI. In *Proceedings of the Sixth European Symposium on Research in Computer Security (ESORICS 2000)*, volume 1895 of *Lecture Notes in Computer Science*, pages 140–158. Springer-Verlag, October 2000.

[43] P. Huy, G. Lewis, and M. Liu. Beyond the Black Box: A Case Study in C to Java Conversion and Product Extensibility. Technical report, Carnegie Mellon Software Engineering Institute, 2001.

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

[45] IBM Watson Global Security Analysis Lab. TCPA Resources. `http://www.research.ibm.com/gsal/tcpa`.

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

[47] Implementing Web Site Client Authentication Using Digital IDs and the Netscape Enterprise Server 2.0. `http://www.verisign.com/repository/clientauth/ent_ig.htm#clientauth`.

[48] Information About the Clear SSL State Option in Windows XP. `http://support.microsoft.com/?kbid=290345`. Microsoft Knowledge Base Article - 290345.

[49] Installing DoD PKI Class 3 Certificates into Windows 2000 for use with Microsoft Products, May 2003.

[50] D. Ireland. BigDigits multiple-precision arithmetic source code. `http://www.di-mgt.com.au/bigdigits.html`.

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

[52] 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.

[53] K. Kain, S.W. Smith, and R. Asokan. Digital Signatures and Electronic Documents: A Cautionary Tale. In *Advanced Communications and Multimedia Security*. Kluwer Academic Publishers, 2002.

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

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

[56] P. Kocher. On Certificate Revocation and Validation. In *Proceedings of the Second International Conference on Financia l Cryptography (FC'98)*, volume 1465 of *LNCS*, pages 172–177. Springer-Verlag, 1998.

[57] 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.

[58] 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.

[59] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.

[60] N. Li, B. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security (TISSEC)*, 6(1):128–171, February 2003.

[61] N. Li and Feigenbaum J. Nonmonotonicity, User Interfaces, and Risk Assessment in Certificate Revocation. In *Proceedings of the 5th International Conference on Financial Cryptography*, pages 166–177. Springer-Verlag, 2002.

[62] N. Li, W. Winsborough, and J. Mitchell. Beyond Proof-of-compliance: Safety and Availability Analysis in Trust Management. In *Proceedings of 2003 IEEE Symposium on Security and Privacy,*, pages 123–139. IEEE Computer Society Press, May 2003.

[63] 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.

[64] 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.

[65] M. Blaze and J. Feigenbaum and J. Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, May 1996.

[66] M. Blaze and J. Feigenbaum and M. Strauss. Compliance-checking in the PolicyMaker Trust Management System. In *Proceedings of Second International Conference on Financial Cryptography (FC '98)*, volume 1465, pages 254–274, 1998.

[67] 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.

[68] A. Malpani, S. Galperin, M. Mayers, R. Ankney, and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol. RFC2560, `http://www.ietf.org/rfc/rfc2560.txt`, June 1999.

[69] J. Marchesini. Secure Hardware Enhanced MyProxy. Technical Report TR2004-525, Dartmouth College, November 2004.

[70] 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.

[71] J. Marchesini and S.W. Smith. SHEMP: Secure Hardware Enhanced MyProxy. Technical Report TR2005-532, Department of Computer Science, Dartmouth College, 2003.

[72] 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.

[73] J. Marchesini, S.W. Smith, O. Wild, J. Stabiner, and A. Barsamian. Open-Source Applications of TCPA Hardware. In *20th Annual Computer Security Applications Conference (ACSAC)*, December 2004.

[74] J. Marchesini, S.W. Smith, and M. Zhao. Keyjacking: Risks of the Current Client-side Infrastructure. In *2nd Annual PKI Resarch Workshop*. NIST, April 2003.

[75] J. Marchesini, S.W. Smith, and M. Zhao. Keyjacking: The Surprising Insecurity of Client-Side SSL. *Computers and Security*, 2004. In Press.

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

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

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

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

[80] Microsoft Authenticode Developer Certificates. http://www.thawte.com/getinfo/products/devel/msauthenticode.html.

[81] Mozilla. NSS Security Tools. http://www.mozilla.org.

[82] M. Naor and K. Nissim. Certificate Revocation and Certificate Update. *IEEE Journal on Selected Areas in Communications*, 18(4):561–570, April 2000.

[83] 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.

[84] 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.

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

[86] Netscape Communications Corp. *Command-Line Tools Guide: Netscape Certificate Management System*, 4.5 edition, October 2001.

[87] J. Niederst. *Web Design in a Nutshell (2/E)*. O'Reilly, 2001.

[88] 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.

[89] Offline NT Password & Registry Editor. http://home.eunet.no/~pnordahl/ntpasswd.

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

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

[92] OS Security, Inc. Round One: "DLL Proxy" Attack Easily Hijacks SSL from Internet Explorer. http://www.securityfocus.com/archive/1/353203/2004-02-09/2004-02-15/2.

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

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

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

[96] Personal Certificates. http://www.thawte.com/html/COMMUNITY/personal/.

[97] M. Pietrek. Under The Hood. *Microsoft Systems Journal*, February 2000.

[98] B. Pinkas and T. Sander. Securing Passwords Against Dictionary Attacks. In *ACM Computer and Communications Security*, 2002.

[99] Preventing HTML Form Tampering. http://advosys.ca/papers/form-tampering.html, August 2001.

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

[101] Pubcookie: Open-source Software for Intra-institutional Web Authentication. http://www.washington.edu/pubcookie/.

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

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

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

[105] D. Safford, J. Kravitz, and L. van Doorn. Take Control of TCPA. *Linux Journal*, pages 50–55, August 2003.

[106] 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.

[107] R. Sandhu, M. Bellare, and R. Ganesan. Password Enabled PKI: Virtual Smartcards vs. Virtual Soft Tokens. In *1st Annual PKI R&D Workshop*. NIST, 2002.

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

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

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

[111] RSA Security. RSA ClearTrust Data Sheet. www.rsasecurity.com/node.asp?id=1186, 2005.

[112] Security Focus. The BugTraq Mailing List. www.securityfocus.com.

[113] 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.

[114] 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.

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

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

[117] S.W. Smith. *Trusted Computing Platforms: Design and Applications*. Springer, 2004.

[118] 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.

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

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

[121] L. Stein and J. Stewart. The World Wide Web Security FAQ. http://www.w3.org/Security/Faq/, February 2002.

[122] 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.

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

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

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

[126] 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.

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

[128] L. van Doorn, G.Ballintijn, and W. Arbaugh. Signed Executables for Linux. Technical Report UMD CS-TR-4259, University of Maryland, June 2001.

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

[130] G. von Laszewski, J. Gawor, P. Lane, B. Alunkal, and M. Hategan. Globus Java CoG Kit. www.cogkit.org.

[131] S. Weeks. Understanding Trust Management Systems. In *Proceedings of 2001 IEEE Symposium on Security and Privacy*, pages 94–105. IEEE Computer Society Press, May 2001.

[132] 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.

[133] 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 Research and Development Workshop*, April 2004.

[134] A. Whitten and J.D. Tygar. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. In *USENIX Security*, 1999.

[135] Windows Prompts You for Your Password Multiple Times When You Use Outlook If Strong Private Key Protection Is Set to High. http://support.microsoft.com/?kbid=821574. Microsoft Knowledge Base Article - 821574.

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

[137] E. Ye and S.W. Smith. Trusted Paths for Browsers. In *USENIX Security*, 2002.

[138] E. Ye, Y. Yuan, and S.W. Smith. Web Spoofing Revisited: SSL and Beyond. Technical Report TR2002-417, Department of Computer Science, Dartmouth College., 2002.

[139] 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.

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

[141] K.-P. Yee. User Interaction Design for Secure Systems. `http://zesty.ca/sid/uidss-may-28.pdf`.

Index