Access Control for Ad-hoc Collaboration

Dirk Balfanz

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

January 2001

Abstract

With the advent of networks that span administrative domains, increasing mobility, and even global-area networks, we find ourselves more and more often in situations where we do not know the potential parties accessing our computer systems. Yet, we choose to collaborate with those parties: For example, we frequently browse unknown Web sites, or invite unknown clients to access our servers. We call a scenario in which parties choose to collaborate that do not necessarily trust each other, or even know each other, an *ad-hoc collaboration*.

This dissertation investigates how we can protect our sensitive resources in the presence of ad-hoc collaboration. In particular, we study three ad-hoc collaboration scenarios and propose novel access control schemes for each of them. In our first system we propose and implement an access control mechanism for distributed Java applications that can span administrative domains. It uses an access control logic to allow servers to reason about the access privileges of unknown clients. Our second system presents a simple security model for the personal computer, in which the user's workstation is divided into multiple desktops. Each desktop is sealed off from the others, confining the possibly dangerous results of ad-hoc collaboration. Our last system investigates ad-hoc collaboration with hand-held computers. We present a framework that allows developers to write "split applications": Part of the application runs on a trusted, but computationally limited, small computer, and part of the application runs on an untrusted, but more powerful PC.

Acknowledgments

There are a number of people that collaborated, contributed, helped, critiqued, encouraged, and led the way. Although I cannot name all of them here, I would like to take this space to acknowledge a few of them: Dan Simon of Microsoft Research came up with the WindowBox idea, and Paul England always had the right pointers about Windows programming. Drew Dean at Xerox PARC collaborated with me on the Placeless access control logic. Other folks at PARC, most notably Doug Terry, Jim Thornton and Mike Spreitzer were also of great help. Ian Goldberg ported SSLeay to the PalmPilot, enhanced Copilot, and provided valuable Pilot programming tips. Bob Relyea at Netscape provided help with some PKCS#11 details. Andrew Appel helped a great deal with the decidability proof for the logic in Chapter 2.

Significant parts of the work presented in this dissertation were done while visiting Microsoft Research and Xerox PARC. I am grateful for having been able to work at, and be supported by, these splendid institutions.

Last, but not least, I would like to thank my advisor Edward Felten. He always had the right idea at the right time, and consistently provided me with invaluable feedback and guidance. Large portions of this work were funded under NSF grant SBR-9729447.

Contents

Abstract . . . iii

1 Introduction 1

1.1 Ad-hoc Collaboration . . . 3

1.2 Access Control . . . 4

1.3 Thesis Contributions . . . 5

2 A Security Infrastructure for Distributed Java Applications 7

2.1 Introduction . . . 7

2.2 Overview . . . 10

2.2.1 The Placeless Documents System . . . 10

2.2.2 Authentication . . . 12

2.2.3 Access Control . . . 12

2.3 The Access Control Logic . . . 14

2.3.1 Syntax . . . 16

2.3.2 Inference Rules . . . 17

2.3.3 Access Control as Proof Finding . . . 20

2.3.4 An Example . . . 21

2.3.5 Semantics . . . 23

2.3.6 Soundness and Consistency . . . 26

2.3.7 Decidability 27

2.4 The Implementation 28

2.4.1 Statements 28

2.4.2 Principals 28

2.4.3 Permissions 30

2.4.4 Policies 33

2.4.5 Certificates 35

2.4.6 Inference Engine 35

2.4.7 Running RMI over SSL 36

2.4.8 Tools 37

2.5 Experience with Access Control Logics 39

2.6 Future Work 41

2.7 Conclusion 43

3 WindowBox: A Simple Security Model for the Connected Desktop 46

3.1 Introduction 46

3.2 The Threat of Ad-hoc Collaboration 48

3.3 Existing Security Mechanisms 50

3.3.1 Access Control Lists 50

3.3.2 Application-Level Security 51

3.3.3 Firewalls and Proxies 52

3.3.4 Sandboxing 53

3.4 A Model For Sandboxing-Based Security 54

3.4.1 The Multi-Desktop Premise 55

3.4.2 The WindowBox Security Model 56

3.4.3 Examples 57

3.4.4 How WindowBox Protects the User 60

3.5 Implementation . . . 61
3.5.1 The Windows NT Security Architecture . . . 62
3.5.2 Desktops as User Groups . . . 64
3.5.3 Confined Objects . . . 66
3.5.4 Restricting Network Access . . . 68
3.5.5 Security Analysis . . . 69
3.6 Related Work . . . 71
3.7 Conclusions . . . 73

4 Splitting Trust **75**
4.1 Introduction . . . 75
4.2 PilotKey . . . 77
4.2.1 Overview . . . 78
4.2.2 PKCS#11 . . . 80
4.2.3 The Implementation . . . 84
4.2.4 Protecting Sensitive Data . . . 87
4.3 The Splitting Trust Framework . . . 89
4.3.1 Writing Splitting-Trust Applications . . . 92
4.3.2 The SJava Class Library . . . 95
4.3.3 An SJava Example . . . 100
4.3.4 Security Considerations of SJava . . . 106
4.4 Conclusions . . . 108
4.5 Related Work . . . 108

5 Conclusions **111**

A Proofs of Inference Rules **116**

B Proof of Decidability **127**

B.1 Restricting the Logic 127

B.2 A Proof-Finding Algorithm 132

B.3 Soundness and Completeness of the Algorithm 134

B.4 A Decision Procedure for $\mathfrak{L}$ 136

List of Figures

2.1 Placeless Documents 11

2.2 Impersonation in Placeless is similar to a quoting chain in ABLP logic. . . . 13

3.1 A sample access token 62

3.2 A sample DACL 63

3.3 A sample access token with desktop SIDs 64

3.4 A sample access token of a desktop process 65

4.1 A private RSA key 81

4.2 Overview of our PKCS#11 implementation 83

4.3 Information flow for PIN, contrasting traditional smart card and PalmPilot. . 85

4.4 Information flow for reading encrypted e-mail, contrasting traditional smart card and PalmPilot. 86

4.5 A Java application as a graph 92

4.6 Java application after a split 94

4.7 Sample calendar application 100

4.8 Moving the GUI to the untrusted platform is not practical. 101

4.9 Design of the calendar application. 102

Chapter 1

Introduction

The field of computer security has followed the development of digital computers throughout the (short) history of computing. As computers and their uses changed, so did our understanding of a “secure” computing experience. Protection was introduced to time-sharing systems mostly to achieve higher reliability — when computations were protected from each other, it was less likely that a failure in one computation could affect others. Users, therefore, were safe from undesired side effects of faulty programs. It was only later that privacy and other concerns in multiuser systems led to the introduction of *access control*. Now, users could be sure that information they owned was safe from unauthorized access or manipulation by other users of the system.

With the advent of the personal computer, security became less of a concern. A faulty program bringing down a personal computer did not affect other users (because there were none), and the fact that a personal computer was usually outside the reach of potential intruders guaranteed privacy and integrity of the information stored on the computer. It should be noted that the use of computers in other fields, for example the military or certain high-stakes business areas, continued to influence the development of computer security. However, those advances, such as mandatory access control, high-assurance or real-time systems, intrusion-detection mechanisms, and others, failed to have an impact on the personal computer.

When personal computers became networked (first by magnetic tapes or disks changing hands, later by actually connecting computers), things started to change. Now in addition to reliability and privacy, being safe from viruses or network attacks was also part of a “secure computing experience.” More recently, even other threats emerged: hostile mobile code, e-mail viruses, Web spoofing, denial-of-service attacks, on-line fraud, *etc.* New mechanisms needed to be developed (or adapted for the mainstream) to address these new issues.

Traditional (pre-personal-computer) security concerns also reemerge as more and more computers become connected with each other, and therefore can no longer claim to be “personal” — a computer on the Internet is a potential target for break-ins. Its physical location can no longer guarantee the safety of the information stored on it. These “old” concerns, however, usually emerge on a scale vastly different from their original occurrence. Cryptography, for example, was one of the first techniques found to be useful for protecting privacy on the Internet. However, because of the size of the Internet, it has become necessary to adopt public-key infrastructures (which scale better to large systems), rather than traditional secret-key systems. Network attacks can now be launched from such a large number of points in the network that traditional solutions (packet filtering, back-tracing of network packets, *etc.*) will no longer suffice. By the same token, the number of potential users of a system (say, a Web sever) is so big that traditional access control mechanisms fail. In fact, *every* person on the Internet is a potential user of any given Web site.

1.1 Ad-hoc Collaboration

One of the recent trends in the mainstream computing world is that two (or more) parties *that do not know, or fully trust, each other* choose to *collaborate*. We call this phenomenon *ad-hoc collaboration*.

In this dissertation, the term “collaboration” shall mean that two (or more) computers *together* provide a computing experience for a user. For example, when a user visits a site on the World Wide Web, the client computer and the Web server computer collaborate to deliver the requested information to the user. The various terminals and back-end servers in a UNIX cluster collaborate to facilitate a login session, file-system access, and more, for the users of the cluster. When two PalmPilot users exchange their contact information by means of an infrared link, the two PalmPilots collaborate in order to facilitate the information exchange. In particular, the term “collaboration” as used here is broader than “on-line collaboration,” which usually refers only to software packages for work collaboration in an office or business environment.

Collaboration of computers is *ad-hoc* when the computers involved have not collaborated before, or for some other reason do not share administrative information about each other. From the examples above, browsing a Web site is ad-hoc, since the Web server and the client do not need to be prepared in advance to enable the browsing. If, however, the Web server wanted to authenticate its clients, then the collaboration would not be ad-hoc. In this case, the Web server had to have certain information about the client (probably its public key, certificates, and a list of rights that the client can exercise) before the collaboration could start. By the same token, a UNIX cluster does not constitute ad-hoc collaboration. Exchanging contact information between two PalmPilots, on the other hand, is ad-hoc because the two particular PalmPilots do not have to be introduced to each other before the collaboration can start.

Ad-hoc collaboration is an intrinsic feature of open systems. In an open system, new parties can join at any time and start communicating with other parties in the system. Therefore, computers must be prepared to accept collaboration requests from parties with whom they have never collaborated before.

1.2 Access Control

Access control is qualified protection. Resources need to be protected from unauthorized access, but they are open to certain, permitted, types of access. An access control system must be able to distinguish the different kinds of access and must enforce the *policies* that govern access.

Access control policies are usually used to answer the question “who is allowed to do what,” since that is a natural way of describing the different kinds of (permitted or forbidden) access. In fact, the model of an *access control matrix* [32] lies at the heart of most access control systems. In an access control matrix, there is a row for each user of the system, and a column for every resource that should be protected. The cells of the matrix describe what kind of actions are permitted. However, in the presence of ad-hoc collaboration, the question of “who is allowed to do what” cannot always be answered, since there is no complete list of the potential parties accessing the system. Imagine a Web site trying to specify which user is allowed to view which documents. This task cannot be accomplished because the Internet is an open system — a list of all users that may access our Web site cannot exist. To put it another way, because browsing a Web site is often an ad-hoc collaboration, we cannot apply traditional access control methods.

To summarize, the purpose of access control is to protect resources in a qualified manner — some access is allowed, while other access is forbidden. In the presence of ad-hoc collaboration, we cannot apply the traditional model of an access control matrix to answer the question “who is allowed to do what,” because the list of potential accessors cannot be known.

Therefore, we need to look for new mechanisms to allow access control in the presence of ad-hoc collaboration.

1.3 Thesis Contributions

As explained above, ad-hoc collaboration provides new challenges for access control mechanisms. The model of an access control matrix – and more fundamentally, the underlying assumption that we can know all the possible *who*s when posing the question “who is allowed to do what” – is no longer satisfactory for a world in which we have to protect our resources in open systems, and during ad-hoc collaborations.

This dissertation investigates the problem of access control in the presence of ad-hoc collaboration. We do not give a “one-size-fits-all” solution. Rather, we provide a number of mechanisms and recipes that can be applied to different ad-hoc-collaboration scenarios. To be sure, we are not the first to consider access control mechanisms that go beyond the model of the access control matrix. Capability-based systems [33], Kerberos-style systems [44], or trust-management systems [11, 21], for example, can all handle some form of ad-hoc collaboration. However, the problem of access control for ad-hoc collaboration has not been investigated this extensively before. In particular, we focus on three quite different problems (which, however, all have in common that they need access control for ad-hoc collaboration), and provide novel access control mechanisms. While the problems are specific in nature, our solutions can be applied to a wider range of ad-hoc-collaboration scenarios.

Chapter 2 discusses the use of access control logics for ad-hoc collaboration. Access control logics allow parties to reason about accessors’ privileges, even if the accessor is unknown to the owner of the resources. While we do not claim to invent access control logics, we do introduce a new access control logic that is simple, yet advances on some of the previous work on access control logics. At the same time, our access control logic is efficiently implementable in Java, and is therefore well suited for distributed Java applications such as the Placeless Documents System [15, 18] from Xerox PARC.

In Chapter 3 we introduce a radically new security model that is well suited for users who want to protect themselves from the potentially dangerous results of ad-hoc collaboration. Instead of specifying access control policies, under this model users compartmentalize their work and can thus confine the results of virus attacks, hostile mobile code, *etc.*

Finally, in Chapter 4 we introduce the concept of *splitting trust*. We explain how computers can delegate parts of computations to untrusted or unknown servers, while still protecting their sensitive resources. Chapter 4 investigates this concept by way of looking at small, slow, but trusted hand-held computers such as the PalmPilot, and big-screen, fast, untrusted PC's such as PC's found in an airport lounge or other public places. As we will point out, though, the concept of splitting trust can be applied in many more situations.

Chapter 2

A Security Infrastructure for Distributed Java Applications[1]

2.1 Introduction

In this chapter, we describe a distributed access control system in which servers can reason about access privileges of unknown clients.

Traditionally, authorization in a client-server system happens on the server. For example, the server has a list with all the clients that are allowed to connect, or has access control lists attached to the resources it needs to protect. This approach is not very scalable, since every new client needs to be introduced to the server before it can start using the service. Ad-hoc collaboration is impossible.

Capability-based systems represent a different approach [45, 33]. Here, the server hands out unforgeable capabilities to clients, but no longer maintains a list of who is allowed to do what. If a client can present a capability to the server, it will grant access to that client. Clients are free to pass on capabilities to other clients, thus allowing clients formerly unknown to the server to collaborate with it.

[1]This chapter is in part based on a previously published paper [5].

While allowing ad-hoc collaboration, the fundamental problem with capability-based systems is that the server has lost all control over who has access to its resources. It has to trust fully the clients to which it hands out capabilities not to pass them on to malicious third parties.

The solution to this problem are trust management systems [11, 21], in which servers can delegate privileges to clients, and can also specify exactly what those clients can do with the privileges they received (*e.g.* whether or not those privileges may be delegated further).

The tradeoff in trust management systems is often between expressibility and practicality. For example, the original PolicyMaker system could express a vast variety of policies, but it was not very practical (policies had to be implemented as programs in a safe version of AWK). On the other extreme, traditional capability-based systems are simple and practical, but the policies expressible in them tend to be all-or-nothing (once a capability is handed out, we lose all control over it).

SDSI/SPKI [21] strikes an interesting middle ground. However, despite some effort to formalize some aspects of the system [28, 1], there does not exist a formal description of SDSI/SPKI. Formal descriptions of trust management systems, however, are very desirable: Once we adopt a distributed trust management system, it becomes hard to reason about the consequences of certain actions. If various parties can delegate arbitrary privileges to each other (including the right to delegate privileges as the result of events), then it can become difficult to figure out exactly what the consequences of our actions are. Consider the following example: I have a rule in my system specifying that whoever has permission both to read and write a document automatically is allowed to delete that document. I then delegate the right to read a certain document to Bob, and delegate the right to write to that document to Alice, giving her the right to delegate that right to other people. Let us assume Alice delegates that right to the group "Managers." Now the moment Bob becomes a manager, he will be allowed to delete my document, which is not obvious from either me giving him read permission or his joining the ranks of management.

Therefore, various formalizations of access control mechanisms exist [2, 4], which allow us to reason in an abstract manner about access control.

In this chapter we present the design and implementation of a trust management system that is expressive enough to allow for ad-hoc collaboration, but at the same time relates to a very simple access control logic. Although our trust management system is generic in nature, it was developed for the Placeless Documents System, a large, Java-based, distributed middleware package [15, 18], and is written in Java itself.

Our work is inspired by SDSI/SPKI [21], but has a few twists of its own.[2] We define a logic for access control, such that access is granted iff a proof that it should be granted is derivable in the logic. In our system, requesters have to supply credentials that lead to a proof that a request is valid.

Our logic supports linked local name spaces, privilege delegation across administrative domains, and attribute certificates. This is in the spirit of more recent work, which emphasizes attribute certificates as a more scalable alternative to X.509-style identity certificates [20].

In our system, entities communicate through encrypted and authenticated network connections (we use the Secure Socket Layer protocol, SSL [23]). Because these connections are authenticated and protected from possible third-party tampering, we treat the connections as channels through which principals can "speak" (in the sense of ABLP logic [2]).

The rest of this chapter is structured as follows: In Section 2.2, we give an overview of the Placeless Documents System and of our overall design. In Section 2.3 we formally introduce our access control logic. This chapter should be understandable even if the reader skips the more formal aspects of Section 2.3 at first reading, as many aspects of the logic are

[2]We also ran into the practical difficulty that when we began our work, no satisfactory implementation of those systems was available in Java.

reintroduced, in a less formal way, in Section 2.4, where we present the implementation of our system. In Section 2.5, we discuss our experience working with access control logics. In Section 2.6, we discuss future work, and Section 2.7 concludes.

2.2 Overview

2.2.1 The Placeless Documents System

The Placeless Documents System is a distributed document management system developed at Xerox PARC [15]. Its major features include

- the ability to manage seamlessly documents of different kinds (*e.g.* files, e-mail messages, Web pages, *etc.*), and
- a query language over document *properties* (as opposed to a directory hierarchy in which documents are stored), which allows to retrieve specific documents from their storage location.

For the purposes of this dissertation, we focus on the following properties of the Placeless Documents System and its current implementation:

- It is a heavily distributed system. It is implemented in Java. The various components of the system communicate via the Java Remote Method Invocation protocol (RMI).
- Every user of the system runs one *kernel*, which manages the *document space* for that user. A space is the conceptual entity that holds all of a user's documents. A kernel is the actual program that implements the space abstraction.[3]

[3]Actually, a user can run more than one kernel. However, there is always a standard kernel, or *gatekeeper*, that lets the user access his or her space.

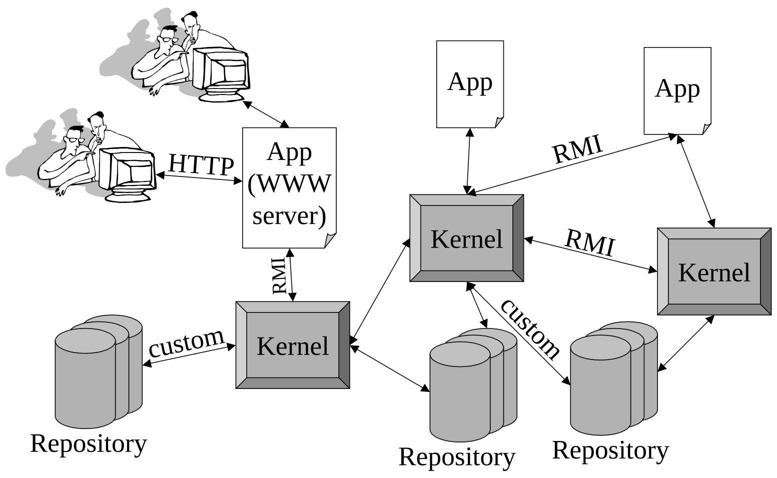

Figure 2.1: Placeless Documents. Kernels communicate with each other, and with Placeless applications, via RMI. Some applications may be used by multiple users at the same time. Kernels might need to manage documents owned by different users.

- Applications that would like to use the Placeless Documents System act as clients of kernels. All communication with kernels is via Java RMI. Kernels can also communicate with each other, if it turns out that a document is managed by another kernel than the one it was originally requested from.

- The document contents are not stored inside Placeless. Instead, Placeless stores document contents in their natural *repositories*, *e.g.* files on the file system, Web pages on a Web server, and so forth.

- Every document is managed by exactly one kernel. If a different kernel asks for access to that document, the other kernel must access the document through the document's managing kernel. That managing kernel is responsible for enforcing access control.

Typical Placeless applications include browser-like graphical interfaces that let users manage their (and others') documents. There are also bridges between Placeless and, for example, the World Wide Web, enabling users to access Placeless documents with any

Web browser. Note that in this last case, different users use the same Placeless application (which acts as a Web server at the front end and connects to Placeless kernels at the back end) to access their respective documents.

Figure 2.1 shows the various components in the Placeless Documents System and how they interact.

2.2.2 Authentication

To facilitate authentication of the various Placeless components to each other, we changed Java RMI to use the Secure Socket Layer (SSL) [23] as the underlying transport. This gives us two properties:

1. We assure the confidentiality and integrity of the communication between two components.

2. Components can authenticate each other, *i.e.* they learn at least the public key of the party they are communicating with. The fact the we require authentication does not preclude ad-hoc collaboration. What SSL authentication yields is (in our case, at least) that both connecting parties know each other's public key. We do *not* require that those public keys should be known beforehand to the parties involved, nor do we require that the parties know who owns these keys.

2.2.3 Access Control

Every kernel has a policy that specifies what the various principals are allowed to do with documents managed by that kernel. A policy is a set of *statements*. There are statements about which names are used for which principals, and about what kind of privileges are extended to those principals. The statements are actually expressions of an access control logic (see Section 2.3). When a client requests an operation to be performed in a kernel,

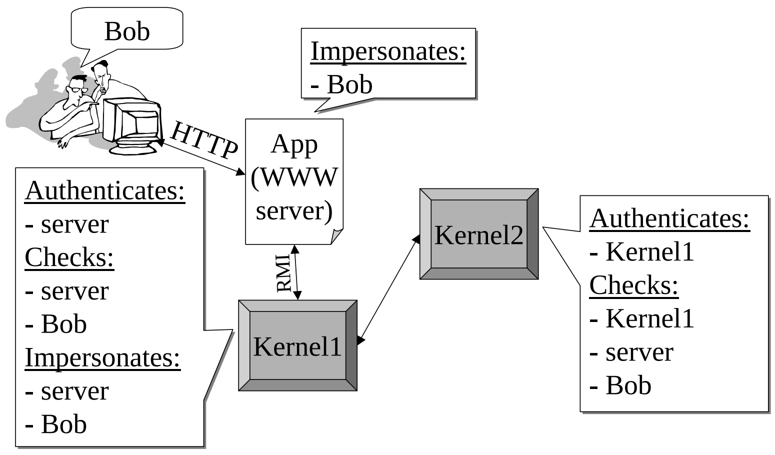

Figure 2.2: Impersonation in Placeless is similar to a quoting chain in ABLP logic.

that kernel knows the public key of the requesting client, and what operation is to be performed, because the request came over an authenticated channel. The request then is also expressed as a statement of said access control logic, using the authenticated public key of the requesting client as the principal making the request. An inference engine can then verify whether the policy supports that request.

A Placeless client can act on behalf of principals other than the principal running the client. Clients can simply announce on whose behalf they act. We often call this *impersonation* (see Figure 2.2). Clients are free to lie about on whose behalf they are acting. The kernel performing the operation will authenticate the client, and will then make sure that both the client and the principal it impersonates have enough privileges to perform the operation. In fact, if a kernel gets a request that is already identified as an impersonation and needs to turn around to another kernel to have that request answered, then the other kernel will have to check three principals: The kernel it just authenticated, the principal that kernel acts on behalf of, and the principal the first kernel acts on behalf of. This is similar to a quoting chain "A says that B says that C says that X" in ABLP logic [2]. There, too, we know that A is making the statement, but we do not know whether A is lying, *i.e.* whether B really said what A is claiming. Figure 2.2 illustrates this approach.

The fact that we do not authenticate principals other than the immediate peer of an SSL connection does not represent a security hole, since lying does not allow an attacker to gain any advantage: If a client *A* announces (erroneously) that it is impersonating principal *B*, then it will be able to perform only actions that both *A* and *B* are authorized to perform. A client cannot lie about who it is, it can only lie about who it impersonates. We provide the ability to impersonate other principals merely as a form of the principle of least privilege. Presumably, the Web server in Figure 2.2 has a lot of privileges. It impersonates other principals to make sure that it is only exercising privileges that the impersonated principal also has.

2.3 The Access Control Logic

We have defined an access control logic that we use for Placeless[4]. The logic heavily borrows from SDSI and SPKI [21] and can be seen as (yet another) attempt to formalize systems of that kind. It is also influenced by ABLP-style [2] logics and tries to fit naturally with the Java `Permission` classes, which is the reason why we did not follow the SDSI/SPKI approach to the letter.

Expressions of the logic are *statements*. A statement is of the form

$$\text{Principal} : \text{Permission}.$$

We can interpret a statement as meaning "the principal making the statement asserts that it is ok to perform the action associated with the permission mentioned in the statement." So, the statement

$$\text{Bob} : \text{Read}$$

[4]We should note that the access control logic can be used in more general settings as well, it is not particularly customized for Placeless.

says that Bob wants to read a document, while the statement

$$\text{Self} : \text{Read}$$

means that it is ok to read the document. The colon can roughly be pronounced as "says" with similar meaning as in ABLP logic. The principal "Self" is the principal that is making the access control decision, and whose resources should be protected.

Policies are expressed as a list of statements. As we will see later, the statement

$$\text{Self} : \text{Delegate}(\text{Bob}, \text{Read})$$

means that we give Bob read permission. The statement

$$\text{Self} : \text{Bind}(\text{Mother's Brother}, \text{Uncle})$$

means that we locally bind the name "Uncle" to any principal that is bound to the name "Brother" in the name space of the principal we call "Mother." The observant reader will notice in the next section that the above statement is syntactically at odds with the definition of our logic. Correctly, the statement above should read

$$\text{Self} : \text{Bind}(\text{Self's Mother's Brother}, \text{Self's Uncle})$$

However, as we will see in Section 2.4.2, the former statement is valid in our implementation of the system, and will be translated into the latter statement before it is passed to the logic inference engine. In our implementation (and sometimes also in this dissertation), we use the former statement simply as an abbreviation for the latter.

2.3.1 Syntax

We now formally present the access control logic on which our system is based. The following states the syntax of legal expressions in our logic in an EBNF-like notation:

$$
\begin{array}{rcl}
\mathit{expr} & ::= & \mathit{stmt} \mid \mathit{impl}. \\
\mathit{stmt} & ::= & \mathit{Principal}\ \text{“:”}\ \mathit{Permission}. \\
\mathit{impl} & ::= & \mathit{Permission}\ \text{“}\Rightarrow\text{”}\ \mathit{Permission}. \\
\mathit{Principal} & ::= & \text{“Self”} \\
 & \mid & \text{“AnyPrin”} \\
 & \mid & \mathsf{Key} \\
 & \mid & \mathit{ReferencePrincipal}. \\
\mathit{ReferencePrincipal} & ::= & \mathit{Principal}\text{“’s”}\quad \mathsf{String}. \\
\mathit{Permission} & ::= & \text{“Bind(”}\ \mathit{Principal}\ \text{“,”}\ \mathit{Principal}\ \text{“)”} \\
 & \mid & \text{“Delegate(”}\ \mathit{Principal}\ \text{“,”}\ \mathit{Permission}\ \text{“)”} \\
 & \mid & \text{“Read”} \\
 & \mid & \text{“Write”} \\
 & \ldots &
\end{array}
$$

Key can be any kind of cryptographic (public) key, and String is a string of characters, such as Bob, Alice, *etc.*

Therefore, the following examples are legal statements in our logic (we denote public keys as K_X. Public keys can make statements either by sending messages through an authenticated channel, or by digitally signing statements):

- Self : Bind(K_B, Self’s Bob)
  which means (intuitively) that we call the principal with the key K_B “Bob.”

- Self : Delegate(Self's Bob, Delegate(AnyPrin, Read))

  which means that we give Bob permission to delegate read permission to anyone he wishes.

- $\mathsf{K_B}$: Bind($\mathsf{K_A}$, $\mathsf{K_L}$'s Alice)

  which means that the principal with cryptographic key $\mathsf{K_B}$ sent us a certificate stating that he thinks that the key $\mathsf{K_A}$ belongs to the principal called "Alice" in the name space of the principal with key $\mathsf{K_L}$.

The last class of statements corresponds to the `implies()` method in the Java Permission class. For example,

- Read("`/foo/*`") $\Rightarrow$ Read("`/foo/bar`")

  means that the right to read all files in the directory `/foo` implies the right to read the file `/foo/bar`. Statements of this kind are not inferred from signed certificates or authenticated channels. Rather, they reflect the fact that some permission objects in the Java security framework *imply* other permission objects.

2.3.2 Inference Rules

Our logic has a small set of inference rules, which allow us to infer new statements from a set of statements already held to be true. Below, X, Y and Z are principals, P is a permission, and B is a string. We will give an intuitive justification for each rule.

Delegation:

$$\frac{\begin{array}{rl} \text{Self} : & \text{Delegate}(X,P) \\ X : & P' \\ P \Rightarrow & P' \end{array}}{\begin{array}{rl} \text{Self} : & P' \end{array}} \qquad \text{(Del)}$$

This rule says that if I delegate a certain permission to some principal, and then that principal asserts its right to perform the action associated with that permission, I will have to believe that it is ok to go ahead and perform the action. We note, however, that the requester may request an action that is "weaker" than the permission granted. For example, if we delegate the right to read all files in directory `/foo`, and the requester requests access to file `/foo/bar`, we will believe that it is ok to go ahead and read file `/foo/bar`.

Transitivity:

$$\frac{\text{Self} : \text{Bind}(X,Y) \qquad \text{Self} : \text{Bind}(Y,Z)}{\text{Self} : \text{Bind}(X,Z)} \qquad \text{(Trans)}$$

This rule makes sure that the binding-permission relation is transitive, *i.e.* if I believe for example that Bob is a department head, and that department heads are managers, then I have to believe that Bob is a manager.

Containment:

$$\frac{\text{Self} : \text{Bind}(X,Y) \qquad \text{Self} : \text{Delegate}(Y,P)}{\text{Self} : \text{Delegate}(X,P)} \qquad \text{(Cont)}$$

This rule gives our delegation permission a subset-like (rather than a speaks-for-like) flavor [28, 1]. If I think that Bob is a manager, and I have given managers certain privileges, I have to believe that Bob has these privileges.

AnyPrincipal:

$$\frac{}{\text{Self} : \text{Bind}(X, \text{AnyPrin})} \quad \text{(AnyPrin)}$$

This rule simply states that any principal can take the place of the AnyPrin principal. Combined with rule (Cont), we can see that if we delegate a right to the AnyPrin principal, we have delegated it to any principal, which is exactly what the AnyPrin principal is supposed to mean.

Monotonicity:

$$X \text{ is not AnyPrin}, Y \text{ is not AnyPrin} \quad \frac{\text{Self} : \text{Bind}(X, Y)}{\text{Self} : \text{Bind}(X\text{'s } Z, Y\text{'s } Z)} \quad \text{(Mon)}$$

This rule states that the binding-permission relation is monotonic with respect to the "apostrophe" operation. For example, if I believe that Bob is a manager, then I have to believe that Bob's secretary is a manager's secretary.

Namespace Ownership:

$$\frac{\text{X} : \text{Bind}(Y, X\text{'s } B)}{\text{Self} : \text{Bind}(Y, X\text{'s } B)} \quad \text{(Own)}$$

This rule establishes that principals can speak about their own name space. If Bob tells me that the key $\mathsf{K_A}$ is his mother's, I will believe that $\mathsf{K_A}$ is Bob's mother's key. However, if Bob tries to tell us something about Alice's name space, this inference rule does not apply, and we do not automatically believe Bob's statements about Alice's name space.

Implication: There is a whole class of axioms that state implications of permissions. Every instance of the Java class Permission induces an implication axiom for every instance that it implies. For example, we would have that

$$\frac{}{\text{Read}(\text{“}\texttt{/foo/}*\text{”}) \Rightarrow \text{Read}(\text{“}\texttt{/foo/bar}\text{”})}$$

In particular, we have

$$\frac{}{P \Rightarrow P}$$

One instance of the implication rule is especially important. It reflects how delegation permissions imply each other:

$$\frac{\begin{array}{l}\text{Self} : \text{Bind}(Y, X) \\ P \Rightarrow Q\end{array}}{\text{Delegate}(X, P) \Rightarrow \text{Delegate}(Y, Q)} \qquad \text{(Impl)}$$

2.3.3 Access Control as Proof Finding

As we have seen in Figure 2.2, a request to perform a certain action is presented to a kernel together with a list of principals. The kernel has authenticated the first principal itself, but the other principals are not authenticated, they are merely impersonated. The immediate client to our kernel may or may not faithfully relay who *it* authenticated before it turned to us.

For every principal in this list, our Placeless kernel creates a statement describing the requested action. For example, if the immediate client was authenticated as having key K, and the requested operation is to read a document, then the statement describing the requested action is

$$\mathsf{K} : \text{Read}$$

This statement is added to the list of statements describing the local policy (see Section 2.4.4), and to the statements derived from certificates also presented by the client. As described in Section 2.4.5, certificates are simply statements of the logic that the client can send to the kernel. The certificates are digitally signed to ensure that the client in question indeed supports them.

Then, by applying our inference rules, an inference engine tries to deduce the statement

$$\text{Self} : \text{Read}.$$

If that succeeds for every principal in the list, access is allowed, otherwise it is denied.

2.3.4 An Example

In this example, we know the key of our boss, Alice, and have made an appropriate entry in our policy:

$$\text{Self} : \text{Bind}(\text{K}_\text{A}, \text{Self's}\ \mathsf{Alice}) \tag{2.1}$$

We have also given Alice read permission, and have given her permission to delegate that permission:

$$\text{Self} : \text{Delegate}(\text{Self's}\ \mathsf{Alice}, \text{Read}) \tag{2.2}$$

$$\text{Self} : \text{Delegate}(\text{Self's}\ \mathsf{Alice}, \text{Delegate}(\text{AnyPrin}, \text{Read})) \tag{2.3}$$

A client has connected to our kernel. During the SSL handshake, we have learned that the key of that client is K_B. The client is trying to read one of our documents. This can be expressed as

$$\text{K}_\text{B} : \text{Read}. \tag{2.4}$$

In addition, the client presents certificates. These are delegation or binding permissions signed by principals. We interpret them as statements made by the key that signed them. The first one says that the principal with key $\mathrm{K_A}$ (Alice) has bound the name "Lab" to the key $\mathrm{K_L}$:

$$\mathrm{K_A : Bind(K_L, K_A\text{'s Lab})} \tag{2.5}$$

Alice also tells us that she has bound the name "secretary" to the principal called "Bob" in the name space of the principal she calls "Lab"

$$\mathrm{K_A : Bind(K_A\text{'s Lab's Bob}, K_A\text{'s secretary})} \tag{2.6}$$

and that she has delegated her read permission to the principal she calls secretary:

$$\mathrm{K_A : Delegate(K_A\text{'s secretary}, Read)} \tag{2.7}$$

The last certificate our client presents to us is signed by the key $\mathrm{K_L}$, and states that in $\mathrm{K_L}$'s name space, the name "Bob" is bound to the key $\mathrm{K_B}$:

$$\mathrm{K_L : Bind(K_B, K_L\text{'s Bob})}. \tag{2.8}$$

From statements (2.1) through (2.8) we will have to prove that Self : Read holds.

Here is the proof. On the left, we show which statements and inference rules are needed to deduce the new statements, which are numbered on the right.

$(2.8) - (\mathit{Own})$	$\mathrm{Self} : \mathrm{Bind}(\mathrm{K_B}, \mathrm{K_L}\text{'s Bob})$	(2.9)
$(2.6) - (\mathit{Own})$	$\mathrm{Self} : \mathrm{Bind}(\mathrm{K_A}\text{'s Lab's Bob}, \mathrm{K_A}\text{'s secretary})$	(2.10)
$(2.5) - (\mathit{Own})$	$\mathrm{Self} : \mathrm{Bind}(\mathrm{K_L}, \mathrm{K_A}\text{'s Lab})$	(2.11)
$(2.1)(2.3) - (\mathit{Cont})$	$\mathrm{Self} : \mathrm{Delegate}(\mathrm{K_A}, \mathrm{Delegate}(\mathrm{Any}, \mathrm{Read}))$	(2.12)
$(\mathit{AnyPrin})$	$\mathrm{Self} : \mathrm{Bind}(\mathrm{K_A}\text{'s secretary}, \mathrm{AnyPrin})$	(2.13)
$(2.13) - (\mathit{Impl})$	$\mathrm{Delegate}(\mathrm{AnyPrin}, \mathrm{Read}) \Rightarrow \mathrm{Delegate}(\mathrm{K_A}\text{'s secretary}, \mathrm{Read})$	(2.14)
$(2.12)(2.7)(2.14) - (\mathit{Del})$	$\mathrm{Self} : \mathrm{Delegate}(\mathrm{K_A}\text{'s secretary}, \mathrm{Read})$	(2.15)
$(2.11) - (\mathit{Mon})$	$\mathrm{Self} : \mathrm{Bind}(\mathrm{K_L}\text{'s Bob}, \mathrm{K_A}\text{'s Lab's Bob})$	(2.16)
$(2.10)(2.16) - (\mathit{Trans})$	$\mathrm{Self} : \mathrm{Bind}(\mathrm{K_L}\text{'s Bob}, \mathrm{K_A}\text{'s secretary})$	(2.17)
$(2.9)(2.17) - (\mathit{Trans})$	$\mathrm{Self} : \mathrm{Bind}(\mathrm{K_B}, \mathrm{K_A}\text{'s secretary})$	(2.18)
$(2.15)(2.18) - (\mathit{Cont})$	$\mathrm{Self} : \mathrm{Delegate}(\mathrm{K_B}, \mathrm{Read})$	(2.19)
$(2.4)(2.19) - (\mathit{Del})$	$\mathrm{Self} : \mathrm{Read}$	

Since there is a proof for Self : Read, we know that it is safe to allow the requested read operation.

2.3.5 Semantics

We give a semantics for our logic by expressing the various components (permissions, principals, the ":" operator, *etc.*) in Church's high-order logic (which is sometimes called *type theory*) [3, 13]. For the purpose of this exercise we consider Church's high-order logic to be a logic that consists of formulas (which can be true or false), (typed) functions, strings (constants), and quantifiers over formulas or functions.

Let's first define some types. Let $\mathfrak{F}$ be the type of formulas. Then we define the type Perm of permissions to be the same as $\mathfrak{F}$:

$$\mathsf{Perm} :=_{def} \mathfrak{F}$$

A principal is a function mapping permissions to formulas, *i.e.* the type Prin is defined as:

$$\mathsf{Prin} \quad :=_{def} \quad \mathsf{Perm} \rightarrow \mathfrak{F}$$

From now on, we will denote the type of variables by subscript at the time they are introduced. For example, $X_{\mathsf{Prin}\rightarrow\mathsf{Perm}}$ is a function that maps from principals to permissions.

The ":" operator is defined as follows:

$$\forall A_{\mathsf{Prin}}, \forall F_{\mathsf{Perm}} \qquad A : F \ :=_{def} \ \exists G_{\mathsf{Perm}}.\ A(G) \wedge (G \supset F) \tag{2.20}$$

"$\supset$" means *implies* in Church's logic.

A binding permission is defined as follows:

$$\forall A_{\mathsf{Prin}}, B_{\mathsf{Prin}} \qquad \mathrm{Bind}(A,B) \ :=_{def} \ \forall P_{\mathsf{Perm}}.\ A : P \supset B : P \tag{2.21}$$

Likewise, we can define a delegation permission:

$$\forall A_{\mathsf{Prin}}, \forall P_{\mathsf{Perm}} \qquad \mathrm{Delegate}(A,P) \ :=_{def} \ \forall Q_{\mathsf{Perm}}.\ ((P \supset Q) \wedge A : Q) \supset Q \tag{2.22}$$

Implication is, in fact, the underlying implication of high-order logic:

$$\forall A_{\mathsf{Perm}}, B_{\mathsf{Perm}} \qquad A \Rightarrow B \;:=_{def}\; A \supset B \tag{2.23}$$

The AnyPrin principal is defined as

$$\text{AnyPrin} \;:=_{def}\; \lambda X.true \tag{2.24}$$

and the principal Self is defined as

$$\text{Self} \;:=_{def}\; \lambda X.X \tag{2.25}$$

The definition of the apostrophe (for things like A's B) is a little tricky. The observant reader will note, though, that the definition is designed so that inference rules (Mon) and (Own) can be proven as theorems in high-order logic. Let's assume that the apostrophe is defined as a function:

$$\forall A_{\mathsf{Prin}}, \forall B_{\mathsf{String}}. \qquad A\text{'s } B \;=_{def}\; \text{apos}(A,B)$$

The type of apos is $\mathsf{Prin} \to \mathsf{String} \to \mathsf{Prin}$. First, we define a function called "apos_like," which takes a function of type $\mathsf{Prin} \to \mathsf{String} \to \mathsf{Prin}$ as its argument and returns a formula:

$$\forall A_{\mathsf{Prin}\to\mathsf{String}\to\mathsf{Prin}}. \qquad \text{apos\_like}_{\mathsf{Prin}\to\mathsf{String}\to\mathsf{Prin}\to\mathfrak{F}}(A) \;:=_{def}$$
$$\text{sat\_mono}(A) \wedge \text{sat\_owner}(A) \tag{2.26}$$

Here, sat_mono and sat_owner are defined as follows. sat_mono defines the class of functions that satisfy the monotonicity rule, *i.e.* sat_mono(A) holds if and only if A satisfies the monotonicity rule (Mon).

$$\forall A_{\mathsf{Prin}\to\mathsf{String}\to\mathsf{Prin}}. \quad \text{sat\_mono}_{\mathsf{Prin}\to\mathsf{String}\to\mathsf{Prin}\to\mathfrak{F}}(A) :=_{def}$$
$$\forall P_{\mathsf{Prin}}, Q_{\mathsf{Prin}}, \forall S_{\mathsf{String}}.(\text{Bind}(P,Q) \supset \text{Bind}(A(P,S), A(Q,S))) \quad (2.27)$$

In the same fashion, sat_owner defines the class of functions that satisfy the namespace ownership rule:

$$\forall A_{\mathsf{Prin}\to\mathsf{String}\to\mathsf{Prin}}. \quad \text{sat\_owner}_{\mathsf{Prin}\to\mathsf{String}\to\mathsf{Prin}\to\mathfrak{F}}(A) :=_{def}$$
$$\forall P_{\mathsf{Prin}}, Q_{\mathsf{Prin}}, \forall S_{\mathsf{String}}.(P : \text{Bind}(Q, A(P,S)) \supset \text{Bind}(Q, A(P,S))) \quad (2.28)$$

Now, we can define the function apos as the intersection of all apos_like functions, *i.e.*, we say that apos$(P,S)(F)$ is true if and only if all apos_like functions A would have the principal $A(P,S)$ say F.

$$\forall P_{\mathsf{Prin}}, \forall S_{\mathsf{String}} \quad \text{apos}_{\mathsf{Prin}\to\mathsf{String}\to\mathsf{Prin}}(P,S) :=_{def}$$
$$\lambda F.\forall A_{\mathsf{Prin}\to\mathsf{String}\to\mathsf{Prin}}(\text{apos\_like}(A) \supset (A(P,S) : F)) \quad (2.29)$$

2.3.6 Soundness and Consistency

Given this embedding of our logic into Church's high-order logic, we can prove all our inference rules as theorems in that logic. This means that every model for Church's high-order logic is automatically a model for our logic. From this observation it follows that our logic is sound and consistent (see theorems 5402 and 5403 in Andrews' text book [3]). To be more precise, we know that

- if M is a model for Church's high-order logic, and there exists a proof for a certain statement S in our logic, that statement S is valid in M (soundness), and

- we cannot derive *false* in our logic (consistency).

See Appendix A for proofs of our inference rules as theorems in Church's high-order logic, which proves that our logic is sound and consistent.

2.3.7 Decidability

It is important to make sure that our logic is *decidable*, because we hope to implement a proof finder as the access control mechanism inside Placeless kernels.

ABLP logic [2] is a classic access control logic that inspired this and other work [4, 29]. However, ABLP logic is not decidable and therefore not useful in a scenario where the server performs access checks based on proof finding. Therefore, the authors of ABLP logic propose in their paper to use a decidable subset of their logic for access control decisions. They decide to keep most of their very expressive and powerful algebra of principals, and to give up a lot of the expressiveness of the logic itself in order to gain decidability. For example, security policies for the decidable subset can essentially contain only statements about group memberships and access-control-list entries.

Fortunately, the logic presented in this dissertation is decidable, and there is no need to "restrict" it for an actual implementation. See Appendix B for a proof of this statement. Interestingly, this is true even though our logic has restricted delegation (a principal can delegate only part of its privileges), which ABLP does not have. The main difference between ABLP and our logic (and a major reason why ABLP is not decidable) is our much more restricted notion of principals. There are no roles, quoting (which is handled outside the logic, see the Section 2.3.3), or principals acting on behalf of other principals. We believe, however, that we have constructed a practical and useful access control logic.

2.4 The Implementation

After presenting the logic itself, we now proceed to explain the implementation of a system that uses that logic.[5]

We have implemented our access control system for Placeless using the Java 2 Platform JDK. We use the IAIK [30] cryptographic provider and SSL implementation. We provide Java classes for the various types of principals and permissions, and we implemented an inference engine for our access control logic. We also implemented tools that hide most of the underlying mechanisms and should make it easy for Placeless users to manage their day-to-day security settings.

We start by explaining how the various entities of the logic (permissions, principals, *etc.*) are implemented, before we highlight some of the more interesting problems that we found when we implemented this real-world security infrastructure.

2.4.1 Statements

A statement in our implementation is merely a pair that consists of the principal making the statement, and the permission that the statement is about. A (local) policy is a list of statements all made by the SelfPrincipal (see Section 2.4.4).

2.4.2 Principals

There are five different kinds of principals in our system. They are all represented as subclasses of the Java class `Principal`, and explained in detail below:

[5]The existing implementation of our system is based on a previous version of the logic, as described in an earlier paper [5]. A new implementation would differ from the existing implementation only in a new inference engine, which is described in Appendix B.

LocalName

A LocalName is a string referring to a principal in someone's name space. For example, "Bob" is a LocalName. Which principal this LocalName refers to depends on the name space in which it is evaluated. Alice might call a different principal "Bob" than Charlie does. We call a LocalName *relative* because its interpretation is relative to some name space. There is no corresponding concept in our logic. In fact, although LocalName is a subclass of Principal in our system, a LocalName would not be considered a principal in our logic. Every LocalName has to be turned into an absolute ReferencePrincipal (see below) before it can be used by the system. For example, if Bob (whose key is K_B) issues a certificate mentioning the LocalName Alice, this will be turned into the ReferencePrincipal K_B's $Alice$. So, all that the class LocalName provides to users of the system is a shorthand to talk about names in *their* name space.

GlobalName

A GlobalName is a public key. The reason we call public keys "global names" is that it is like a string that everybody agrees on how to interpret. While we might both have different ideas about who "Bob" is, we do not have different ideas about who the principal with the public key "0x43453456. . . " is. We call a GlobalName *absolute* since it does not depend on a name space.

ReferencePrincipal

A ReferencePrincipal is a series of LocalNames that may or may not start with a GlobalName. If it starts with a GlobalName, we call the ReferencePrincipal *absolute*, otherwise we call it *relative*. Relative ReferencePrincipals are relative to a principal's name space. Absolute ReferencePrincipals are global in the sense that every principal will agree on the identity of an absolute ReferencePrincipal. The ReferencePrincipal

$$ReferencePrincipal(LocalName(Bob), LocalName(Alice))$$

is the principal called “Alice” in the name space of the principal we call “Bob.” From now on, we will simply write $Bob's\,Alice$ instead. Notice that our logic knows only absolute ReferencePrincipals. Just like LocalNames, relative ReferencePrincipals are a shorthand available to the users of the system. The underlying system will convert a relative ReferencePrincipal into an absolute ReferencePrincipal (which is a “proper” principal as far as our logic is concerned) by anchoring it in the name space of the principal using it.

SelfPrincipal

The SelfPrincipal is the principal representing the local system. The SelfPrincipal has a private key to sign certificates, authenticate itself over an SSL connection, and so forth. The identity of the SelfPrincipal is established when a Placeless client or kernel is started. The user has to supply a private key to the Placeless client or kernel to use for its SelfPrincipal. In our logic, we call the SelfPrincipal simply $Self$.

AnyPrincipal

The AnyPrincipal represents the all-embracing group that everybody is a member of.

2.4.3 Permissions

In our system, we use Java Permission classes. While we do not use any of the pre-defined permission classes to describe possible actions in Placeless, we do use the notion that permissions can *imply* each other [26]. Permission A implies permission B if the set of possible actions that permission A describes is a superset of the set of possible actions that permission B describes. For example, the permission `ReadPermission(/usr/local/*)` implies the permission `ReadPermission(/usr/local/foo)`.

In Placeless, we have permission classes denoting the following actions:

- Reading from documents. This includes reading document contents as well as reading the values of document properties.
- Writing to documents and deletion of documents. This includes the modification, deletion, and creation of properties.
- Creation of documents.
- Notification about events happening in kernels.

These are called primitive permissions. We also have special permissions denoting name bindings of principals and permission delegation, which are discussed below.

DelegationPermission

A DelegationPermission is a subclass of `java.security.Permission`. The constructor of a DelegationPermission takes two arguments: A principal, and a permission. Informally,

$$\text{Self} : \text{DelegationPermission}(\mathrm{X}, \mathrm{P})$$

means that we have given permission P to principal X. The statement

$$\mathsf{SomeKey} : \text{DelegationPermission}(\mathrm{X}, \mathrm{P})$$

means that the principal with the key SomeKey has delegated permission P to principal X.[6]

For every new permission class we introduce, we need to define when an instance of this class implies another permission. A DelegationPermission can imply only other

[6]If X is a LocalName, it is evaluated in the name space of the principal making the statement.

DelegationPermissions. $\mathrm{DelegationPermission}(\mathrm{X},\mathrm{P})$ implies $\mathrm{DelegationPermission}(\mathrm{Y},\mathrm{Q})$ if

- P implies Q and
  1. principal Y is bound to principal X, or
  2. principal X is the AnyPrin principal, or
  3. principals X and Y are ReferencePrincipals that share a common "tail" (*i.e.*, they are of the form X'sA_1's $\ldots A_i$'sC_1's $\ldots C_n$ and Y'sB_1's $\ldots B_j$'sC_1's $\ldots C_n$) and principal Y's "head" is bound to principal X's "head" (*i.e.*, Y'sB_1's $\ldots B_j$ is bound to X'sA_1's $\ldots A_i$).

In Appendix B we give a rationale for this implementation. As we show there, the above implementation is equivalent to rule (Impl) in a logic that is missing rules (Mon) and (Own).

BindingPermission

A BindingPermission is another subclass of `java.security.Permission`. Its constructor takes two principals, and *binds* one to the other. But note that this binding is not symmetric — a $\mathrm{BindingPermission}(\mathrm{X},\mathrm{Y})$ does not imply a $\mathrm{BindingPermission}(\mathrm{Y},\mathrm{X})$. Rather, a BindingPermission establishes a kind of subset-superset relationship between the two principals. For example,

$$\mathrm{Self} : \mathrm{BindingPermission}(\mathrm{Bob},\mathrm{Managers})$$

can be interpreted as saying that Bob is a member of the group Managers. Sometimes, it is easier to think of BindingPermissions as establishing a "speaks-for" relationship. For example,

$$\text{Self} : \text{BindingPermission}(\mathsf{BobsKey}, \text{LocalName}(\text{Bob}))$$

says that the key BobsKey speaks for the principal I call Bob.

A BindingPermission can imply only itself, *i.e.* BindingPermission(X,Y) implies BindingPermission(X',Y') iff $X' = X$ and $Y' = Y$.

Why Separate Binding and Delegation Permissions

Both BindingPermission(..., ...) and DelegationPermission(..., ...) are instances of a more general "speaks for" relation. Why do we separate them? We keep them separate to model the separation found in a corporate enterprise: a human resources or corporate security office is in charge of issuing a credential saying that John Doe is an employee of Company X, while Doe's management chain (or other colleagues) are responsible for setting access controls on their documents. Thus, while we support attribute certificates, we also find names to be useful for access control: people are used to thinking in terms of names, and names provide a useful level of indirection to keys. We write policies in terms of names, not keys, so that a user's key can change over time without requiring changes to the security policy.

2.4.4 Policies

Calls to a remote Placeless kernel are handled by Java RMI server objects. They pass the identity of the caller, together with certificates the caller provides[7] and a statement describing the intended action, to a reference monitor. That reference monitor, just like the Java `SecurityManager`, then makes a decision whether to proceed with the call or not.

[7]These are signed statements of our our access control logic. A client might, for example, provide certificates to prove that someone delegated privileges to him.

The decision will be based on the *policy* associated with the object on which the proposed action is to be performed.

As noted earlier, a policy is a set of statements in our access control logic. All statements in a policy are of the form

$$\begin{array}{rcl} \text{Self} & : & \text{BindingPermission}(\ldots,\ldots) \\ \text{or} & & \\ \text{Self} & : & \text{DelegationPermission}(\ldots,\ldots) \end{array}$$

We store the BindingPermission statements in the user's *name space*. The DelegationPermission statements (also sometimes referred to by themselves as the "policy" as opposed to the name space) are stored with the documents. This is similar to simple access control lists, which are also stored with the document they are protecting. There is a possible level of indirection for Placeless policies, though: A document may, instead of specifying its own policy, specify the name of a policy. That policy is then looked up in the user's *policy document* (a collection of policies). If a document does not specify a policy for its access control, the policy called "default" from the user's policy document is used. Note that these named policies merely provide a convenient way to name a set of statements, they do not add any functionality to our system beyond the convenience they provide.

Sometimes, certain actions do not pertain to documents. For example, which policy should we consult when we are about to create a new document? The document does not exist yet, so it cannot specify a policy governing its creation. For these types of actions we consult the policy called "space" from the user's policy document. It is up to the reference monitor to decide which policy to apply. In our implementation, the reference monitor will first decide whether the proposed action calls for the space policy or for a document-specific policy. If the action needs a document-specific policy, it tries to fetch the policy from the document. This can either be an actual policy (a set of statements), or a name. In the latter case, the reference monitor resolves the name against the list of policies stored in the user's policy document. If there is no policy with the given name, the "default" policy is used.

The reference monitor will then add the *BindingPermission* statements from the user's name space, together with the statements extracted from the caller's certificates and the statement describing the intended action, to the policy and engage the inference engine to try to prove that access should be granted (see Section 2.3.3).

2.4.5 Certificates

Certificates are signed BindingPermissions or DelegationPermissions. A client presents certificates with his request. Certificates are interpreted as statements made by the key that signed them. In our implementation, we cache certificates on the kernel side so that for a given SSL connection, a client has to present its certificates only once. Our implementation does not support certificate revocation lists. Instead, our certificates expire. The inference engine will not take into account statements gained from expired certificates.

2.4.6 Inference Engine

As explained in Section 2.3.3, the inference engine tries to find a derivation of a statement of the form

$$\mathit{Self} : \mathit{Permission}$$

from the set of statements gained from the policy, name space, certificates, and the requested action. We implemented a simple forward-chaining inference engine for the version of our logic that is described in an earlier paper [5]. We keep generating new statements from already existing ones.

For the version of the logic presented in this dissertation, the inference engine has to be changed slightly, since inference rules (AnyPrin) and (Mon) introduce new principals in their conclusions. A possible implementation of the inference engine is presented in Appendix B, as part of a proof that the logic presented here is decidable.

2.4.7 Running RMI over SSL

Since the release of the Java 2 Platform, it has been relatively easy to change the transport mechanism underlying RMI. We chose to use SSL to gain both privacy and authenticated communication. In Placeless, the SSL layer always requires client authentication and uses a specific cipher suite[8] since there is no need for cipher suite negotiation. In SSL, the handshake between client and server can fail if either client or server do not authorize the other party to connect. In Placeless, we always allow two parties to establish an SSL connection with each other. We note the identity of the other side of the communication link and defer access control decisions to higher layers in the system.

When an RMI client wants to talk to an RMI server, it uses *RMI stubs,* which are objects that impersonate remote (server) Java objects in the local Java Virtual Machine (VM) and forward calls to the (remote) VM. If the client does not have the stub code available locally, it can download it from the server. In the case of a nonstandard transport mechanism (such as SSL), the client might also have to download code that implements the non-standard transport mechanism. Note that in the case of SSL this presents a problem. The client needs a trustworthy SSL implementation so that (a) it can trust the identity of the other end of the connection, and (b) the client believes that nonces and session keys are properly generated. However, in certain situations it might acquire the code for that SSL implementation from the very party it is trying to authenticate. This is clearly a security hole. In our current implementation, we just disallow dynamic downloading of stub code.

[8] 1024-bit RSA for key exchange, 128-bit RC4 for encryption, and SHA for MACs.

This, however, disables a whole set of features for distributed Java programs, so that a more satisfying solution is yet to be found. Unfortunately, the Java 2 RMI implementation does not provide hooks to control the downloading of stub classes via RMI.

The SSL implementation we used provides new `Socket` classes, which can be used just like normal TCP sockets, but implement SSL functionality. An `SSLSocket` provides a method to query the identity (*i.e.* the X.509 certificate) of the other party of an SSL connection. This feature is lost in the "higher" RMI layer – an RMI server object has no way of knowing through which specific SSL connection a call to one of its methods has been invoked. This, however, is necessary to establish the identity of the party originating the call and to perform access control decisions. There is no easy way to add this functionality. After all, RMI is designed to *hide* the transport layer from upper layers, since it is supposed to be transport layer independent. It turns out, though, that in the current implementation of RMI the thread in which a communications socket is created for an RMI connection is the same thread in which calls to the server object associated with this connections are executed. There is a new thread for every RMI connection. While this may be questionable in terms of performance, it gave us a handle to track identities of SSL peers into the RMI layer: When an SSL socket is created, it notes which thread created it. Later, when an RMI server object executes a call that was initiated by a remote party, it can again check which thread is currently executing, and lookup which `SSLSocket` is associated with that thread. This way, RMI server objects can learn the identity (an X.509 certificate) of the parties invoking calls on them.

2.4.8 Tools

We implemented several tools for users to handle our access control system.

Key creation: We use the Java keytool to create 1024-bit RSA keys and self-signed X.509 certificates. The X.509 certificates are not used for any purpose except to exchange public keys in the SSL handshake.

Key export and import: There is a tool to export the user's public key to a file. Other users can then import that public key and map it to local names in their name spaces.

Name space manager: The name space manager is a tool that allows users to create statements of the kind

$$\mathit{Self} \quad : \quad \mathit{Bind}(\ldots,\ldots).$$

for their own local name space. They can bind local names to keys, ReferencePrincipals, and other local names.

Policy tool: The policy tool allows users to make statements of the kind

$$\mathit{Self} \quad : \quad \mathit{Delegate}(\ldots,\ldots).$$

and store them in named policies. In the current implementation, the kinds of supported statements are restricted. Permissions can only be granted to LocalNames, and we do not support the granting of DelegationPermissions or BindingPermissions.

The policy tool can also be used to create new policies, delete policies, and remove statements from a policy. There is also a tool that populates the "default" and "space" policy with reasonable default values.

Certificate tool: The certificate tool allows users to import certificates, *i.e.* signed statements of the form

$$\begin{array}{rcl}\mathsf{SomeKey} & : & \mathit{Bind}(\ldots,\ldots)\\ & \text{or} & \\ \mathsf{SomeKey} & : & \mathit{Delegate}(\ldots,\ldots)\end{array}$$

into their certificate collection. The certificate tool can also be used to export certificates of the form

$$\mathsf{UserKey} \quad : \quad \mathit{Delegate}(\ldots,\ldots).$$

This is equivalent to delegating permissions to third parties.

The above-mentioned name space manager can be used to export certificates of the form

$$\mathsf{UserKey} \quad : \quad \mathit{Bind}(\ldots,\ldots)$$

This will allow third parties to make statements about principals known in our name space.

2.5 Experience with Access Control Logics

We wanted to base our system on an access control logic because we wanted to be able to develop a formal model of the complex relationships between privilege delegation and linked local name spaces. This way, it is easier to reason about certain properties of the system.

Abadi, Halpern and van der Meyden [1, 28] have tried to formalize SDSI-style linked local name spaces and point out that there is no obviously right way to do so. While they can prove that their logics are sound with respect to a certain semantics, the choice of semantics is arbitrary.[9] One cannot prove that a certain logic will not be surprising, *i.e.* allow intuitively undesirable conclusions. There has not been a published attempt to formalize the whole SPKI framework, which would combine the linked local name spaces with an access control framework. For our work, we attempted to do that, although on a smaller scale (*e.g.* we do not have the notion of a threshold principal).

As mentioned in an earlier paper [5], we have encountered similar surprises in previous versions of our logic. For example, the following scenario was possible in the version of our logic presented in that paper. Let us assume that we give all managers the right to delegate read permissions:

$$Self : Delegate(Managers, Delegate(AnyPrin, Read)) \tag{2.30}$$

and we also know that Bob is a manager:

$$Self : Bind(Bob, Managers) \tag{2.31}$$

Let us also assume that Bob tells us he has delegated his right to read a document to his secretary:

$$Bob : Delegate(Secretary, Read) \tag{2.32}$$

[9]So, by the way, is ours. We just needed *some* semantics in order to prove consistency of our inference rules.

The containment rule in the previous version of the logic allows us to infer

$$Managers : Delegate(Secretary, Read) \tag{2.33}$$

from (2.31) and (2.32), and the delegation rule in that version allows us to infer

$$Self : Delegate(Managers's\ Secretary, Read)$$

from (2.30) and (2.33). This means that every manager’s secretary has read permission, which is clearly not what we would have wanted to conclude, given the assumptions above.

The logic as presented here does not suffer from this problem, but (as explained in Section 2.4.6) has a rather inefficient inference engine. To find a logic that meets intuition about what is secure and what is not, and at the same time is efficiently implementable, remains an open problem.

2.6 Future Work

As we have just seen, this work has led to some interesting ongoing research. The most pressing and interesting question is whether our access control logic is “good.” While it is relatively easy to prove that our logic is sound (see Appendix A), it is harder to see whether it will “surprise,” *i.e.*, allow conclusions that we intuitively would consider insecure (see Abadi’s and Halpern’s and van der Meyden’s papers [1, 28] for some examples on “surprising” conclusions found in earlier attempts to formalize SDSI).

Our current implementation of the inference engine can certainly be improved upon. For example, we need to implement a caching mechanism that allows quick lookups of frequently asked access control queries. Also, a backward-chaining or a combination of backward- and forward-chaining might exhibit better performance than our current forward-chaining implementation.

Other topics that are not yet resolved in the current implementations include:

Certificate dissemination Clients have to present certificates to the kernels to support their requests. How do clients find out which certificates are relevant for the requested action? Currently, clients supply all certificates they collected to kernels. Even though they are cached at the kernel side, this still is a performance problem. It also presents a privacy problem. If the client sends more certificates than needed for a specific type of access to a server, it tells the server more about itself than is necessary to gain access. In some situation, it may be undesirable to reveal more about oneself than is necessary. Some work has already been done in this area [19].

Securing the name sever Placeless uses a name server similar to the RMI registry to locate kernels. Currently, the name server does not follow a security policy when registering kernels under their respective names. It could therefore be possible for a rogue kernel to impersonate another kernel to a client. Clients could in theory test the identity of the kernel they are connected to (after all, they have authenticated it over an SSL connection), but in practice hardly ever do so. We need to secure the Placeless name server.

Legacy authentication Recall the WWW servers in Figure 2.1. When they authenticate a browser, they do not necessarily use strong SSL authentication, which yields the public key of the browser. They might use a different kind of authentication that just yields a user name (for example, if they use password-based authentication). How do we translate that user name into a Placeless principal? In our current implementation, we assume that the WWW server knows (say, by convention) the local name under which the authenticated user is known to the kernel to which the server is connected. The server will therefore impersonate a principal local to the kernel when it connects.

This works only if the Web server and the kernel(s) it talks to agree on the names of principals. We are currently looking for a better solution for this problem of legacy authentication.

Parameterized permissions In the current system, every Placeless document (conceptually) has its own policy. Therefore, most of our permission classes (*e.g.* ReadPermission, WritePermission) do not have any parameters. For example, if we add this statement to the policy of document Foo

$$Self : DelegationPermission(Bob, ReadPermission)$$

then it is implicitly clear that the ReadPermission pertains to the document Foo. However, if Bob now wants to delegate his ReadPermission for Foo, he has no way of expressing this. He can delegate only his parameterless ReadPermission, effectively delegating his privilege to read any document he is allowed to. If our permissions took parameters (like the standard Java permission classes do), Bob could delegate his permission to read specific documents to third parties.

2.7 Conclusion

We have designed and implemented a distributed access control system for the Placeless Documents System. In addition to strong authentication, we also provide communication confidentiality and integrity.

Borrowing ideas from SDSI/SPKI, our system can be used across administrative domains, and supports attribute certificates in addition to identity certificates.

Policies are expressed as a set of statements of the kind

$$Self : DelegationPermission(Principal, Permission)$$

and

$$\mathit{Self} : \mathit{BindingPermission}(\mathit{Principal}, \mathit{Permission}).$$

The permissions can take arbitrary arguments, and can therefore be used to express a wide range of security policies. For example, we could have an `IntervalPermission` class that would take three arguments: A not-before-date A, a not-after-date B, and a permission P. Then, we could define that

$$\mathit{IntervalPermission}(A, B, P)$$

implies permission Q if and only if P implies Q and the current time is between A and B. This way, we can limit arbitrary privileges to certain times. Other arrangements, and permission classes implementing them, are possible.

Central to our system is a simple access control logic that tries to capture the spirit of SDSI, SPKI and ABLP-style systems while naturally fitting within the Java framework. There are several tradeoffs one has to make when designing such a system. One is the tradeoff between overly inflexible but simple systems like access control lists (which do not allow for delegation, for example, and usually cannot span administrative domains) on one side, and very general, but more difficult-to-use systems like proof carrying authentication [4] (which usually put the burden of rather complex proofs on the requester of an operation) on the other side. We were trying to strike a balance by designing a system in which it is comparatively easy to find a proof, but that is more expressive than simple access control lists or capability-based systems.

Another tradeoff is that between privacy of the server and privacy of the client. One could imagine a system in which servers publish their security policies (giving up privacy), and clients create proofs based on those policies. Our system is at the other end of the spectrum: The requester has to supply a sufficient set of certificates (possibly giving up privacy), while the inference engine on the server side constructs the proof on its own. The reason we took this approach is twofold. One, our philosophy is server-centric: we were building a system to protect resources on the server from unauthorized access, not to protect the privacy of the clients. The second reason is of purely practical nature - in a first-shot implementation it proved more convenient to have the clients provide all certificates they have, rather than implementing a protocol in which server and client try to negotiate how much to reveal about their respective policies/credentials.

We find the SDSI approach of linked local name spaces very appealing, but encountered known subtleties in its formalization, especially when we tried to combine this with SPKI-like delegation statements (see Abadi's and Halpern's and van der Meyden's discussions of the formalization of SDSI [28, 1]).

All in all, we have created a cryptographic security infrastructure based on an access control logic that allows servers to reason about access control privileges of unknown clients. We demonstrated its practicality by implementing it for the Placeless Documents System.

Chapter 3

WindowBox: A Simple Security Model for the Connected Desktop[1]

3.1 Introduction

In the previous chapter, we looked at mechanisms servers could use to specify access control policies for ad-hoc collaboration. In this chapter, we are going to shift the focus of our investigation to the client side of things: we are investigating what kind of mechanisms users of personal computers could apply to protect themselves from possibly dangerous results of ad-hoc collaborations, in which they engage with other computers on an (open) network.

The fundamental assumption for this chapter is that a lot of security breaches do not stem from implementation bugs in applications or operating systems, but rather from mismanaged protection mechanisms. We do not claim that implementation bugs play no role in security breaches, but the most notorious breaches seem to be connected with "human error." For example, the 1988 Internet Worm exploited (among other things) the fact that most administrators had configured the sendmail daemon to run in "debug" mode,

[1]A version of this chapter was published in an earlier paper [7].

which made it very easy to launch a network attack. The vast majority of traditional virus infections can be contributed to careless behavior of people, and not to “security holes” in the applications or operating systems used. More recently, e-mail viruses like the “Melissa” or “ILOVEYOU” virus have been spread by millions of users who answered “yes” when asked by the system whether they wanted to open a particular, *possibly virus-infected*, e-mail attachment.[2]

It is not our intention to blame the users for “human errors” they committed. After all, computers should adapt to humans, and not the other way around. On the contrary, we believe that the tools available to protect sensitive resources and networks are tedious to use, non-intuitive, and often require expert knowledge. That can hardly be expected of an average PC user. As a result, many PC and workstation users end up administering their system security poorly and creating serious security vulnerabilities.

What we need, therefore, are tools that are easy to comprehend and to use. Moreover, if the underlying *security model* was more intuitive than setting up access control lists or protection domains, then maybe there is hope that we can devise tools that users would actually use to set up a secure environment on their personal computers.

For example, one idea users can understand is that of absolute physical separation. Given a set of unconnected machines, between which all information must be explicitly carried (say, on a floppy disk), a user can understand that sensitive information or privileges on one machine are safe from potential threats on other machines.

In this chapter we present WindowBox, a security model based on that observation. WindowBox presents the user with a model in which the workstation is divided into multiple desktops. Each desktop is sealed off from the others, giving users a means to confine the possibly dangerous results of their ad-hoc collaborations. If the party with which

[2]There is actually some controversy as to whether Microsoft (the manufacturer of the e-mail software that was vulnerable to the attack) had lacked prudence when designing the security features of their e-mail software. It remains a fact, though, that the software *technologically* was capable of preventing the spread of the virus.

they engaged in ad-hoc collaboration turns out to be malicious, that party can attack only the desktop from which the ad-hoc collaboration was initiated, leaving the other desktops protected.

We have implemented our security model on Windows 2000, leveraging the existing desktop metaphor, the ability to switch between multiple desktops, and specific kernel security mechanisms.

3.2 The Threat of Ad-hoc Collaboration

Today's typical computing environment no longer consists of a centrally managed mainframe accessed through terminals; more often it consists of networked, Internet-aware PCs and workstations. The security threat model has thus changed in several ways:

- Because individual users have greater control over the PCs or workstations they typically use, they also have greater responsibility for the administration of these machines and their security. Where knowledgeable administrators once made professional decisions about security risks, those decisions (such as permission and denial of access to system and data resources) are increasingly in the hands of relatively uninformed users.

- The more open, distributed architecture of PCs and workstations makes it possible for users to introduce new applications and even operating system modifications to their machines – opening up another avenue of attack, through viruses, Trojan horses and other malicious applications.

- Internet connectivity has greatly expanded the range of possible attackers with access to a user's machine.

These threats are closely related to the fact that ad-hoc collaborations happen more and more often. Viruses or Trojan horses usually come from unknown or (as is the case with the latest e-mail worms) friendly – but unsuspecting – sources, against which normal access control mechanisms are not applicable. The unprecedented connectivity of personal computers also increasingly exposes them to attacks by strangers, making ad-hoc collaboration an especially dangerous activity to engage in.

Various tools are already available to combat each of the above threats. For example, existing discretionary access control mechanisms are usually available to limit access by (potentially hostile) applications to files. Firewalls and proxies can thwart certain kinds of network attacks. Finally, sandboxing has become a popular method for restricting a process' privileges to a subset of its owner's privileges, usually in the case where that process is executing some untrusted code. However, in practice we often see these techniques fail, not necessarily because they were poorly implemented, but because they are being poorly applied. The reason for this is that the currently used techniques, taken on their own, tend to be too complex for ordinary users to administer effectively. Taken together, they can be completely overwhelming even to fairly experienced users. As a result, confused users, faced with the responsibility of using and managing these techniques, often make ill-informed decisions with misunderstood implications — including, perhaps, serious security vulnerabilities.

In the next section, we will analyze further existing security mechanisms and explain why they fail to deliver, especially in the hands of "average" users. In Section 3.4 we introduce the WindowBox model in more detail, before we explain our implementation on Windows 2000 in Section 3.5.

3.3 Existing Security Mechanisms

3.3.1 Access Control Lists

In the traditional, mainframe-based model, users wish to restrict access to their resources (which are almost always data files) to some limited set of users. Conceptually, an access control matrix [32] describes what each user can do to each file. We often find that in a given system, the concept of an access control matrix is not implemented as an actual two-dimensional table listing all possible actions in the system against all possible users. Instead, we find more efficient implementations, such as Access Control Lists (in Windows 2000 and other operating systems). Access Control Lists (ACLs) are used to restrict access of users to files. In practice, the vast majority of such restrictions limit access to a resource to a single owner (or not at all). In this manner files, for instance, can be fairly easily made "private" (accessible only to the file's owner) or "public" (accessible to all the system's users).

In the PC/workstation setting, however, a user may own many other objects – systems resources, communications resources, and so on. These are often more difficult to control by ACL, for various reasons: the object may perhaps be created and used by applications without the owner's knowledge in the first place; its ACL may not be accessible to the owner, for lack of an appropriate user interface; or it may be (as in the case of remotely accessible resources) difficult for the owner to tell who should or should not have access to the object – or even who is capable of attempting to access it. This last point is one of the reasons why capability-based systems [33] and trust-management systems [11, 21] were invented, and is especially pronounced in the presence of ad-hoc collaboration. We have devoted an entire chapter of this dissertation (Chapter 2) to explaining how access control lists can be replaced by a more powerful mechanism.

Moreover, deciding on the appropriate ACL (let alone the correct policy in terms of statements of an access control logic) for every individual object (even whether it is "public" or "private") is a complicated, tedious and distracting task. Hence systems typically apply defaults to construct an ACL without consulting the user (such as when a new file is created). Since these default ACLs are assigned in the absence of information about the context in which they are created, they often conflict with the user's intentions.

3.3.2 Application-Level Security

When the resource in question is an application, it is typically the application itself that is made responsible for its own security. The operating system provides applications with various security services (such as authentication for remote connections); using these services, an application can limit access to itself as specified by the user running it, thus (hopefully) preventing its own exploitation by an external attacker.

This application-oriented approach has two major disadvantages:

- The user making the access decisions must deal with a different configuration procedure for each application – hopefully involving the same system-provided infrastructure, but sometimes even depending on application-specific authentication and access control mechanisms.

- Each application's security is implemented separately – and therefore has its own independently buggy security. A user would need to know all the security holes in each application, and how to defend against attacks on each one, in order to secure his or her system against attack.

A preferable alternative would be for the user to be able to set access control rights for any application from "above," requiring any inter-process communication reaching the application to be authenticated as coming from a permitted user or user/application pair. A natural analogy is the file system: just as access (of any sort) to a file can be limited to a particular set of users, so too should access to an application. (Indeed, as the distinction between applications and data blurs, so does the distinction between the two types of access.)

Of course, sometimes the problem is not that of unauthorized access *to* applications, but of unauthorized access *by* applications (consider viruses or Trojan horses). We discuss these cases in Section 3.3.4.

3.3.3 Firewalls and Proxies

The problem of controlling access to an application becomes even more difficult when the application is intended to be "network-aware." A well-connected PC or workstation, for example, runs a number of applications of various types; these may include both traditionally server-based applications (HTTP servers, FTP servers and the like) and traditionally user-oriented (but increasingly network-aware) applications. All of these may at various times send data onto the network or monitor and read arriving network data, using network services provided by the operating system. We might assume that the careful user avoids installing deliberately malicious applications; however, even established commercial applications can have security holes which may be exploited by hackers sending them unexpected data.

To protect against the possible existence of such holes, many administrators of large networks of PCs/workstations install firewalls or proxies, which filter out network traffic that does not conform to the formats that the applications are expected to handle correctly. But applications may still have bugs which allow an attacker to subvert them by sending them data that is allowed through by the firewall or proxy, but has unanticipated effects due to bugs or design oversights in the application.

Therefore, the problem would be much more effectively addressed by enforcement of the following two restrictions:

- Limiting applications' access to the network to authenticated connections, with access control applied to these connections "from the outside," as discussed above.
- User-imposed blocking of applications involved in network communications (especially unauthenticated connections) from accessing sensitive resources on the same machine.

For example, a personal HTTP server might be allowed to accept unauthenticated connections with the outside world, but be forbidden to write to files or communicate with other applications beyond an "internal firewall." The other side of this firewall might contain "private" applications permitted to communicate with each other, but not with the HTTP server, and not with the network at all except over authenticated connections to members of a particular group of trusted users (say, an Intranet).

3.3.4 Sandboxing

Meanwhile, the recent proliferation of mobile code via the Internet has spurred demand for mechanisms by which such mobile applications can be "sandboxed" (granted only restricted access to system resources, possibly including files, communication channels, miscellaneous system services and other applications). Originally, mobile code consisted in practice only of small "applets" with very limited functionality (and thus easy to sandbox tightly). Increasingly, however, the crisp lines between simple, casually distributed applets and "store-bought" applications, and between entire applications and individual components, have begun to blur. Sophisticated software is now being distributed over the Internet in the form of applets or even entire applications, and components distributed in this fashion are being used together in complicated ways. Even operating system updates, for instance, are being distributed over the Internet. The distinction between “trusted” and “untrusted” software is thus increasingly a continuum, with no software enjoying the confidence once (possibly too naively) accorded store-bought, “shrink-wrapped” software. It follows that some degree of sandboxing will become desirable for a wide range of applications and components.

A prominent example of the necessity of ubiquitous sandboxing is the use of cryptographic keys for sensitive functions such as electronic commerce or Intranet authentication. The susceptibility of such keys to compromise by viruses or security holes, and the dire consequences of such compromises, make it imperative that keys be accessed only by applications that can be trusted to use them properly (requesting direct user approval, for instance). A broadly enforced sandboxing policy could make such protection possible.

The major existing sandboxing framework is the Java virtual machine (VM), which can be used to allow “applets” (typically embedded in Web pages) restricted access to resources. This VM is only available for applications distributed in the form of a particular Java byte code, and then translated (with some performance overhead) into native machine code. The Java security model allows applications to be accompanied by digitally signed requests for various system access privileges; the user or administrator can decide whether to grant applets their requested privileges based on the identities of the digital signers.

This, of course, means that traditional signature-based sandboxing cannot work for ad-hoc collaboration. In this case, we do not know in advance whose mobile code we would like to upload onto our computer, and can therefore not make appropriate policy decisions. While sandboxing is a good technique to restrict privileges of unknown code, basing those restrictions on the identity of code signers is of no use in the ad-hoc collaboration scenario.

3.4 A Model For Sandboxing-Based Security

The above issues suggest the following goals for a security model:

- It should unify the per-user access control functions of the ACL model with the security properties of firewall- and sandboxing-based models, so that users have to deal with a single model only;
- It should allow basic application-level security (restricting both access to the application and its range of permitted behavior) to be applied independent of the application;
- It cannot be based on policies that determine before-the-fact what the privileges of other parties are, if those parties are unknown to us before we collaborate with them.
- It should be simple enough to be managed via a natural, intuitive user interface.

The last goal is perhaps the most crucial, as well as the most difficult. Creating an understandable user interface for a security model is generally a daunting task, given the complexity and criticality of security administration. The only hope is to design a simple and natural model that users can grasp and manipulate intuitively, then present a UI which reflects this model. Otherwise, confused users will inevitably make dangerously mistaken security decisions – such as disabling all security features entirely, to avoid the inconvenience of dealing with them.

3.4.1 The Multi-Desktop Premise

We propose here a model based on an extremely simple idea: while users cannot really intuit the complex rules associated with zones or ACLs, one idea they can understand is that of absolute physical separation. Given a set of group or zone permission rules, a user will have a difficult time determining if it expresses his or her idea of security. But given a set of unconnected machines, between which all information must be explicitly carried (say, on a floppy disk), a user can understand that sensitive information or privileges on one machine are safe from potential threats on other machines. It may be that many users (particularly small businesses) are using multiple machines in this manner today to secure their sensitive data and applications from the Internet. And of course, large enterprises use proxies all the time to protect their internal machines from the world outside.

Moreover, users' business (both information and – to a lesser extent – applications) tends to divide up fairly cleanly into categories, such as "personal finance," "business/office/intranet," "Internet gaming," and so on. The amount of information flowing between these categories is typically relatively small, and therefore should be manageable through direct user intervention (carrying floppy disks between machines, or its drag-and-drop analog in the virtual context) without excessive strain on the user.

In particular, users can more or less confine the possibly dangerous results of ad-hoc collaboration to a single desktop. All they need to do is restrict their ad-ho collaborations to one of the desktops. If they then download a Trojan horse or catch a virus, that malicious software will not be able to access the other desktops, which presumably hold the sensitive, or private data to which users would like controlled access.

Sometimes, users might get tricked into moving malicious pieces of data into a sensitive desktop (consider for example a cleverly engineered Melissa-type virus that appears to have been sent by a colleague). The premise of this work is that this happens very rarely, as most people's work *naturally* divides up into different categories (*i.e.* there should rarely be a reason why a possibly infected Word document should be copied from the ad-hoc collaboration desktop). Hence, the multi-desktop system provides better security than a single-desktop one.

3.4.2 The WindowBox Security Model

In the WindowBox security model, the user can construct multiple desktops, which are kept completely isolated, except by explicit user action (such as a direct point-and-click command or response to a dialog or warning box). To a first approximation, the desktops start off completely identical, and are provided just so that the user can use different desktops for different tasks. As the user uses the different desktops, each one accumulates its own files, and applications in one desktop cannot access files in other desktops except by the aforementioned user action. To aid the user in consistently using the desktops for their specific purpose, each desktop has its own network access restrictions and code-verifying criteria (although the user would manipulate these only indirectly, by defining and configuring desktops). Many special-purpose applications would be confined to a single desktop; other, more general applications would be "installed" in multiple desktops, but would have different access rights, and possibly even different behaviors, in each desktop (for example, a word processor might have different defaults depending on whether it is being run in an enterprise/Intranet desktop or a personal one).

In some ways, the multiple desktops can be considered as representing different users logged on simultaneously. A key difference, however, is that simple user-mediated actions would always be able to transfer data between one desktop and another, and (if necessary) create new desktops or change the properties of existing ones. From the network's perspective, on the other hand, these desktops would have very different security properties. For example, access to the private key necessary for authenticating as the user in a secure connection to a particular server may be restricted to applications in a particular desktop. Hence a server that accepts only secure connections would implicitly require that the user access it from only that desktop.

3.4.3 Examples

Most of the work of defining and configuring desktops should be a matter of choosing among standard desktop types with preset, mildly customizable attributes. We suggest a few natural ones here.

The Personal Desktop

A simple example of a useful separate desktop is a "personal" desktop to isolate sensitive personal (e.g., financial) applications and data from the rest of the user's machine. Applications in such a desktop would be limited to those handling such personal matters, plus a few trusted standard ones such as basic word processing. These applications would also be isolated from all inter-process communication with applications on other desktops, and files created by them would be inaccessible from any other desktop. Network access in this desktop would be limited to secure, authenticated connections with a small number of trusted parties, such as the user's bank(s) and broker(s); no ad-hoc collaborations such as general browsing or Internet connections would be permitted. Similarly, the authentication credentials required to establish authenticated connections to these trusted parties would be isolated in this desktop. A user with such a separate desktop should feel comfortable using the same machine for other purposes without fear of exposing sensitive personal data or functionality to attackers.

The Enterprise Desktop

Like personal data, a user's work-related data and applications are best kept isolated from the rest of the user's machine. In an "enterprise" desktop, only enterprise-approved work-related applications would be allowed to run, and network access would be limited to secure authenticated connections to the organizational network or intranet (and hence to the rest of the Internet only through the enterprise proxy/firewall). This means that ad-hoc collaboration is severely limited (restricted to trusted hosts inside the intranet, or firewall-filtered traffic beyond that). Like in the personal desktop, all applications on this desktop would be "sandboxed" together, and denied inter- process communication with applications outside the desktop. The capability to authenticate to the enterprise network would also be isolated in this desktop. Such isolation would allow a user to access an enterprise Intranet safely from the same machine used for other, less safe activities.

Note that enterprise-based client-server applications actually benefit enormously from such isolation, because they typically allow the user's client machine to act in the user's name for server access. Thus if an insufficiently isolated client application opens a security hole in the client machine, it may implicitly open a hole in the server's security, by allowing unauthorized attackers access to the server as if they were at the same authorization level as the attacked client. On the other hand, if the client application and all associated access rights are isolated in an "enterprise desktop," then malicious or vulnerable applications introduced onto the client machine for purposes unrelated to the enterprise are no threat to the enterprise server's security.

The "Play" Desktop

For games, testing of untrusted applications, and other risky activities, a separate desktop should be available with full Internet access but absolutely no contact with the rest of the machine. There may be multiple play desktops; for example, a Java-like sandbox for untrusted network-based applets would look a lot like an instance of a play desktop.

This is the desktop in which ad-hoc collaboration can take place. We note that no special security policies or configurations need to be established for this desktop. We do not control access through a user-definable security policy, but through a system-imposed separation of desktops.

The Personal Communication Desktop

Since users are accustomed to dealing with e-mail, Web browsing, telephony/conferencing and other forms of personal communication in an integrated way (as opposed to, for instance, receiving e-mail in different desktops), these functions are best protected by collecting them in a single separate desktop. This desktop should run only trusted communications applications; those communications (e-mail, Web pages, and so on) which contain executable code (or data associated with non-communications applications) would have to be explicitly moved into some other desktop to be run or used. For example, a financial data file contained in an e-mail message from the user's bank would have to be moved into the personal desktop before being opened by the appropriate financial application. Note that some communications functions could also be performed in other desktops; for example, a Web browser in the enterprise desktop might be used to browse the enterprise Intranet (to which applications – including browsers – in other desktops would have no access).

3.4.4 How WindowBox Protects the User

Before we look at our implementation of the WindowBox model, let us recap how WindowBox can prevent common security disasters, which could be results of ad-hoc collaboration. Consider, for example, users who like to download games from unknown Web sites. Every now and then, one of the downloaded games may contain a virus, which can destroy valuable data on people's machines. If the game is a Trojan Horse, it might also inconspicuously try to access confidential files on the PC and send them out to the Internet. If the users were employing WindowBox, they would download the games into a special desktop, from which potential viruses could not spread to other desktops. Likewise, a downloaded Trojan Horse would not be able to access data in another desktop.

As a second example, let us now consider the recent spread of worms contained in e-mail attachments. For example, the Melissa worm (often called the "Melissa virus"), resends itself to e-mail addresses found in the user's address book. On a WindowBox-equipped system, users would open e-mail attachments in a desktop that is different from the desktop in which the e-mail application is installed. This might be because the attachments logically belong in a different desktop, or simply because the user judges them not trustworthy enough for the e-mail desktop. This could even be enforced by the fact that e-mail attachments simply cannot be opened in the communications desktop, so the user *has* to open them in another desktop. From that other desktop, the worm cannot access the e-mail application, or the network, to spread itself to other hosts.

3.5 Implementation

We implemented the WindowBox security architecture on a beta version of Windows 2000 (formerly known as Windows NT 5.0 Workstation). In our implementation, the various desktops are presented to the user very much in the manner of standard "virtual desktop" tools: at any given time, the user interacts with exactly one desktop (although applications running on other desktops keep running in the background). There are GUI elements that allow the user to switch between desktops. When the user decides to switch to a different desktop, all application windows belonging to the current desktop are removed from the screen, and the windows of applications running in the new desktop are displayed. Windows 2000 already has built-in support for multiple desktops. For example, if a user currently works in desktop A, and an application in desktop B pops up a dialog box, that dialog box will not be shown to the user until he or she switches to desktop B. Windows 2000 provides an API to launch processes in different desktops and to switch between them. We simply had to provide GUI elements to make that functionality available to the user.

However, the desktops provided by Windows 2000 do not, in any way, provide security mechanisms in the sense of the WindowBox security architecture. Our implementation therefore had to extend beyond what is offered in Windows 2000. In this section we describe these extensions.

Before explaining how we represent desktops as user groups, and what changes we made to the NT kernel to implement WindowBox security, we will briefly recap the Windows NT security architecture.

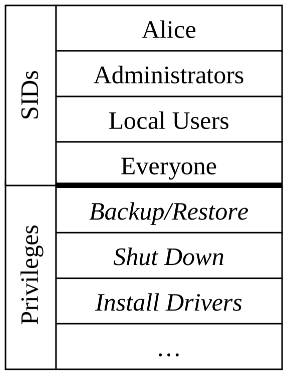

Figure 3.1: A sample access token

3.5.1 The Windows NT Security Architecture[3]

In Windows NT, every process has a so-called access token. An access token contains security information for a process. It includes identifiers of the person who owns the process, and of all user groups that person is a member of. It further includes a list of all privileges that the process has. Let's assume that Alice is logged on to her Windows NT workstation and has just launched a process. Figure 3.1 shows what the access token of such a process might look like: It includes an identifier (also called Security Identifier, or SID) of Alice as well as of all the groups she is a member of. These include groups that she has explicitly made herself a member of, such as "Administrators," as well as groups that she implicitly is a member of (like "Everyone"). Since she is an administrator on her workstation, her processes get a set of powerful privileges (these privileges are associated with the user group "Administrators"). The figure shows a few examples: The "Backup/Restore" privilege allows this process to read any file in the file system, regardless of the file's access permissions. The "shut down" privilege allows this process to power down the computer, *etc*.

[3]See "Inside Windows 2000" [43] for a more detailed look at the Windows 2000 security architecture.

1. *Allow*	Write	Administrators
2. *Deny*	Read	Alice
3. *Allow*	Read	Everyone

Figure 3.2: A sample DACL

Access tokens are tagged onto processes by the NT kernel and cannot be modified by user-level processes[4]. When a user logs on to the system, an access token describing that user's security information is created and tagged onto a shell process (usually Windows Explorer). From then on, access tokens are inherited from parent to child process.

The second fundamental data structure in the Windows NT security architecture is the so-called security descriptor. A security descriptor is tagged onto every securable object, *i.e.* to every object that would like to restrict access to itself. Examples of securable objects include files or communication endpoints. A security descriptor contains, among other things, the SID of the object's owner and an access control list. The access control list (also called Discretionary Access Control List, or DACL) specifies which individual (or group) has what access rights to the object at hand. It is matched against the access token of every process that tries to access the object. For example, consider a file with the access control list shown in Figure 3.2. What happens when Alice tries to access this file? Since her access token includes the Administrators group, she will have write access granted. However, the DACL of this file explicitly denies read access for Alice (let's forget for a moment that Alice's access token also contains the Restore/Backup privilege, which enables her processes to override this decision). Because the order of the DACL entries matters, it is not enough that the group "Everyone" (of which Alice is a member, as her access token specifies) has read access. The entry that denies Alice read access comes first, effectively

[4]This is an oversimplification. In reality, there are some limited operations that a user-level process can do to an access token: It can switch privileges on and off and even (temporarily) change the access token of a process (for example, when a server would like to impersonate the client calling it). However, a process can never, of its own volition, gain more access rights than were originally assigned to it by the kernel.

SIDs	Alice
	Local Users
	Everyone
	Administrators
	Alice.Personal
	Alice.Enterprise
	Alice.Play
Privileges	*Shut Down*
	Install Drivers
	Backup/Restore
	…

Figure 3.3: A sample access token with desktop SIDs

allowing everyone but Alice read access. We can see that the combination of access tokens and security descriptors provides for an expressive and powerful mechanism to specify a variety of access policies.

Only the kernel can modify the security descriptor of an object, and it will only do so if the process that is requesting modifications belongs to the owner of that object. Also, the access check described above happens inside the kernel. No user process can, for example, go ahead and read a file if the file’s DACL forbids this.

3.5.2 Desktops as User Groups

Apart from the graphical representation to the user, we internally represent each desktop as a user group. For example, if Alice wanted three desktops for her home, work, and leisure activities, she could create three user groups called Alice.Personal, Alice.Enterprise, and Alice.Play. She would make herself a member of all three groups[5]. Now, whenever she logs on to her computer, her access token would look like the one shown in Figure 3.3. Note that the SIDs that represent her desktops are added to the access token. This happens

[5]In our prototype, this process is automated and happens when a new desktop is created.

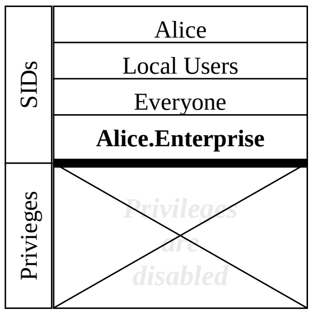

Figure 3.4: A sample access token of a desktop process

automatically since Alice is a member of all these groups. We call these SID's desktop SIDs. They are marked as desktop SIDs in the access token (denoted by bold typeface in the picture), but are otherwise just normal group SIDs.

In our implementation, we create a desktop for every desktop SID in the access token when the user logs on. Continuing our example, when Alice logs on, we create three desktops. In every desktop, we start a shell (Windows Explorer)[6]. However, we limit what each shell can do by giving it a restricted token (the restricted token API has been introduced in Windows 2000 and allows processes to limit their own, or their children's, privileges): First, we remove all privileges from the access token. Then we remove all desktop SIDs except the one representing the desktop for which we are preparing the access token[7]. Lastly, we remove the "Administrators" SID from the token in a move to restrict access to system files, which usually allow write access to the Administrators group. Each shell is launched with this restricted token. Figure 3.4 shows the access token of the shell running in Alice's enterprise desktop.

[6]For the record we should mention that there will also be a fourth desktop that serves as a "root" desktop in which all applications have full privileges and access to the system. Alice should stay away from that desktop for her day-to-day work, but can use it for administrative tasks.

[7]The diligent reader will object that removing a group SID from an access token doesn't necessarily restrict the process' rights. Windows 2000 does the right thing: The SID is not completely removed, it is "disabled for positive DACL entries": If you disable the SID "Group A" in an access token, and then try to open a file that allows only access to Group A, this access will not be allowed. However, if you try to access a file that explicitly denies access to Group A, access *will* be denied.

When Alice now starts applications or processes in one of her desktops, they will inherit the restricted token of the desktop's shell. Note that this also holds for ActiveX components and other executable content downloaded from the network, which runs inside descendants of the desktop's shell.

With the system described so far, we could already implement some kind of WindowBox security. If Alice judiciously restricted access to some files to the user group Alice.Enterprise (and only to that group), these files could only be accessed by applications running in the enterprise desktop. However, one of the major goals of the WindowBox security architecture is to relieve the user from the burden of making difficult access control decisions, and to "automate" this process. Furthermore, while the user can modify the DACLs of files, this is not true for all objects. For example, processes in one desktop can still communicate with processes in another desktop, and there is nothing that the user can do about this. For this reason, we introduced the concept of "confined" objects.

3.5.3 Confined Objects

A confined object is an object that belongs to a certain desktop. The idea is that an object confined in desktop A should not be accessible by any process from desktop B. We confine objects by tagging their security descriptor with the SID of the desktop they should be confined to. For example, to confine a file to Alice's enterprise desktop, we would add the SID Alice.Enterprise to the file's security descriptor. This extension of the security descriptor is our first modification to the NT kernel.

Our second modification makes sure that objects automatically become confined: Whenever the kernel creates a securable object (*i.e.*, an object with a security descriptor) such as a file or a communications endpoint on behalf of a process, we confine that object to the desktop that the creating process runs in. Note that not all objects have to be confined. There are processes on the system (for example system services) that do not belong to any desktop because they are not descendants of any of the desktops' shells. The kernel can also create objects on its own behalf.

Our third modification concerns the access check performed in the kernel. The original access check implements the semantics of the access control list of an object with respect to the accessing process' access token as explained above. We changed the access check as follows:

1. Is the object confined? If so, go to 2; otherwise go to 3.
2. Check whether the process' access token contains a SID that is equal to the desktop SID the object is confined to. If so, go to 3; otherwise deny access.
3. Perform a normal access check.

Our modifications implement the WindowBox security architecture: The desktops are completely sealed off from each other. For example, a file saved by a word processor in one desktop will be confined to that desktop and cannot be accessed by any process from another desktop, including potentially malicious applications downloaded from the outside world. This goes beyond what default ACLs in Windows 2000 or the umask feature in UNIX offer: First, the confinement cannot be undone by an ordinarily privileged process (see below). Second, there is no customizable setting that the user has to decide on (*i.e.*, what should the default ACL or umask be?). Object confinement is mandatory, and cannot be customized, or mismanaged, by the user. Also note that confined files are usually visible to other desktops, they are merely inaccessible. An access to a file confined to another desktop would fail in the same way that access to file belonging to another user would. However, by the same token, this also implies that files within a directory in another desktop will *not* be visible.

Notice that processes also become confined as they are created (they, too, are securable objects).

Once an object is confined, it takes a special privilege to “un-confine” it (by removing the confining SID from its security descriptor) or to confine it to another desktop. Since we strip all privileges from the shells (and their children) in each desktop, no application in any desktop can move objects from one desktop to another. However, in each desktop we provide one process that is not restricted in the same way as the shell and its children are. This process provides the GUI to switch to other desktops (the restricted processes in the desktops would not even have enough privileges to make that switch). That privileged process also serves as a COM server exporting the service to move objects between desktops. Every process can connect to that server and ask it to move an object from one desktop to another. The server can then decide whether or not to do so. In our implementation, it asks the user for confirmation before any object (such as a file) is moved between desktops.

3.5.4 Restricting Network Access

The system, as described so far, can already safeguard against a number of attacks if used consistently. Our user Alice should never do her finances in her play desktop, for example. Nor should she visit untrusted Web sites while she is in her enterprise desktop. To encourage her to abide by the latter rule, we have restricted network access for processes running in desktops.

We modified the kernel to deny any network access to a process running in a desktop (*i.e.*, a process with a desktop SID in its access token). However, we also added a layer in the (user-space) network stack that relays network calls of a desktop process (which would fail in the kernel) to the privileged COM server mentioned above. This privileged process can connect to the network, but will only do so if the requested network address satisfies the desktop’s policy (*e.g.*, it is explicitly included in a list of permitted addresses). If so, it connects to the requested network address and returns the handle representing the network connection back into the unprivileged process.

In our implementation, users can specify a different network access policy for each desktop. The mechanism used is powerful enough to express policies like: “allow only connections to the corporate intranet,” “allow only connections to www.mybank.com,” “deny any network access,” *etc*.

3.5.5 Security Analysis

How secure is WindowBox? This question does not have a generic answer. Rather, we need to ask how secure a specific implementation of the WindowBox security model is. A “secure” WindowBox implementation does not allow malicious code to affect another desktop. For example, a virus should not be able to infect files in a different desktop, a data-gathering Trojan Horse should not be able to read files in other desktops, *etc*. In other words, there should be no *channels* between desktops, which malicious code could exploit. The work on covert channels has shown that it is not possible to close every covert channel. As far as we are concerned, a covert channel can be defined as a means by which information could leak from one desktop to another. Note that this is considerably less powerful a channel than one which a virus could actually exploit to propagate itself to a different desktop. For that purpose, a virus would have to be able to write a file in one desktop, and then cause that file to be executed in a different desktop. While our situation is not as hopeless as with covert channels, we still believe that it is impossible to get formal assurance that no dangerous channels exist. Windows 2000 is too complex a system for us to hope to model it in a way that would yield relevant statements about a given WindowBox implementation.

What, then, can we say about the security of our WindowBox implementation? We tried to make our implementation as secure as possible by implementing it at as “low” a level as possible inside the kernel. The rationale is that every malicious program has to go through certain parts of the kernel - most notably, the access control reference monitor - in order to do anything useful, including any attempt to access another desktop. Therefore, we placed the WindowBox enforcement code inside that reference monitor.

There are, however, covert channels left between desktops in our current implementation. For example, files that are not confined to a desktop could potentially be writable by one desktop and then readable by another desktop. To prevent this covert channel from turning into a channel that a virus could exploit, we made sure that none of the executable system files are writable from any desktop. Ultimately, we believe that a lot of scrutiny will be necessary to find and deal with other potential channels, and that for this reason a production version of WindowBox would need extensive testing and prolonged exposure to the security community.

The last part of the answer is that a WindowBox implementation would be most secure on top of an operating system that was designed in anticipation of this kind of security model. For example, we mentioned above that certain applications should be installed in only one desktop, or that applications should be installed separately in multiple desktops. Windows 2000 does not really allow us to do that. For example, applications written for Windows 2000 like to keep configuration data in the (system-wide) registry or use well-known (system-global) files to store information. Ideally, the operating system should be designed from the ground up with the WindowBox model in mind, thus eliminating potential cross-desktop channels that malicious programs could exploit. However, a few key modifications to Windows 2000 would go a long way towards supporting WindowBox more securely; for example, some parts of the registry could be replicated, with separate copies for each desktop, to allow applications to install transparently on some desktops but not others.

Finally, we would like to remind the reader that no system is more secure than the decisions of its user or administrator, and that defining a security model in which the protection of sensitive data is relatively convenient (as is the case in WindowBox) creates the possibility of implementations converging towards security that is not only free of serious intrinsic holes, but also usable enough to avoid many of the types of holes introduced by the unsafe practices of users battling cumbersome systems.

3.6 Related Work

We are not the first to recognize the specific security requirements of a ubiquitously networked world, especially in the light of mobile code. A standard goal is to prevent mobile code, or compromised network applications, from penetrating the system. The generic term for ways of achieving this goal is "sandboxing." The term was first used in Wahbe et al.'s paper [46] to describe a system that used software fault isolation to protect system (trusted) software components from potentially faulty (untrusted) software components. Perhaps the best-known example of sandboxing is the Java Virtual Machine. It interprets programs written in a special language (Java bytecode). It can limit what each program can do based on who has digitally signed the program and a policy specified by the local user. The biggest drawback of the Java approach is that it can sandbox only programs written in Java.

Our system sandboxes processes regardless of which language they have been written in. That, too, is not new. Goldberg et al. show how any Web browser helper application can be sandboxed [25]. Their work, however, targets only processes that are directly exposed to downloaded content, and requires expertise in writing and/or configuring security modules.

Our system has certain resemblance to role-based access control in that one could think of our desktop SIDs as a user's different roles. In fact, in one paper [41], Sandhu et al. imagine a system in which "a user might have multiple sessions open simultaneously, each in a different window on a workstation screen." In their terminology, each "session" comprises a certain subset of the user's roles. Hence, in each window the user would have a different set of permissions.

Another concept with similarities to our desktops is that of “compartments” in Compartmented Mode Workstations (CMWs). CMWs are implementations of the Bell-LaPadula model [8] found in high-security military or government systems. Like CMW, we introduce “mandatory” security to an otherwise “discretionary” security model. Zhong explains how vulnerable network applications – such as a Web server – can be made less of a threat to the rest of the system if they are run in special compartments, shielding the rest of the system [53]. The WindowBox security model is in some sense weaker than the Bell-LaPadula model. We are merely trying to assist the user in separating his or her different roles. For example, covert channels from one desktop to another are much less of a concern to us than they are for a Compartmented Mode Workstation: In CMWs, an application voluntarily surrendering its data is a problem, in WindowBox it is not (it is simply not part of the threat model).

Domain and Type Enforcement (DTE) can also be used to sandbox processes in a way similar to ours. Walker et al. explain how they limit what compromised network applications can do by putting them in “domains” that don’t have write access to certain “types” of objects, for example system files [47].

What distinguishes our work from the long list of other sandboxing approaches is that all of the above use sandboxing as a flexible, configurable technique to enforce a given – usually complicated – security policy. We argue that figuring out a complex, customized security policy is too difficult a task for most users to handle. In contrast, in our system we do not use a general sandboxing mechanism to implement specific security policies. In fact, we don’t have any security policy in the traditional sense, except for the requirement of strict separation of desktops. It is this simplicity of the model that enables users to go out and connect to unknown and possibly dangerous sites without having to go through tedious security configurations. Therefore, users avoid mistakes and can enjoy ad-hoc collaboration while their sensitive data remains protected.

Another area of related research is that of user interfaces and security. Recently, more and more people have realized that poor user interface design can seriously jeopardize security [50]. We very much concur with the thrust of that research. Our approach, though, is more radical. Instead of suggesting better user interfaces for existing security tools, we propose a completely redefined security model. One of the features of WindowBox is that is naturally lends itself to a more intuitive user interface for security management. Users have to understand, and learn tools that visualize, only *one* concept - the fact that separated desktops can confine the potentially harmful actions of code.

3.7 Conclusions

In this chapter, we argue that existing security mechanisms – while possibly adequate in theory – fail in practice because they are too difficult to administer, especially for a networked personal computer or workstation under the control of a non-expert user. We present an alternative to this dilemma, WindowBox, a security model based on the concept of complete separation. While it has similarities to some existing security mechanisms, it is unique in that we do not try to provide a general mechanism to enforce all sorts of security policies. In contrast, we have only one policy – that of complete separation of desktops. We believe that this “policy” is easy to understand and has promise for the connected desktop.

It removes the threat of ad-hoc collaboration because it allows users to confine the possibly dangerous results of ad-hoc collaboration to a single desktop, while keeping sensitive resources in the other desktops secure.

The WindowBox model and prototype raise several interesting questions: What kinds of hidden security vulnerabilities might they contain, and how might they be eliminated? For instance, instead of using our own user interface, should we have used Window 2000’s “Secure Attention Sequence”[8] to make sure that an application cannot trick the users into

[8]This is the Ctrl-Alt-Del feature, which provides a trusted path to an unforgeable screen.

believing they are in a different desktop than they really are? How usable is a multiple-desktop environment for average users? (We conjecture that users' typical activities divide themselves up naturally in ways that correspond well with distinct desktops, but we have done no large-scale usability testing.) Should the separation be branched out into other parts of the system, *e.g.*, would it be useful to have a separate clipboard for every desktop?

These questions aside, we have presented in this chapter a new security model based on the idea of complete separation. While that idea itself is not new, we apply it to a world of ad-hoc collaboration and show how users can use that model to protect themselves from the threat that comes with it. We have also demonstrated the practicality of this approach by implementing a prototype for a mainstream operating system.

Chapter 4

Splitting Trust

4.1 Introduction

In the previous two chapters, we have looked at two different ad-hoc collaboration scenarios. In Chapter 2, we described how servers can specify policies for client access in an open system, where privileges might be delegated to clients initially unknown to the servers. In Chapter 3, we explain how a vastly simplified security model can make it easier for average users to protect themselves from potentially dangerous results of ad-hoc collaboration.

In this chapter we look at an entirely different instance of ad-hoc collaboration. Computational power is currently finding its way into a diverse array of devices. Most prominently, palm-sized computers are becoming more and more ubiquitous. In the future we will find more and more personal devices to be "computationally enabled," *i.e.* able to perform computations and communicate with their owner through a user interface.

However, many of these personal devices will be constrained by their form factor. If they are to be carried around, they cannot be as big as a personal computer, or laptop. Likewise, low power consumption and small size will always dictate lower computational power, at least per dollar, than full-sized computers. These drawbacks make it necessary that hand-held computers can collaborate with (bigger) personal computers in order to perform computational services for their owner. For example, an important feature of today's hand-held computers is their ability to *synchronize* their data with personal computers. That way, users can enter data comfortably at home, through a full-sized keyboard, using a big monitor, and – after synchronizing – take that data "on the road." Note that the synchronization is a "collaboration" in the sense of the term as used in this dissertation.

What if the owner of a hand-held computer wanted to connect his or her device to an unknown (or untrusted) computer, *e.g.* a public terminal in an airport lounge? In this case, the collaboration is ad-hoc, and we need to start worrying about the security implications. An untrusted computer should not be able to read sensitive information off personal devices connected to it, nor should it be able to manipulate the data stored on them. Nonetheless, we might want to enable collaboration between the personal hand-held computer and the public and untrusted terminal.

Consider the case where the hand-held computer holds private keys that can be used to authenticate e-mail messages or online connections. We might want to use a public terminal to compose an e-mail message, and digitally sign it. Obviously, exposing the private keys to the public, untrusted terminal poses a security risk. On the other hand, composing a message on a small palm-sized computer is tedious. Therefore, we would like to be able to compose the message on the terminal, but sign it on the hand-held computer, very much like a smart card. Section 4.2 explains in some detail the implementation of such a system for the 3COM PalmPilot. In the following sections, we describe a more general architecture for *splitting trust*, which enables developers to write applications that can run both on a trusted and, at the same time, on an untrusted device.

While we have designed the splitting trust architecture with trusted hand-held computers and untrusted more powerful PCs in mind, it can be applied to a variety of ad-hoc collaboration scenarios. Consider for example the currently very fashionable field of content distribution and caching for the World Wide Web. The main goal there is to move content as close as possible to the ultimate consumer of the content, in order to improve download times. If the content is created dynamically by an application, then it might make sense to move the content-generating application close to the consumer. However, parts of the application may be sensitive (*e.g.* a huge customer database). In that case, we have a splitting trust scenario: Part of the application can be executed on an untrusted node close to the content consumer, but part of the application must be executed in a trusted environment.[1]

4.2 PilotKey[2]

In this section we describe in some detail the PilotKey project. In PilotKey, we implemented smart card functionality on a 3COM PalmPilot. A smart card is an example for splitting trust. Because the cryptographic material on the card cannot be retrieved, certain operations using that cryptographic material have to happen *on* the card itself, splitting any software application that wants to make use of that cryptographic material into the part that has to run on the smart card and the part that can run somewhere else.

Smart cards are convenient and secure. They protect sensitive information (*e.g.*, private keys) from malicious applications. However, they do not protect the owner from abuse of the smart card: An application could for example cause a smart card to digitally sign any message, at any time, without the knowledge of the owner. Even if the user needs to supply a PIN or password to unblock the smart card, the smart card can still be abused. For example, a malicious computer could ask the user for that PIN when he or she is about to send a signed e-mail message (which is something that the user would expect), but then use the smart card to sign something other than the e-mail message the user is about to send. The problem is that smart cards do not have a user interface that would signal to the user

[1]Thanks to Vinesh Verma at IBM Research for suggesting this application of splitting trust.

[2]A version of this section was published in an earlier paper [6].

when, and for what, the smart card is being used. Therefore, we contrast smart cards with our PilotKey implementation on the PalmPilot, which is a palm-sized hand-held computer. Hand-held computers can communicate with the user directly and therefore do not exhibit the above mentioned problem.

4.2.1 Overview

Public key systems (like RSA [38] or DSA [35]) promise to play a major role in the evolving global networked community. Since they solve the key distribution problem, they are especially useful for a big, open network like the Internet. In a public key system every individual possesses two keys: a *public* and a *private* key. The public key is published and known to everyone. The private key is known only to its owner, and kept secret from everyone else. Public keys are used to encrypt messages and verify signatures, and private keys are used to decrypt messages and produce signatures. If a thief steals your private key, he can not only read confidential messages sent to you, he can also impersonate you in electronic conversations such as e-mail, World Wide Web connections, electronic commerce transactions, *etc*.

People should therefore protect their private keys adequately. One way to do so are *smart cards*. Smart cards are small[3], tamper-resistant devices that can be connected to a PC and store private keys. The PC cannot learn the private key stored on the smart card, it can only ask the card to perform certain cryptographic functions for which the private key is needed, such as calculating a digital signature or decrypting an e-mail message. These functions are executed on the smart card. This way, the connected PC can be used to engage in electronic conversations on behalf of the smart card owner, but it does so without knowledge of the owner's private key.

[3]Usually, smart cards have the form factor of a credit card.

Contrast this with a scenario in which the private key is stored on the PC itself (which is the most common case today). Although the key will probably be protected by a password, the PC itself could fall victim to an attack from the outside ([27]). If and when that happens, it is usually just a question of determination for the intruder to break the protection mechanism and learn the private key. Even worse, if the application that uses the private key turns out to be malicious (it could be infected by a virus, or turn out to be a tampered-with version of a well-known application, or some other Trojan horse), it can get to know the private key without any further ado.

Smart cards protect users from security breaches of that kind. They are also quite convenient. Since they allow users to carry around their private keys with them, and at the same time connect to any PC, they allow users to engage in electronic conversations from any PC.

However, smart cards also have problems. Imagine, like in the scenario described above, that the PC to which the smart card is connected has been compromised, and that an e-mail application that talks to the smart card is malicious. While the application will not be able to learn the private key on the smart card, it may do other things that are almost as bad. For example, when the user composes an e-mail message to `president@whitehouse.gov` that reads

> I support you,

then the malicious e-mailer could, instead, send out an e-mail that reads

> I conspired to kill you,

and could even have the smart card sign the latter. The digital signature might stand up in court and could get the owner of the private key into a lot of trouble.

The problem here is that the smart card does not have an interface to the user. A smart card is designed to protect the private key even if it is talking to a malicious application.

However, it does not protect against *abuse* of the private key as shown in the example above. If the smart card had a user interface it could have shown – for verification purposes – the message it was asked to sign, and the user would have learned that the e-mailer was trying to frame him.

Recently, hand-held computers like the 3COM PalmPilot emerged in the marketplace. They are almost as small as smart cards, have superior computing power[4], and provide an interface to the user. In the next sections we report on how we implemented smart card functionality for the 3COM PalmPilot (which we will simply call “Pilot” from now on). We describe how even in our initial attempt we achieved certain security properties that normal smart cards cannot deliver.

4.2.2 PKCS#11

We have implemented a PKCS#11 library for Netscape Communicator. PKCS#11 [39] is a “standard” drafted by RSA Data Security Inc. It defines sixty-odd prototypes for functions that together can be used to perform a wide range of cryptographic mechanisms, including digital signatures, public key ciphers, symmetric key ciphers, hash functions *etc*. The standard is designed with smart cards in mind[5]: The caller can have the PKCS#11 library perform certain functions, like producing a signature, without ever knowing the private key that is used in the process. In this section we explain how PKCS#11 works, how Netscape Communicator uses it, and what the design of our PalmPilot implementation is.

PKCS#11 describes function prototypes and semantics for a *library*, which we will call *PKCS#11* or *cryptoki library*. An application can bind to a cryptoki library and call into its functions to perform certain cryptographic operations. The PKCS#11 world is populated by *objects*, which are sequences of attribute-value pairs. For example, Figure 4.1 shows an object.

[4]With the exception of certain cryptographic operations - see Section 4.2.3.

[5]It is also called the *cryptoki*, or cryptographic token interface, standard.

Attribute	**Value**
OBJECT_CLASS	PRIVATE_KEY
KEY_TYPE	RSA
TOKEN	YES
EXTRACTABLE	NO
LABEL	BOBS_PRIVATE_KEY
PRIVATE_EXPONENT	8124564…
MODULUS	7234035054…
⋮	⋮

Figure 4.1: A private RSA key

There are five different classes of objects: certificates, public keys, private keys, secret (*i.e.*, symmetric) keys, and "data" objects. Objects can be created temporarily (*e.g.*, a session key for an SSL connection), or can be long-lived, in which case they are assumed to be stored on a "cryptographic token." The example in Figure 4.1 describes a private RSA key. It is stored on a smart card and marked "unextractable." This means that calls into the cryptoki library that try to read the values of various attributes of this key will fail. Hence, there is no way for the application to learn the value of this secret key (assuming that in fact it is stored on a smart card and not in the same address space as the application).

How can the application use this key to sign a message? The cryptoki library returns *handles* to the application, which are usually small integers and uniquely identify objects. Although the application will not be able to learn the private exponent of the key above, it can for example ask the cryptoki library if it knows of an object with the label "BOBS_PRIVATE_KEY." If such an object is found, a handle H is returned to the application, which can then ask the cryptoki library to sign a certain message with the object identified by H. Using the information known to it, the cryptoki library can sign the message and return the signature to the application. If the object is stored on the smart card, and the handle, message and signature are the only things exchanged between smart card and application, then the private key is safe. Object search and signature calculation have to happen on the smart card in this case.

PKCS#11 defines many more functions besides object search and message signing. The application can learn which cryptographic mechanisms are supported by the library, create and destroy objects, encrypt and decrypt messages, create keys, and more.

The cryptoki library will not perform certain functions (like signing a message) unless the user is *logged on* to the cryptographic token. Usually, the user has to provide a PIN or password to log on to the token. PKCS#11 distinguishes between tokens with or without a *trusted authentication path*. A token with a trusted authentication path is a smart card to which the user can log on directly, *e.g.*, by means of a numeric key pad on the card. If the card does not have a trusted authentication path, then the application on the PC has to gather the PIN or password from the user, and send it to the smart card. We note that this is less secure than a trusted authentication path: The application can learn the PIN or password needed to "unlock" the private key on the smart card. Smart cards usually do not have a trusted authentication path.

Netscape Communicator supports PKCS#11, which means that smart card vendors can provide a cryptoki shared library [36]. Communicator will then bind to that library and use it for certain cryptographic operations. In particular, Communicator uses the library for key pair generation, S/MIME encrypted e-mail [16, 17] and client authentication in SSL connections [23]. For S/MIME the cryptoki library has to support RSA [38], DES [34], RC2 [37], and Triple-DES [42]. Communicator supports RSA key pair generation only, so client authentication – although technically feasible with any signature scheme – is done using RSA signatures.

We implemented a cryptoki library that supports the above mentioned ciphers and signature schemes. The shared library that plugs into Communicator serves only as a

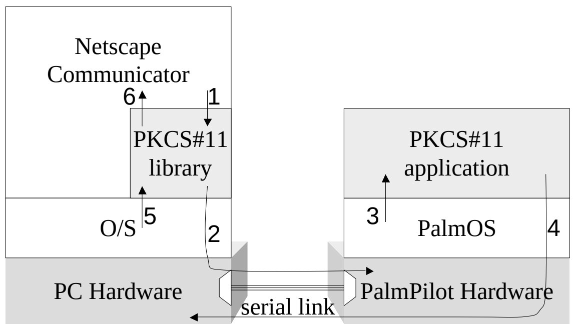

Figure 4.2: Overview of our PKCS#11 implementation

dispatcher of requests to the Pilot. For the Pilot, we have written an application that receives those requests, performs the necessary functions and sends the results back to the PC.

Figure 4.2 shows how our implementation works. The PC and the PalmPilot are connected through a serial link. We implemented two pieces of software: a PKCS#11 plug-in library for Netscape Communicator and a PKCS#11 application for the PalmPilot. Communicator calls into our library when some cryptographic operation needs to be performed (1). If the request cannot be handled by the library itself, it forwards it to the PalmPilot (2). On the PalmPilot, the operating system notifies our PKCS#11 application when a request arrives (3). The application processes the request and returns the result to the PC (4), where it is received by the cryptoki library (5). The library will then return this result or a result based on what it received from the PalmPilot back to Communicator (6).

It is worth pointing out what the trusted components are in this model: We trust that the PKCS#11 application on the PalmPilot is not tampered with and performs correctly. We trust in the same way the operating system on the PalmPilot and its hardware. On the other hand, we do not have to trust Communicator, the operating system on the PC or its hardware to collaborate with us. We do not even trust that the PKCS#11 library does what it is supposed to do. The PKCS#11 application on the Pilot is written defensively and works[6] even in the case where the PKCS#11 library is replaced by malicious code.

4.2.3 The Implementation

Our implementation runs on Windows 95, Windows NT, and Linux for the Communicator plug-in, and on a 3COM PalmPilot Professional for the Pilot side. It practically turns the Pilot into a smart card. For the cryptographic functions we used the PalmPilot port of SSLeay [52]. In the following section we will highlight a few points of our particular implementation.

Key Pair Generation

To create an RSA key pair, we need a good source of randomness. In a traditional smart-card, if the attacker knows the random number generating algorithm and initial seed, we cannot hide the private key from him. He can simply perform the same computation as the smart card. PKCS#11 provides functions to reseed the random number generator on the card, but the application never has to call them.

On the Pilot, we can use external events to provide randomness that the application on the PC cannot observe. When the Pilot generates a new key pair, we ask the user to scribble randomly on the screen of the Pilot, thus providing random bits we use for seeding the random number generator.

Logging on to the Pilot

The Pilot is a cryptographic token with a trusted authentication path. Therefore, the user does not enter his password through Communicator, but directly into the Pilot. This way Communicator cannot learn the PIN or password needed to unlock the private keys. If it

[6]"Works" means that it does not leak any information it is not supposed to leak. It does not mean that the system does not crash, for example.

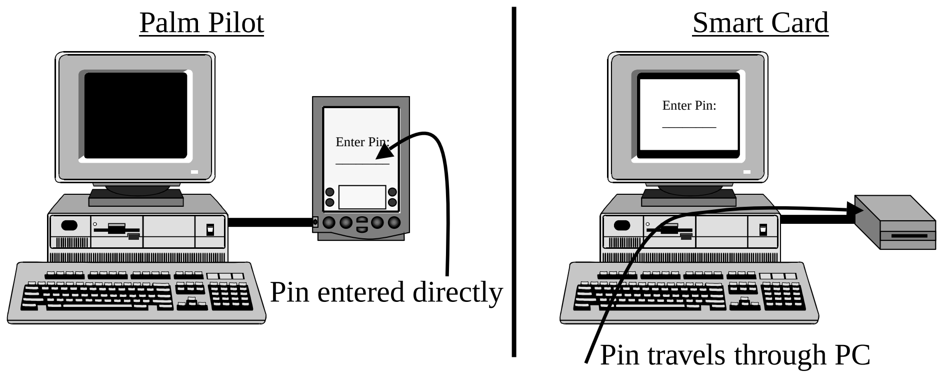

Figure 4.3: Information flow for PIN, contrasting traditional smart card and PalmPilot.

knew the PIN or password, it could try to use the Pilot without the user's knowledge or – even worse – make the PIN known to adversaries who might later find an opportunity to steal the Pilot from its owner. Figure 4.3 shows the difference in information flow between a traditional smart card and our PalmPilot implementation.

Signing E-mail Messages

In order to send a signed e-mail message, all the user needs to do is log on to the Pilot and put it into its cradle, which will connect it to Communicator. Communicator will ask the Pilot to sign the message and also retrieve a certificate stored on the Pilot (unlike private keys, certificates are extractable objects and can therefore be retrieved by the application). Then, Communicator adds the signature and the certificate to the message according to the S/MIME standard, and sends it off.

Ideally, we would like the Pilot to display the message on its display for verification before it is signed. Alas, because of an implementation peculiarity of Netscape Communicator, this is not possible (see footnote 9 on page 90).

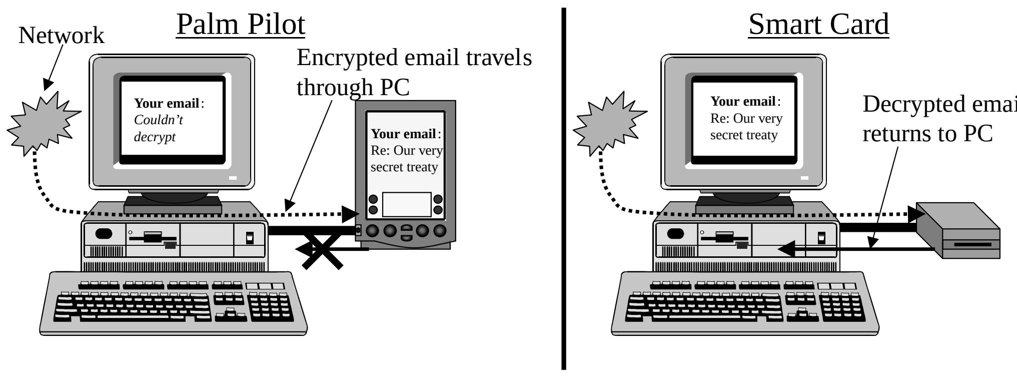
Palm Pilot
Network
Encrypted email travels through PC
Your email: Couldn't decrypt
Your email: Re: Our very secret treaty
Smart Card
Your email: Re: Our very secret treaty
Decrypted email returns to PC

Figure 4.4: Information flow for reading encrypted e-mail, contrasting traditional smart card and PalmPilot.

Receiving Encrypted e-mail

When Communicator receives a message that has been encrypted with the user's public key, it will first ask the Pilot to use the corresponding private key to unwrap the symmetric (RC2 or DES) key used to encrypt the contents of the message. Depending on the preference setting on the Pilot, it will either signal an error, or unwrap the symmetric key and store it as an unextractable object on the Pilot. In the former case, Communicator will ask to "decrypt" a certain string which happens to be the wrapped key, and hence obtain the symmetric key needed to decrypt the message.[7] In the latter case, Communicator will send the encrypted message to the Pilot and ask it to decrypt it using the symmetric key just unwrapped. The Pilot will decrypt the message, display it on its screen, and send bogus information back to Communicator. So, depending on the preference settings on the Pilot, Communicator may or may not be able to see the decrypted contents of a message[8].

Users that have little trust in the PC that their Pilot is connected to can use this feature to read encrypted e-mail on their Pilot without risking its contents to become known to

[7]Note the difference between "unwrappinng" and "decrypting." An unwrapped key is not returned to the caller (only a reference to it), while a decrypted key is.

[8]If the Pilot agrees to unwrap the key, it will refuse to "decrypt" it, so that even a malicious version of Communicator would not be able to decrypt the message on its own.

anyone. This cannot be done with traditional smart cards since they lack a user interface. See Figure 4.4 to illustrate this point.

Performance Issues

A Pilot is not as fast as a smart card when it comes to long integer arithmetic. Signing/decrypting a message with a 512 bit key (the key size in exportable versions of Communicator) takes about 5 seconds. Signing/decrypting with a 1024 bit key takes about 25 seconds. (These measurements were made with version 2.01 of pilotSSLeay.)

Creating a 512 bit RSA key pair takes a couple of minutes. Creating a 1024 bit key pair takes 30 minutes. These numbers may vary since a randomized algorithm is used to find the primes.

Communicator often calls into the cryptoki library to re-assure that the token is still connected, to exchange information, to search for objects, *etc*. Through a rather slow serial link, this conversation takes up a lot of time. We built in Communicator-side caches for most of the information that Communicator learns from the Pilot. So what users experience is a flurry of messages going back and forth between PC and Pilot when they start using it during a session. However, after a while the Pilot will be contacted only to actually perform a cryptographic operation, and the time experienced by the users closes in on the numbers given above.

4.2.4 Protecting Sensitive Data

How safe is the user's sensitive data if the Pilot gets into the hands of adversaries? In our implementation, sensitive parts of the private key are stored encrypted in non-volatile RAM. We derive a DES key from the PIN or password that the owner uses for logging on to the Pilot and use it to encrypt the sensitive parts of the private key. Later, they are only decrypted just before they are used, and erased after they are used. When the user logs off, the PIN or password and the derived DES key are erased.

It is very easy for an adversary to read the encrypted key out of non-volatile RAM. He then needs to perform an attack on the DES encryption, or alternatively a dictionary attack on the PIN or password. Since only the actual bits of sensitive data (which are unknown to the attacker) are encrypted, a straightforward known plaintext attack is not possible. Instead, the attacker would have to perform an expensive multiplication to test whether he had found the correct private key. We therefore assume that the private key is reasonably safe, with the weakest link in the chain probably being the PIN or password, which could be too short or part of a dictionary.

For a device like a Pilot, a different threat is more imminent. Our PKCS#11 application usually shares the Pilot with many other applications. Since there is no memory protection, other applications might be able to read the decrypted private key if they manage to interrupt the PKCS#11 application just at the right time. So, for better security users should be very careful about what kind of applications they install on their Pilot.

To alleviate this situation, the PalmPilot would need a comprehensive security architecture. First, the operating system should enforce memory protection and add access control to its resources such as the databases in non-volatile RAM. Second, downloading software onto the PalmPilot should be restricted; we could for example imagine a password needed to download new software onto the Pilot. Third, downloaded software should be put in sandboxes that are enforced by the operating system. There should be a way to relax sandboxes, perhaps based on digital signatures, in order to give certain applications access to more functionality. With a system like this, a user could for example download a game from an unknown source and place it into a tight sandbox from where it cannot possibly learn things about the other applications running.

A number of vendors and research projects have tried to address the problem of operating system security in hand-held computers. Recent versions of Microsoft's Windows CE operating system include a feature that allows only applications signed by designated principals to be considered "trusted." Untrusted applications are barred from calling a set of system calls considered sensitive. Sun and 3COM have announced that Sun's Java Platform 2, Micro Edition, will be available for the 3COM Palm series of devices. Also, since smart cards themselves have become more powerful, the operating systems on smart cards now tend take into account the scenario of multiple, mutually suspicious applications running on the same card. The techniques used on modern smart cards (secure file systems, *etc.*) could be applied to hand-held computers. Furthermore, techniques based on complete separation, like the WindowBox security model introduced in Chapter 3, could be used to seal off different parts of the operating system on the hand-held computer from each other.

Once this security architecture is in place, we can turn to physical security: Now that it is impossible for an adversary to inject software into the Pilot, we need to make sure that the data cannot be read out in some other way. To prevent this, the Pilot could be equipped with tamper-resistant hardware, which would make it difficult to obtain the data in question by physical means. Equipping a Pilot with tamper-resistant hardware is not too hard since it is bigger than a smart card and there is more space to accommodate tamper-aware protection mechanisms. But as long as there is an easy way to install arbitrary applications on the Pilot, the additional cost of a tamper-resistant hand-held computer would not be justified.

4.3 The Splitting Trust Framework

So far, we have described a particular system that implements smart card functionality on a PalmPilot. We have also seen that a hand-held computer has potential for better security, since it provides a direct user interface. The fundamental reason for the desirable security features is that the Pilot is more *trusted* than the PC. Just like a smart card, we always carry it around with us and are fairly certain that no-one tampered with it. We have a good overview of the software running on the Pilot and usually know where it came from. Contrast this with a PC, which may easily run hundreds of different processes at a time. These different processes may open up vulnerabilities to attack. Moreover, if we use our Pilot with an arbitrary PC, we do not know whether that PC has not been specifically prepared to undertake malicious acts. The contrast of trustworthiness between a personal PalmPilot and a random PC would be even more pronounced if the Pilot employed security enhancements such as those discussed in the previous section.

In this section we are going to explore the possibilities of the following scenario: A trusted, small, moderately powerful computer collaborating ad-hoc with a more powerful, big, but untrusted, PC to achieve some functionality. The trusted small device is needed to assure a certain amount of security, and the big PC is needed to give a convenient and powerful interface for security-insensitive operations. For example, the PC could display media-rich documents that are not sensitive. Sensitive resources, however, need to be protected on the small device as it is collaborating with the untrusted PC.

This is really just a generalization of the case discussed so far in this chapter.

As a first example, let us get back to e-mail. Our implementation presented in Section 4.2.3 has at least two shortcomings:

1. The user does not see, on the Pilot, the message that is to be digitally signed[9]. This means that a malicious version of Communicator could forge e-mail.

2. Only for encrypted messages can the user decide whether or not they should be displayed on the PC.

The reason for these shortcomings is that the ultimate control over the application lies with Netscape Communicator, which runs on the untrusted PC. Our PilotKey implementation

[9]Communicator never passes the message content into the cryptoki library. Rather, it calculates the hash itself and then just lets the cryptoki library sign the hash of the message.

was merely a plug-in into the Communicator system and was called only whenever Communicator felt inclined to do so.

A better approach would be if the e-mail application were controlled from the Pilot. The Pilot establishes an encrypted connection – through the PC it is connected to – to the user's mail host. Then the user decides where a certain message should be displayed.

Another possible application is electronic commerce. Here, also, smart cards are often used. However, because of the lack of an interface, we do not really know what our smart card is doing and how much money it is spending. With a Pilot, the picture looks different: We again consider a scenario where three players are involved. First, there is a server offering goods to purchase. In addition, there is a trusted Pilot (or similar device) which is connected to an untrusted PC. The Pilot carries our electronic cash. On the PC, we launch the client application that connects to the server and displays the offered goods. The Pilot also authenticates itself to the server and establishes an encrypted channel that the PC cannot decrypt.

The client application can be used to view the items, get price information, initiate a purchase, *etc*. Before the Pilot spends any money, it displays relevant information such as the price and a description of the product on its screen. Only if and when the user confirms this *on the Pilot* can the transaction proceed. We note that even if the server and PC collaborate, the Pilot, which acts as an electronic "wallet," will not dispense more money than the user acknowledges.

Generalizing from these examples, we will now present a new programming model we call *Splitting Trust*. In this model, applications are split to run on different devices. Part of the application runs on a small, trusted device, and part of the application runs on a bigger, more powerful, but untrusted, device.

Using applications written in a splitting-trust fashion, users get both security and computing power. They get security because a crucial part of their application runs on a trusted

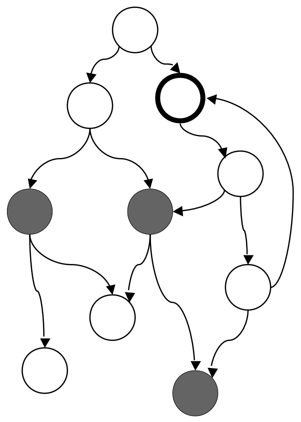

Figure 4.5: A Java application as a graph. Circles are objects, arrows are references, dark circles are sensitive. The bold object is to be moved to an untrusted platform.

device. They get computing power because the more powerful device can be used to run non-crucial parts of the application.

We have implemented a splitting-trust framework. The framework consists of a Java class library and a set of rules about how to write splitting-trust applications. The result of using the splitting-trust framework is an application that runs on the hand-held device, but parts of which can be "out-sourced" to the untrusted PC. Access to sensitive resources on the hand-held device is controlled.

4.3.1 Writing Splitting-Trust Applications

The splitting-trust framework for Java (called SJava) is a collection of classes and interfaces, as well as a set of rules that authors of splitting-trust applications have to follow.

A running Java application is a collection of objects. These objects are instances of Java classes. They hold references to each other, and they call methods upon each other. The basic idea behind SJava is that some of the objects in a running Java application are sensitive, while others are not. Imagine for example an e-mail application that we want to run on a hand-held computer and on an untrusted PC at the same time. The objects in the e-mail application that contain private key material are sensitive, while the objects that fetch (encrypted) data from the Internet are not.

Figure 4.5 shows a Java application as a directed graph. The circles are objects, and the arrows are references those objects hold. Some of the objects are sensitive (depicted in darker color), while most objects are not. The object with the bold circumference is to be moved to an untrusted platform. For example, that object might represent some sort of graphical user interface that we would rather see on a big PC, because reading and composing of (insensitive) messages is more convenient there.

Moving an object from one host to another is easy in Java. The Remote Method Invocation (RMI) system gives us the functionality we need. We can send objects (as arguments of method calls) to another host. The Java RMI system will send all the instance variables contained in an object across to the receiving side. If some of the instance variables happen to be object references, RMI will recursively apply the same process to the referenced objects, while at the same time taking care of tricky details such as cycles in the reference graph or two instance variables pointing to the same object. This process is known as "pickling." RMI will even dynamically download code that is needed to execute objects in their new environment. However, the pickling algorithm that RMI uses is not suitable. When RMI copies an object to another host, it will also copy all objects that are referenced from within that object, and all objects that are referenced from those, and so forth, regardless of their sensitivity. For SJava, we need a system that takes into account whether or not an object bears sensitive information, and only copies it to an untrusted host if it is not sensitive.

Figure 4.6 shows this approach. It shows an SJava application after it has been split in two. We notice that sensitive objects remain on the trusted device, even though they are referenced from objects that have been copied over to the untrusted PC.

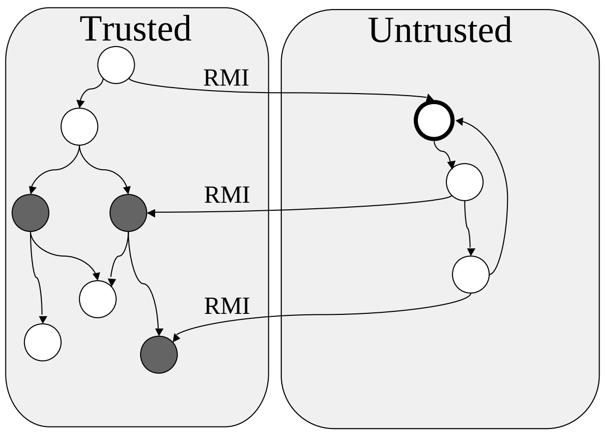

Figure 4.6: Java application after a split. Parts of the application have moved to an untrusted platform, and references have been replaced by RMI connections.

The way to write splitting-trust application, therefore, presents itself as follows to a developer:

1. Design the application such that sensitive data is encapsulated in objects that can remain on the trusted device. For example, do not plan for a user interface object that receives a sensitive password to be placed on the same window as a non-sensitive list of, say, to-do items.

2. Provide a means by which objects can specify, at run time, whether they are sensitive or not. In SJava, this means that potentially sensitive objects need to implement a certain interface. See Section 4.3.2 for details.

3. Use the SJava class library to implement the splitting-trust functionality. The SJava class library provides ways to move parts of an application onto a different host, while taking sensitivity of objects into account, and leaving sensitive objects on the local host.

4.3.2 The SJava Class Library

In this section we present the classes, and explain the rules application developers have to follow in more detail. The first part of the SJava class library are classes that allow seamless export of objects to a remote host. SJava provides the following API to developers for that purpose:

```
public class SJava {
    ...
    public static SplitServer getSplitServer(String hostName);
    ...
}

public interface SplitServer extends Remote {
    public Remote copyObject(Remote o);
}
```

`SJava.getSplitServer` can be used to connect to an untrusted host. We require that a small server, also provided by SJava, run on that host. The method returns a reference to that server.

`SplitServer.copyObject` can be used to copy an object to a remote host, returning a remote reference to the object just copied. The following shows how this API could be used (omitting the necessary exception handling):

```
public interface MyInterface extends java.rmi.Remote {
    public void doSomething();
}

public class MyInterfaceImpl implements MyInterface {
    public void doSomething() {
        ...
    }
}

// here starts the main program
...
MyInterface foo = new MyInterfaceImpl();
SplitServer otherHost = SJava.getSplitServer("otherhost.com");
MyInterface copyOfFoo = (MyInterface)otherHost.copyObject(foo);

foo.doSomething();       // do something here
copyOfFoo.doSomething(); // do it over on otherhost
...
```

Notice that when `foo` is copied over to the other host, the object is actually pickled up and *copied*. However, the "copy" that is returned back to the caller is a remote *reference* to the copied object, which now lives on the remote host.

It is worth noting how this functionality is achieved. When Java's Remote Method Invocation (RMI) subsystem copies an object to a remote host (for example, when that object is passed as an argument in a method call), it checks whether the object has been *exported*, *i.e.* whether there is a proxy object on the local host, listening on a network connection and ready to forward method calls to the exported local object. If that in fact is the case, then the RMI subsystem will send a *stub object* to the remote host, which knows how to connect back to the proxy on the local host. Hence, the remote host receives only a reference to the local object, instead of a copy.

If, however, the local object has not been exported, then there is no local proxy running. Therefore, the RMI subsystem actually pickles up the object and sends it across to the remote host, which unpickles it and thus receives a copy of the object.

`SplitServer.copyObject(Remote o)` takes as argument an object that must not be exported. Therefore, the RMI subsystem will send an actual copy of it to the remote implementation of `SplitServer`. The server on the remote host will then export the copy it received, and simply return that copy back to the caller. However, since now that copy is exported, the RMI subsystem will send a stub object back to the original caller, who ends up with a reference to a remote copy of the object that it originally passed to `copyObject`.

This functionality alone can give developers a primitive API to develop splitting-trust applications: Java's `transient` keyword can be used to prevent certain class members from being copied to a remote host during an RMI call:

```
public class ExampleClass {
   // when an instance of Example class is copied to
   // a remote host, this integer will be included in
   // the copy:
   int anInt;

   // this object will also be copied along:
   Object anObject;

   // this Object will not be copied:
   transient Object secretObject;
}
```

In the case of private keys and the like this may be sufficient. Often, however, different instances of the same class must be treated differently:

```
public class EmailMessage {
   // sometimes, this string holds sensitive information,
   // sometimes it does not.
   String contents;
}
```

In the above example, sometimes an instance of class `EmailMessage` could be sensitive, and sometimes it might not be. Java has no feature in the language to instruct the RMI subsystem of the different requirements for different instances of the same class.

For this purpose, we introduced the second part of the SJava class library. To developers, this part merely consists of an interface that they can choose to implement for some of their classes. Hidden to the developer are changes to the RMI subsystem, which make sure that the sensitivity of objects is taken into account.

The interface is defined as follows:

```
public interface Transient extends Serializable, Remote {
    public final static int COPYABLE  = 0;
    public final static int CALLABLE  = 1;
    public final static int SENSITIVE = 2;

    public int getSensitivityLevel() throws RemoteException ;
    public Object getUnclassified() throws RemoteException ;
}
```

Developers can implement this interface to enable objects to designate their *level of sensitivity*. When an object is about to be copied to an untrusted host, the (modified) RMI subsystem will call into the object to find out its sensitivity level. The object may declare one out of three different sensitivity levels:

- **COPYABLE** An object that declares itself to be copyable is not sensitive at all. It can be copied to any untrusted host. If that host has a modified Java Virtual Machine, or uses native code to introspect the object, it can learn the internal state of the object, including state held in private class members.

- **CALLABLE** An object that declares itself to be callable cannot be copied to an untrusted host. A remote reference to this object, however, *will* be made available to the remote host. That means that the untrusted party can call methods on this object, and that way infer information about its internal state. It cannot, however, inspect the object directly.

- **SENSITIVE** An object that declares itself to be sensitive will not be copied to an untrusted host, nor will there be a reference available to call methods on this object. Rather, the RMI subsystem will call the `getUnclassified` method on this object, requiring the object to return a sanitized version of itself, which presumably has all the sensitive state erased. For example, an `EmailMessage` object that holds the contents of a sensitive message in a member variable could return another instance of class `EmailMessage`, with the contents member containing an empty string instead of a highly sensitive message.

The third, and final, part of the SJava class library consists of various changes to the standard Java system classes (the JDK class library) in order to facilitate splitting-trust functionality. In particular, many classes that are not declared `Serializable` in the standard Java class library are declared `Serializable` in SJava so that objects of these classes can be copied to other hosts. Also, SJava's RMI subsystem is modified to take into account the special treatment of objects that implement the `Transient` interface.

In the next section, we will look at an example of how to use the SJava class library and framework.

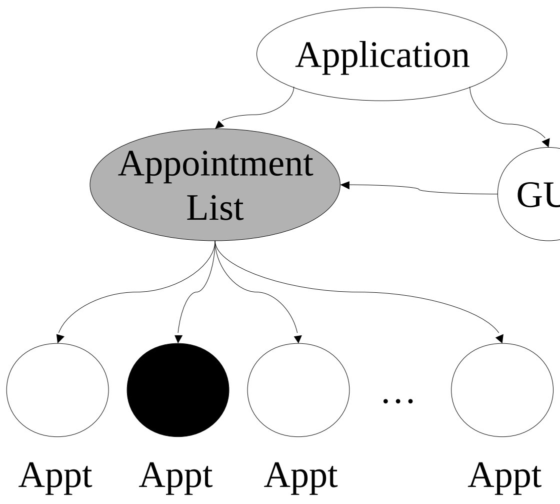

Figure 4.7: Sample calendar application. As in Figure 4.5, circles denote objects, and arrows denote references. White objects are COPYABLE, grey objects are CALLABLE, and black objects are SENSITIVE.

4.3.3 An SJava Example

Imagine a calendar application for CIA employees. Some of the appointments in their calendars will be sensitive (*i.e.*, outsiders should not be able to know the nature of the appointment), while other appointments are not (*e.g.*, it does not matter whether outsiders learn about hairdresser appointments). Imagine further that this calendar application runs on a small hand-held computer. Because it is quite tedious to view, and especially enter, appointments on the hand-held computer, it would be nice if we could connect the hand-held computer to a bigger PC and view/manipulate at least the non-sensitive appointments.

In this section, we show how such an application can be written in SJava, demonstrating how easy it is to develop a security-sensitive distributed application in SJava. At the same time, we will see some potential pitfalls that the programmer has to be aware of, even when using SJava.

As a user interface, we choose a window that displays one day's worth of appointments, together with a calendar view that lets users easily switch to a different day. To enter a new

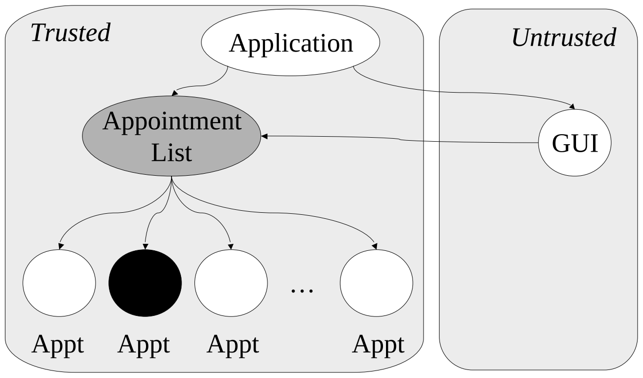

Figure 4.8: Moving the GUI to the untrusted platform is not practical.

appointment, users can – upon pressing a button – launch a dialog that will let them enter details about the appointment, including whether or not that appointment is to be considered secret.

There is a main application object (which has methods to display or hide the user interface), and there is a list object that holds all the appointments.

Figure 4.7 shows the design. The main application object holds references to the appointment list and the user interface objects. The user interface object holds a reference to the appointment list. Finally, the appointment list hold references to the various appointments. We label the main application object and the GUI object to be copyable. The appointment list is callable, and the individual appointment objects are either copyable or sensitive, depending on whether the appointments they represent are secret.

Our goal is to be able to move the user interface from the hand-held computer (where the application will be started) to an untrusted PC while making sure that the untrusted PC will not learn about secret appointments. One way to do that would be use the `SplitServer.copyObject()` method to copy the user interface to the other host (see Figure 4.8). This, however, turns out to be not practical. The internal structure of the GUI object is very complex, and consists of hundreds of objects. Some of these objects are not serializable, and therefore cannot be copied to the untrusted PC. Even if we change the standard Java

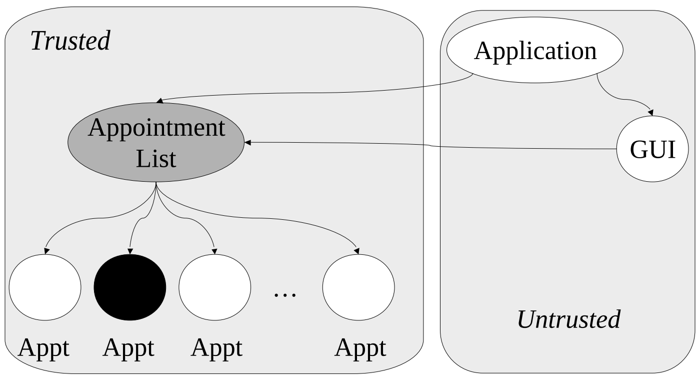

Figure 4.9: Design of the calendar application. Instead of moving the user interface object, we move the main application object.

class libraries to make all the necessary classes serializable, the amount of state contained in these objects is considerable. Copying all the state from the hand-held computer to the untrusted PC over a slow (serial) link, using a slow protocol (RMI) turns out to take too long.

Therefore, we choose a different approach. Instead of the user interface object, we copy the application object across, and equip it with methods to destroy and recreate (on the other side) the user interface component. Figure 4.9 demonstrates this approach.

Because the main application object needs to be copied to the untrusted PC, we need to declare both an interface and an implementing class for it. The interface needs to inherit from `Remote` (because we want to call the application remotely), and `Serializable` (because we want to copy an application object across a network):

```
public interface CalendarAppInterface extends Remote, Serializable {
    public void showFrame();
    public void removeFrame();
    ...
}
```

The methods of the `CalendarAppInterface` are implemented as follows:

```
public class CalendarApp implements CalendarAppInterface {
    private MainFrame mainFrame;
    private AppointmentCollection ac;
    ...

    public void showFrame() {
        mainFrame = new MainFrame(ac,this);
        mainFrame.setVisible(true);
    }

    public void removeFrame() {
        mainFrame.dispose();
        mainFrame = null;
    }
    ...
}
```

Before we copy an instance of class `CalendarApp` to the untrusted PC, we call `removeFrame` on it. When the object is now copied, the `mainFrame` member variable is null, and does not add much in terms of overhead to the copying process. Then, the hand-held PC calls `showFrame` on the newly-arrived instance on the untrusted PC. As we can see, the user interface is re-constructed using a reference of the appointment collection. Since the appointment collection is labeled CALLABLE (see Figure 4.7), the member variable `ac` in the application instance on the untrusted PC is actually a remote reference to the appointment collection object that remained on the hand-held computer.

Labeling the appointment collection as CALLABLE is done as follows. First, we need to define an interface `AppointmentCollection` that the GUI components can use. It needs to inherit from `Transient`:

```
public interface AppointmentCollection extends Transient {
    void addAppointment(calendar.Appointment p0);
    Appointment getAppointmentFor(java.util.Date p0);
}
```

Then, we need to implement the interface, including the methods defined in `Transient`:

```
public class AppointmentCollectionImpl
                  implements AppointmentCollection {

  private ArrayList apps = new ArrayList();

  public void addAppointment(Appointment a) {
    apps.add(a);
  }

  public Appointment getAppointmentFor(Date d) {
    // for each appointment a in apps
    // if a is scheduled during d then
    return a;
  }

  public int getSensitivity() {
    return Transient.CALLABLE;
  }

  public Object getUnclassified() {
    return null;
  }
}
```

The appointments themselves are also defined as an interface inherited from `Transient`, and have methods such as `setStartTime`, `getStartTime`, `setDescription`, *etc.* An appointment instance is either COPYABLE or SENSITIVE, depending on whether the user labeled the appointment “secret.”

When the user interface component on the untrusted PC tries to display a certain day’s worth of appointments, it will call into the appointment collection (across to the hand-held PC - see Figure 4.9), which will just return appointment objects to its caller. It can be ignorant of the fact that the caller may actually be on the untrusted side. When an appointment object is returned to the user interface object across the network, and if that appointment object is labeled SENSITIVE, then the SJava RMI subsystem will actually return a different instance to the caller - an object that was retrieved by calling `getUnclassified` on the sensitive object. The appointment instance that will be returned to the untrusted PC may have a bogus start and end time, and a bogus appointment description (in our implementation, it has a bogus description, but keeps the start and end time).

To summarize, the calendar application can easily be written using SJava. It is possible to move the user interface component quickly to the untrusted PC, transparently setting up RMI connections to objects where local references used to be. Sensitive objects are never delivered to the untrusted side.

We have also seen that the splitting trust framework is not entirely transparent to the developers. Often, they need to define interfaces where normally classes would have sufficed.

They need to generate stubs for objects that need to be accessed across the network, and they need to circumvent certain obstacles such as the impracticality of copying an entire user interface over a slow serial link.

These issues can be addressed, and some of them should be. For example, a slow serial connection is not likely to remain a problem in the future, and just-in-time stub generators are certainly easy to implement. One could also imagine that the whole Java class library were designed in a way that would make distribution completely transparent to the programmer. If, for example, all classes would have to be split into an interface (which clients bind to at link time) and an implementation (which can be replaced at runtime), object distribution could be made much more transparent. However, the Java designers have consciously chosen not to go down that path, and to give the programmer the choice between performance and distributability (at the expense of convenience). We believe that a similar tradeoff is necessary for splitting trust applications, and therefore do not anticipate that splitting trust will become, or should be, completely transparent to the programmer.

4.3.4 Security Considerations of SJava

The primary security concern for SJava is whether it duly protects access to sensitive resources. In terms of access control policies, the policy the system needs to enforce is very simple: Objects marked as CALLABLE or SENSITIVE should not be accessible from the untrusted (remote) side.

It turns out that the built-in Java RMI mechanism has a mode of operation where a remote object reference is expressed as a pair of numbers. One is a TCP port number, the other is a long, pseudo-randomly generated object identifier. In order to call methods from the outside on an object, one has to connect to the correct TCP port, and then present the appropriate object identifier. Since that identifier is hard to guess, it is hard for an outsider to call into an object unless they know the object identifier. Therefore, remote object references act like capabilities — the hand-held computer needs to give them to the untrusted PC before it can make calls on any objects that reside on the hand-held computer.

Having said that, we need to point out that using SJava does not guarantee a "secure" application. It is entirely possible that the developer marks objects as COPYABLE that should have been sensitive, or that some classes that should implement the `Transient` interface do not. Also, overall application design can hurt, or benefit, the security of the application. As mentioned above, an example are password dialogs that should never appear on the untrusted PC.

A third area of security concerns are domain-specific. Again, SJava is merely a tool for writing splitting-trust applications, not a guarantee that these application will be "secure," for all domain-specific definitions of the word "secure." For example, as we mentioned above, in our implementation of the calendar application, the untrusted PC actually learns the start and end time of secret appointments (the description of the appointments is blinded, though). Arguably, that is more than the untrusted side should know. But even if the start and end times would have been hidden from the untrusted PC, should we hide the fact that at a certain time there is already an appointment if we try to schedule a second one? Also, is the fact that my appointment collection holds five appointments for April 2nd information that is sensitive or not? SJava cannot help making these information-flow decisions. When an object is considered sensitive is entirely up to the developer of the application.

SJava provides only low-level security. It guarantees that it is hard for an intruder to call into objects that have not been made available to the untrusted side. Beyond that, it is a handy tool to write splitting-trust applications. It cannot guarantee that these applications will be "secure," in a sense that is above and beyond the scope of SJava.

4.4 Conclusions

In this chapter we have first argued that small hand-held computers can be used instead of smart cards. Moreover, they provide a direct interface to the user, which entails some desirable security properties. We implemented a PKCS#11 library for Netscape Communicator and corresponding smart card functionality for a 3COM PalmPilot.

In our implementation, the PalmPilot provides a trusted authentication path and gives the user a choice where an encrypted e-mail message should be displayed: in Communicator on the PC or on the PalmPilot itself. This increases the user's privacy above the level provided by traditional smart cards.

We have also introduced, and presented an implementation of, a new programming paradigm, which we call "splitting trust." Under this new paradigm, applications are split into two parts: One part runs on a trusted, but small and only moderately powerful, device; the other part runs on a bigger, more powerful, but possibly untrusted, device like a PC in a public place. Splitting applications gives us both certain security assurances on one hand, and convenience and speed on the other hand.

The framework we provide assists developers in the process of writing split applications. While the framework was developed with hand-held computers, and untrusted and more powerful PCs in mind, it can be applied to a variety of scenarios.

4.5 Related Work

Yee introduces the notion of a *secure coprocessor* [51]. A secure coprocessor is a tamper-resistant module that is part of an otherwise not necessarily trusted PC. Certain goals (notably copy protection) can be achieved by *splitting* applications into a *critical* and *uncritical* part; the critical part runs on the secure coprocessor. While this idea is very similar to ours, the context is different: In the world of secure coprocessors the user is not necessarily trusted (the coprocessor secures information from, among others, the user using it). On the other hand, the user always trusts the secure coprocessor, even if it is part of an otherwise unknown PC. In our world, the user/owner of the PalmPilot is trusted by definition (the whole point of our design is to protect the user). The PC, or any of its parts (even if they look like secure coprocessors) is never trusted.

Gobioff et al. notice that smart cards lack certain security properties due to their lack of user I/O [24]. They propose that smart cards be equipped with “additional I/O channels” such as LEDs or buttons to alleviate these shortcomings. Our design meets their vision, but we come from the opposite direction: We take a hand-held computer that already has user I/O and implement smart card functionality on it.

Boneh et al. implemented a electronic cash wallet on a PalmPilot, which is quite similar to what we describe in the introduction to Section 4.3 [14].

Cryptographers have dealt with the “splitting trust” scenario for some time now, even though the work is often not presented from that perspective. For example, Blaze et. al [10, 9] want to use a powerful PC in conjunction with a smart card for symmetric key encryption because the PC provides higher encryption bandwidth. However, the PC is not trusted to learn the secret key. “Function hiding” work [40] is usually presented from a persective of copyright protection, but essentially it is also an instance of splitting trust. Boneh et al. speed up RSA key pair generation on the PalmPilot with the help of an untrusted server, which participates in the key pair generation, yet will not learn the private key [12]. The SJava class library and framework present an ideal platform to implement these kinds of applications.

The mobile agent community deals with the problem of how programs can efficiently and reliably be sent to remote hosts [49]. While SJava cannot guarantee the integrity of agents on remote hosts, it can guarantee the integrity of the part of the application that remains in the trusted environment. And although SJava is an implementation of the splitting trust idea in Java, it is entirely conceivable that other remote programming frameworks [31, 48] could provide similar splitting trust implementations. Mobile agent frameworks usually provide a high degree of transparency to the programmer in terms of where programs can actually execute. With a splitting trust implementation, programmers can retain this transparency, and at the same time express some of the security requirements of their applications.

Chapter 5

Conclusions

In this dissertation, we have presented a number of approaches to different aspects of the problem of access control for ad-hoc collaboration. While seemingly contradictory at first (the need to protect resources from access of unauthorized entities seems to be at odds with the welcoming philosophy of ad-hoc collaboration), we have presented examples in which access control and ad-hoc collaboration can coexist.

Even more, sometimes one cannot exist without the other. Consider for example the future of online commerce. The very nature of the Internet makes it possible for anyone to become a merchant or a customer of electronic, or traditional, goods. Without the knowledge that files and other resources on their personal computers are protected while they explore new sites on the Internet, customers will be hesitant to adopt this way of interacting in general, and shopping in particular.

On the other hand, if all we ever did was to connect to known and trusted sites, privacy protection and access control would be less of an issue for Internet users. The need to protect ourselves from unpleasant results of those interactions arises because we like to explore and interact with unknown sites.

There is no “one-size-fits-all” solution for access control for ad-hoc collaboration. However, the mechanisms presented in this dissertation can be used to address certain issues that often arise in different contexts.

In a corporate setting, for example, it becomes increasingly impractical to rely on centrally managed access control for in-house servers. Corporations are more in flux than they have ever been - mergers and acquisitions are more frequent, and employee turnover is higher. Therefore, even inside a corporation, access to an in-house server is increasingly of an ad-hoc nature, since the clients accessing the server may not be known beforehand. A decentralized trust-management system can address this particular aspect of access control in large corporations. When a division leaves or joins the corporation, the administrative overhead should be minimal.

In this dissertation, we presented a system that allows a great deal of flexibility in delegating access control decisions, while at the same time permitting abstract and formal reasoning about the properties of the system in general, and access control policies in particular. With this system, servers can authenticate clients with which they have never interacted before, and can check whether those clients have the privileges needed to perform certain actions.

In Chapter 3 we described how the concept of complete separation (which, by itself, is not new) can be used to protect resources in the presence of ad-hoc collaboration. Many practical issues needed to be addressed, especially that in the real world *complete* separation is often not desirable. However, the fact that only a sealed-off compartment of a given computer should engage in ad-hoc collaboration is a powerful concept. It removes the need for any particular access control policy. Instead, inside this ad-hoc collaboration compartment, “anything goes,” except for interactions with other compartments on the user’s computer. This lack of complicated security mechanisms makes it a very simple system for people to understand. Access control is achieved because we forbid access from the ad-hoc collaboration compartment to other compartments inside the user's computer.

This technique could be used in other contexts as well. For example, we could put a compartmented operating system on a small hand-held computer that we use for authentication purposes. Depending on who we authenticate ourselves to, we could instruct the hand-held computer to use different compartments, having their own sets of cryptographic keys, applications, *etc.* This allows for ad-hoc collaboration since we now need to be less concerned about who we talk to and what they might learn about us, because the compartmented hand-held computer in effect provides us with a variety of different personae.

The splitting-trust framework introduced in Chapter 4 is another example for a technique that allows ad-hoc collaboration while at the same time protecting sensitive resources. While introduced in the context of trusted hand-held computers and untrusted PC's, the concept can be applied in different settings as well. As mentioned earlier, one could imagine a splitting trust setup for the distribution of active content on the Internet. The part of the application that is sensitive can run at the content provider's site, while other, non-sensitive parts of the application can run closer to the consumer of the content, enhancing the browsing experience of the end user.

At the same time, the work presented in this dissertation opens the door for more research. For example, the question of how clients in Chapter 2 pick the right credentials to present to the servers is still open. If the clients present all credentials that they have (as it is the case in our implementation) they may reveal more about themselves than they might want to (performance considerations aside). If they present too few credentials, they may not gain access to a resources that they should have gained access to. If the server helps the client to select credentials by publishing its security policy, *it* might reveal more to the clients than it should.

Likewise, we still do not know how to construct an access control logic that will not be "surprising." While we have proven certain formal properties of the logic, we do not know whether it conforms with our intuition of what it means to be secure, precisely because that intuition is not formal.

The WindowBox system presents an interesting prototype for the complete-separation approach to access control for ad-hoc collaboration. It remains to be investigated whether users find this approach really more intuitive than, say, putting up firewalls or specifying security policies for mobile code.

We also need to find a better understanding for how users can interact with a system in a way that cannot be fooled by malicious software [22]. How can a user really be sure that he or she is currently in the ad-hoc collaboration compartment, and not in some other compartment that is just made to look the same by some piece of malicious software? What is probably needed is a rethinking of the user-computer-interaction model, taking into account the fact that I, as a user, would like to interact with trusted software on my computer while I know that at the same time there is also untrusted software running on the computer.

Lastly, while the splitting-trust framework presented in Chapter 4 is a great tool to write applications that can run parts of themselves on untrusted platforms, it does not help programmers yet in making those applications truly secure. The application programmer has to figure out how the split should happen, and the splitting-trust framework will allow him or her to accomplish the split. It does not, however, guide the programmer as to how the split should be accomplished, let alone automatically split the application into a security-sensitive and security-insensitive part. While this is conceivable, we again struggle with the intuitive notion of security that the automated system would have to capture.

Summarizing, this dissertation gives some answers to what we believe will be a pervasive problem in the years to come: the fact that increasingly online contacts will be impulsive, non-repeating and, hence, ad-hoc. At the same time, people will become increasingly aware of, and concerned about, the security risks associated with a universally connected society in general, and with ad-hoc collaboration in particular. The goal of this work has been to lead the way into addressing some of the security problems that we will have to face in the near future.

Appendix A

Proofs of Inference Rules

In this appendix, we will prove the six inference rules from Section 2.3.2 as theorems in Church's high order logic.

Lemma 1. $\forall X_{\mathsf{Perm}}. \quad \mathrm{Self} : X \quad \Longleftrightarrow \quad X$

Proof.

$$\begin{aligned}
\mathrm{Self} : X &\Longleftrightarrow \exists G.\ \mathrm{Self}(G) \wedge (G \supset X) && \text{by (2.20)} \\
&\Longleftrightarrow \exists G.\ G \wedge (G \supset X) && \text{by (2.25)} \\
&\Longleftrightarrow \exists G.\ G \wedge (\neg G \vee X) \\
&\Longleftrightarrow \exists G.\ G \wedge X \\
&\Longleftrightarrow X
\end{aligned}$$

□

Lemma 2.

$$\forall A_{\mathsf{Prin}}, \forall F_{\mathsf{Perm}}, G_{\mathsf{Perm}}. \qquad A : F \quad \wedge \quad F \supset G \qquad \supset \qquad A : G$$

Proof.

$$\begin{aligned}
A : F \quad \wedge \quad F \supset G \quad &\Longleftrightarrow \quad \exists H_{\mathsf{Perm}}. \quad A(H) \wedge H \supset F \quad \wedge \quad F \supset G \\
&\supset \quad \exists H_{\mathsf{Perm}}. \quad A(H) \wedge H \supset G \\
&\Longleftrightarrow \quad A : G
\end{aligned}$$

□

Lemma 3.

$$\begin{aligned}
&\forall X_{\mathsf{Prin}}, \forall P_{\mathsf{Perm}}, P'_{\mathsf{Perm}}. \\
&\qquad \text{Self} : \text{Delegate}(X, P) \\
&\qquad \supset \quad X : P' \\
&\qquad \supset \quad P \vee P'
\end{aligned}$$

Proof. Assuming Delegate(X,P), we get

$$\forall Q_{\mathsf{Perm}}. \quad (P \supset Q) \wedge X : Q \quad \supset \quad Q \qquad\qquad \text{by (2.22)}$$

Since this holds for all Q, we get in particular

$$(P \supset (P \vee P')) \wedge X : (P \vee P') \quad \supset \quad (P \vee P')$$

Since P indeed implies $(P \vee P')$, we get

$$X : (P \vee P') \quad \supset \quad (P \vee P')$$

We know that

$$X : P' \quad \supset \quad X : (P \vee P') \qquad\qquad \text{by Lemma 2}$$

because $P' \supset (P \vee P')$. Therefore, we have

$$X : P' \quad \supset \quad X : (P \vee P') \quad \supset \quad (P \vee P')$$

which proves the lemma. □

Theorem 1. ***(Delegation)***.

$$\begin{aligned} &\forall X_{\mathsf{Prin}}, \forall P_{\mathsf{Perm}}, P'_{\mathsf{Perm}}. \\ &\qquad \text{Self} : \text{Delegate}(X, P) \\ &\qquad \supset \quad X : P' \\ &\qquad \supset \quad P \supset P' \\ &\qquad \supset \quad \text{Self} : P' \end{aligned}$$

Proof. This follows immediately from Lemma 3, because $((P \vee P') \wedge (P \supset P')) \supset P'$. □

Theorem 2. ***(Transitivity)***.

$$\begin{aligned} &\forall X_{\mathsf{Prin}}, Y_{\mathsf{Prin}}, Z_{\mathsf{Prin}}. \\ &\qquad \text{Self} : \text{Bind}(X, Y) \\ &\qquad \supset \quad \text{Self} : \text{Bind}(Y, Z) \\ &\qquad \supset \quad \text{Self} : \text{Bind}(X, Z) \end{aligned}$$

Proof. We assume $\mathrm{Bind}(X,Y)$ and $\mathrm{Bind}(Y,Z)$. Hence we have

$$\forall P_{\mathsf{Perm}}. \quad X : P \supset Y : P \qquad \text{by (2.21)}$$

and

$$\forall Q_{\mathsf{Perm}}. \quad Y : Q \supset Z : Q. \qquad \text{by (2.21)}$$

The latter implies $Y : P \supset Z : P$. Hence, we have

$$\forall P_{\mathsf{Perm}}. \quad (X : P \supset Y : P) \wedge (Y : P \supset Z : P).$$

Therefore,

$$\forall P_{\mathsf{Perm}}. \quad X : P \supset Z : P$$

which is equivalent to $\mathrm{Bind}(X,Z)$ and therefore proves the theorem. □

Theorem 3. ***(Containment)***.

$$\begin{aligned} &\forall X_{\mathsf{Prin}}, Y_{\mathsf{Prin}}, \forall P_{\mathsf{Perm}}. \\ &\qquad\qquad \mathrm{Self} : \mathrm{Bind}(X,Y) \\ &\qquad \supset \quad \mathrm{Self} : \mathrm{Delegate}(Y,P) \\ &\qquad \supset \quad \mathrm{Self} : \mathrm{Delegate}(X,P) \end{aligned}$$

Proof. We assume $\mathrm{Bind}(X,Y)$ and $\mathrm{Delegate}(Y,P)$. That yields

$$\forall Q_{\mathsf{Perm}}. \quad X : Q \supset Y : Q \qquad \text{by (2.21)} \tag{A.1}$$

and

$$\forall R_{\mathsf{Perm}}.\quad (P \supset R \quad\wedge\quad Y : R) \quad\supset\quad R \qquad\qquad \text{by (2.22)} \qquad\qquad \text{(A.2)}$$

Now if we assume that $\forall S_{\mathsf{Perm}}.\, P \supset S \wedge X : S$ we get

$$\forall S_{\mathsf{Perm}}.\quad P \supset S \wedge Y : S \qquad\qquad \text{by (A.1)}$$

and, finally, S (by (A.2)). In summary, we have

$$\forall S_{\mathsf{Perm}}.\quad (P \supset S \wedge X : S) \supset S$$

which proves the theorem since this is equivalent to $\text{Delegate}(X, P)$. □

Theorem 4. ***(AnyPrincipal)***.

$$\forall X_{\mathsf{Prin}}.\qquad \text{Self} : \text{Bind}(X, \text{AnyPrin})$$

Proof.

$$\begin{array}{lllll}
 & \forall P_{\mathsf{Perm}}.\quad X : P & \supset & \mathsf{true} & \\
\supset & \forall P_{\mathsf{Perm}}.\quad X : P & \supset & (\mathsf{false} \supset P) & \\
\supset & \forall P_{\mathsf{Perm}}.\quad X : P & \supset & (\exists G_{\mathsf{Perm}}.\quad G \supset P) & \\
\supset & \forall P_{\mathsf{Perm}}.\quad X : P & \supset & (\exists G_{\mathsf{Perm}}.\quad \mathsf{true} \wedge G \supset P) & \\
\supset & \forall P_{\mathsf{Perm}}.\quad X : P & \supset & (\exists G_{\mathsf{Perm}}.\quad \text{AnyPrin}(G) \wedge G \supset P) & \text{by (2.24)} \\
\supset & \forall P_{\mathsf{Perm}}.\quad X : P & \supset & \text{AnyPrin} : P & \text{by (2.20)} \\
\supset & \text{Bind}(X, \text{AnyPrin}) & & &
\end{array}$$

$\square$

In the following, apos is the function defined in (2.29), while Apos is a variable of type $\mathsf{Prin} \rightarrow \mathsf{String} \rightarrow \mathsf{Prin}$.

Lemma 4.

$$\forall A_{\mathsf{Prin}}, \forall S_{\mathsf{String}}, \forall G_{\mathsf{Perm}}, \forall \mathrm{Apos}_{\mathsf{Prin} \rightarrow \mathsf{String} \rightarrow \mathsf{Prin}}.$$
$$\mathrm{apos\_like}(\mathrm{Apos}) \quad \supset \quad \big(\mathrm{apos}(A,S)(G) \iff \mathrm{Apos}(A,S) : G\big)$$

(See definitions (2.26) and (2.29)).

Proof.

1. $\mathrm{apos\_like}(\mathrm{Apos}) \supset \mathrm{apos}(A,S)(G) \supset \mathrm{Apos}(A,S) : G$

   We assume that we are given an Apos for which holds $\mathrm{apos\_like}(\mathrm{Apos})$, and also that we are given A, S, and G such that $\mathrm{apos}(A,S)(G)$.

   $$\begin{aligned}\mathrm{apos}(A,S)(G) &= \big(\lambda F. \forall \mathrm{Apos}'_{\mathsf{Prin} \rightarrow \mathsf{String} \rightarrow \mathsf{Prin}}\ (\mathrm{apos\_like}(\mathrm{Apos}') \supset \mathrm{Apos}'(A,S) : F)\big)(G) \\ &= \forall \mathrm{Apos}'_{\mathsf{Prin} \rightarrow \mathsf{String} \rightarrow \mathsf{Prin}}\ (\mathrm{apos\_like}(\mathrm{Apos}') \supset \mathrm{Apos}'(A,S) : G)\end{aligned}$$

   Since we know $\mathrm{apos\_like}(\mathrm{Apos})$ we get $\mathrm{Apos}(A,S) : G$.

2. $\mathrm{apos\_like}(\mathrm{Apos}) \supset \mathrm{Apos}(A,S) : G \supset \mathrm{apos}(A,S)(G)$

   follows directly from the definition of apos (2.29).

$\square$

Lemma 5.

$$\forall A_{\mathsf{Prin}}, B_{\mathsf{Prin}}, \forall S_{\mathsf{String}}, \forall \mathrm{Apos}_{\mathsf{Prin}\to\mathsf{String}\to\mathsf{Prin}}.$$
$$\mathrm{apos\_like}(\mathrm{Apos}) \supset \mathrm{Bind}(A,B)$$
$$\supset \mathrm{Bind}(\mathrm{Apos}(A,S), \mathrm{Apos}(B,S))$$

Proof. If we assume that apos_like(Apos) then we know that sat_mono(Apos) (see (2.26)), which by (2.27) gives us $\mathrm{Bind}(\mathrm{Apos}(A,S),\mathrm{Apos}(B,S))$. □

Theorem 5. ***(Monotonicity)***.

$$\forall A_{\mathsf{Prin}}, B_{\mathsf{Prin}}, \forall S_{\mathsf{String}}.$$
$$\mathrm{Self} : \mathrm{Bind}(A,B) \supset \mathrm{Self} : \mathrm{Bind}(\mathrm{apos}(A,S), \mathrm{apos}(B,S))$$

Proof. If we assume $\mathrm{Bind}(A,B)$ and $\mathrm{apos}(A,S) : F$ for some given F_{Perm}, then we need to prove $\mathrm{apos}(B,S) : F$ (see definition (2.21)). $\mathrm{apos}(A,S) : F$ yields

$$\exists G_{\mathsf{Perm}}. \qquad \mathrm{apos}(A,S)(G) \quad \wedge \quad G \supset F \qquad\qquad \text{by (2.20)}$$

If we can show that for that particular G, we have $\mathrm{apos}(B,S)(G)$ we are done, since then we get

$$\exists G_{\mathsf{Perm}}. \qquad \mathrm{apos}(B,S)(G) \quad \wedge \quad G \supset F$$

which is the same as $\mathrm{apos}(B,S) : F$.

Therefore, we are left to prove $\mathrm{apos}(B,S)(G)$. We know that $\mathrm{apos}(A,S)(G)$ is the same as

$$\forall \mathrm{Apos}_{\mathsf{Prin}\to\mathsf{String}\to\mathsf{Prin}}(\mathrm{apos\_like}(\mathrm{Apos}) \supset \mathrm{Apos}(A,S) : G) \qquad\qquad \text{by (2.29)}$$

On the other hand, because we know $\mathrm{Bind}(A,B)$, we get

$$\forall \mathrm{Apos}_{\mathsf{Prin}\to\mathsf{String}\to\mathsf{Prin}} \quad (\mathrm{apos\_like}(\mathrm{Apos}) \supset \mathrm{Bind}(\mathrm{Apos}(A,S),\mathrm{Apos}(B,S))) \qquad \text{by Lemma 5}$$

Therefore, we get

$$\forall \mathrm{Apos}_{\mathsf{Prin}\to\mathsf{String}\to\mathsf{Prin}}(\mathrm{apos\_like}(\mathrm{Apos}) \supset \mathrm{Apos}(B,S) : G) \qquad \text{by (2.21)}$$

which is the same as $\mathrm{apos}(B,S)(G)$. □

Lemma 6.

$$\forall A_{\mathsf{Prin}}, B_{\mathsf{Prin}}, \forall S_{\mathsf{String}}, \forall \mathrm{Apos}_{\mathsf{Prin}\to\mathsf{String}\to\mathsf{Prin}}. \quad \mathrm{apos\_like}(\mathrm{Apos}) \supset \mathrm{Bind}(B,\mathrm{apos}(A,S)) \supset \mathrm{Bind}(B,\mathrm{Apos}(A,S))$$

Proof. Let us assume we are given an Apos for which $\mathrm{apos\_like}(\mathrm{Apos})$ holds. We have

$$\begin{array}{rll}
& \mathrm{Bind}(B,\mathrm{apos}(A,S)) & \\
\Longleftrightarrow & \forall G_{\mathsf{Perm}}. \quad B : G \supset \mathrm{apos}(A,S) : G & \text{by (2.21)} \\
\Longleftrightarrow & \forall G_{\mathsf{Perm}}\, \exists F_{\mathsf{Perm}}. \quad B : G \supset (\mathrm{apos}(A,S)(F) \wedge F \supset G) & \text{by (2.20)} \\
\Longleftrightarrow & \forall G_{\mathsf{Perm}}\, \exists F_{\mathsf{Perm}}. \quad B : G \supset (\mathrm{Apos}(A,S) : F \wedge F \supset G) & \text{by Lemma 4} \\
\supset & \forall G_{\mathsf{Perm}}. \quad B : G \supset \mathrm{Apos}(A,S) : G & \text{by Lemma 2} \\
\Longleftrightarrow & \mathrm{Bind}(B,\mathrm{Apos}(A,S)) &
\end{array}$$

□

Theorem 6. ***(Namespace Ownership)***.

$$\forall X_{\mathsf{Prin}}, Y_{\mathsf{Prin}}, \forall B_{\mathsf{String}}.$$
$$\mathrm{X} : \mathrm{Bind}(Y, \mathrm{apos}(X,B)) \supset \mathrm{Self} : \mathrm{Bind}(Y, \mathrm{apos}(X,B))$$

Proof. If we assume $X : \mathrm{Bind}(Y, \mathrm{apos}(X,B))$ and $Y : F$ for some given F, then we need to prove $\mathrm{apos}(X,B) : F$ (see definition (2.21)). $Y : F$ yields

$$\exists G_{\mathsf{Perm}}. \quad Y(G) \quad \wedge \quad G \supset F \qquad \text{by (2.20)} \tag{A.3}$$

If we can show that for that particular G, we have $\mathrm{apos}(X,B)(G)$ we are done, since then we get

$$\exists G_{\mathsf{Perm}}. \quad \mathrm{apos}(X,B)(G) \quad \wedge \quad G \supset F$$

which is the same as $\mathrm{apos}(X,B) : F$.

Therefore, we are left to prove $\mathrm{apos}(X,B)(G)$. We know

$$X : \mathrm{Bind}(Y, \mathrm{apos}(X,B)).$$

By Lemma 6 and Lemma 2 it follows that

$$\forall \mathrm{Apos}_{\mathsf{Prin}\to\mathsf{String}\to\mathsf{Prin}}. \quad \mathrm{apos\_like}(\mathrm{Apos}) \supset X : \mathrm{Bind}(Y, \mathrm{Apos}(X,B))$$

This yields

$$\forall \mathrm{Apos}_{\mathsf{Prin}\to\mathsf{String}\to\mathsf{Prin}}. \quad \mathrm{apos\_like}(\mathrm{Apos}) \supset \mathrm{Bind}(Y, \mathrm{Apos}(X,B)) \quad \text{by (2.26) and (2.28)}$$

On the other hand, from (A.3) we know that $Y(G)$, which implies $Y : G$ by definition (2.20). So, by (2.21) it follows that

$$\forall \mathrm{Apos}_{\mathsf{Prin}\to\mathsf{String}\to\mathsf{Prin}}. \quad \mathrm{apos\_like}(\mathrm{Apos}) \supset \mathrm{Apos}(X,B) : G$$

which is the same as $\mathrm{apos}(X,B)(G)$. □

Theorem 7. ***(Implication)***.

$$\begin{aligned} &\forall X_{\mathsf{Prin}}, Y_{\mathsf{Prin}} \quad \forall P_{\mathsf{Perm}}, Q_{\mathsf{Perm}} \\ &\qquad\qquad \mathrm{Self} : \mathrm{Bind}(Y,X) \\ &\qquad \supset \quad P \supset Q \\ &\qquad \supset \quad \mathrm{Delegate}(X,P) \supset \mathrm{Delegate}(Y,Q) \end{aligned}$$

Proof. We assume that

$$\forall R_{\mathsf{Perm}}. \quad Y : R \supset X : R \qquad \text{by (2.21),} \tag{A.4}$$

$$P \supset Q \tag{A.5}$$

and

$$\forall R_{\mathsf{Perm}}. \quad ((P \supset R) \wedge X : R) \supset R \qquad \text{by (2.22).} \tag{A.6}$$

Now if we assume that

$$\forall S_{\mathsf{Perm}}. \quad Q \supset S \ \wedge \ Y : S$$

we get

$$\forall S_{\mathsf{Perm}}.\quad Q \supset S \land X : S \qquad \text{by (A.4)}$$

$$\forall S.\quad P \supset S \land X : S \qquad \text{by (A.5)}$$

and finally

$$S \qquad \text{by (A.6).}$$

which means that Delegate(Y, Q). □

Appendix B

Proof of Decidability

In this appendix, We will show that the logic presented in Chapter 2 is decidable. Let us call that logic $\mathfrak{L}$. We will proceed as follows: First, we will introduce a logic $\mathfrak{L}'$ that we will derive directly from $\mathfrak{L}$ by replacing certain inference rules. $\mathfrak{L}'$ will be equivalent to $\mathfrak{L}$ except for the fact that there are some statements provable in $\mathfrak{L}$ that are not provable in $\mathfrak{L}'$. We will characterize those statements.

We will then give a decision procedure for $\mathfrak{L}'$, thus proving that $\mathfrak{L}'$ is decidable. Finally, we will show a decision procedure for $\mathfrak{L}$ that is based on the decision procedure for $\mathfrak{L}'$, thus proving that $\mathfrak{L}$ is decidable.

B.1 Restricting the Logic

Let us call the logic presented in Chapter 2 $\mathfrak{L}$. In this section we define a logic $\mathfrak{L}'$ that has the following properties:

- It has the same syntax as $\mathfrak{L}$.
- It has inference rules (Del), (Trans), (Cont), (Own) and (Impl).
- It does not have inference rules (AnyPrin) and (Mon).

- It has a number of additional inference rules (see below).

While introducing the new inference rules, we will show that:

- given a set of axioms, every goal provable in $\mathfrak{L}'$ is provable in $\mathfrak{L}$, and
- given a set of axioms, every goal provable in $\mathfrak{L}$ that is not of the form $\mathrm{Self} : \mathrm{Bind}(X,Y)$ or $\mathrm{Self} : \mathrm{Delegate}(X,Y)$ is provable in $\mathfrak{L}'$.

In particular, this means that $\mathfrak{L}'$ is equivalent to $\mathfrak{L}$ as far as the Placeless Documents System is concerned. In Placeless, kernels never have to prove goals of the form $\mathrm{Self} : \mathrm{Bind}(X,Y)$ or $\mathrm{Self} : \mathrm{Delegate}(X,Y)$. Rather, goals to be proven are always statements about primitive permissions, such as Self : Read, *etc.*

The rule (AnyPrin) is missing from $\mathfrak{L}'$. In proofs, the conclusion of (AnyPrin) can serve as premise in either (Trans), (Impl), or (Cont). Let us call the composition of (AnyPrin) and (Trans) “AnyTrans,” the composition of (AnyPrin) and (Impl) “AnyImpl,” and the composition of (AnyPrin) and (Cont) “AnyCont.” The conclusion of AnyCont can serve as premise only in (Cont) or (Del). Let us call the composition of AnyCont and (Cont) “AnyContCont,” and the composition of AnyCont and (Del) “AnyContDel.”

Let us state the rules AnyTrans, AnyImpl, AnyContCont, and AnyContDel below:

$$\frac{\mathrm{Self}\ : \mathrm{Bind}(X,Y)}{\mathrm{Self}\ : \mathrm{Bind}(X,\mathrm{AnyPrin})} \qquad \text{(AnyTrans)}$$

$$\frac{P \Rightarrow Q}{\mathrm{Delegate}(\mathrm{AnyPrin},P) \Rightarrow \mathrm{Delegate}(X,Q)} \qquad \text{(AnyImpl)}$$

$$\frac{\begin{array}{l}\text{Self} : \text{Delegate}(\text{AnyPrin}, P)\\ \text{Self} : \text{Bind}(A, X)\end{array}}{\text{Self} : \text{Delegate}(A, P)} \qquad \text{(AnyContCont)}$$

$$\frac{\begin{array}{rcl}\text{Self} & : & \text{Delegate}(\text{AnyPrin}, P)\\ X & : & P'\\ P & \Rightarrow & P'\end{array}}{\begin{array}{rcl}\text{Self} & : & P'\end{array}} \qquad \text{(AnyContDel)}$$

Note that although there are two possible ways that (AnyPrin) can be composed with (Trans) (the conclusion of (AnyPrin) can be used as either the first or second premise of (Trans)) there is only one (AnyTrans) rule. Here is why: Let us call the composition in which the conclusion of (AnyPrin) is used as the first premise of (Trans) AnyTrans'. AnyTrans' would conclude Self : $\text{Bind}(A,B)$ from Self : $\text{Bind}(\text{AnyPrin},B)$. The premise of AnyTrans' (Self : $\text{Bind}(\text{AnyPrin},B)$), however, can not be concluded from any inference rule, unless AnyPrin was already bound to something in the axioms. Hence, (if we establish the reasonable requirement that thou shall not bind AnyPrin to another principal in thy axioms), the rule AnyTrans' could never be applied. Therefore, we include only the rule AnyTrans, which uses the conclusion of (AnyPrin) as the second premise of (Trans).

It is trivial to prove that each of these rules is a theorem in $\mathfrak{L}$. Furthermore, if we substitute the rule (AnyPrin) in $\mathfrak{L}$ with the rules (AnyTrans), (AnyImpl), (AnyContCont), and (AnyContDel), we can still prove everything provable in $\mathfrak{L}$ except for statements that immediately follow from (AnyPrin) or AnyCont. However, those statements are of the form Self : $\text{Bind}(X,Y)$ or Self : $\text{Delegate}(X,Y)$, which means they do not interest us for the purpose of constructing $\mathfrak{L}'$.

In a similar manner, we replace (Mon) in $\mathfrak{L}$ with the set of all possible compositions of rule (Mon) with rules that take the conclusion of (Mon) as one of their premises. These rules are (Mon) itself, (Trans), (Impl), and (Cont). Note that (Mon) can be composed with (Trans) in either of two ways: the conclusion of (Mon) could be the first or the second premise of (Trans). So, following the naming scheme from above (and acknowledging that there are two possible compositions of (Mon) and (Trans)), we could replace (Mon) with rules "MonTrans1," "MonTrans2," "MonCont," "MonImpl," and "MonMon."

But rather than introducing the rule MonMon, we note that in order to prove something other than $\text{Self} : \text{Bind}(X\text{'s}Z, Y\text{'s}Z)$, another rule has to be eventually applied after the application of the (Mon) rule. That leaves only rules (Trans), (Impl), and (Cont), but it also makes it necessary to account for the fact that rule (Mon) might have to be used consecutively a number of times for a proof in $\mathfrak{L}$. Therefore, we replace the rule (Mon) in $\mathfrak{L}$ by the infinite set of rules MonTrans1, MonTrans2, MonCont, MonImpl, MonMonTrans1, MonMonTrans2, MonMonCont, MonMonImpl, MonMonMonTrans1 *etc.* like this:

$$\frac{\begin{array}{l}\text{Self} : \text{Bind}(X, Y)\\ \text{Self} : \text{Bind}(Y\text{'s}A, Z)\end{array}}{\text{Self} : \text{Bind}(X\text{'s}A, Z)} \qquad \text{(MonTrans1)}$$

$$\frac{\begin{array}{l}\text{Self} : \text{Bind}(Z, X\text{'s}A)\\ \text{Self} : \text{Bind}(X, Y)\end{array}}{\text{Self} : \text{Bind}(Z, Y\text{'s}A)} \qquad \text{(MonTrans2)}$$

$$\frac{\begin{array}{l}\text{Self} : \text{Bind}(X,Y) \\ \text{Self} : \text{Delegate}(Y\text{'s}A,P)\end{array}}{\text{Self} : \text{Delegate}(X\text{'s}A,P)} \qquad \text{(MonCont)}$$

$$\frac{\begin{array}{c}\text{Self} : \text{Bind}(Y,X) \\ P \Rightarrow Q\end{array}}{\text{Delegate}(X\text{'s}A,P) \Rightarrow \text{Delegate}(Y\text{'s}A,Q)} \qquad \text{(MonImpl)}$$

$$\frac{\begin{array}{l}\text{Self} : \text{Bind}(X,Y) \\ \text{Self} : \text{Bind}(Y\text{'s}A\text{'s}B,Z)\end{array}}{\text{Self} : \text{Bind}(X\text{'s}A\text{'s}B,Z)} \qquad \text{(MonMonTrans1)}$$

$$\vdots$$

Again, it is trivial to show that all the Mon* rules are theorems in $\mathfrak{L}$. Likewise, by replacing (Mon) with the infinite set of rules above we lose only the ability to prove statements that follow directly from (Mon), which means they are of the form Self : $\text{Bind}(X\text{'s}Z, Y\text{'s}Z)$, and therefore not interesting for us when constructing $\mathfrak{L}'$.

To summarize: In this section, we have constructed a logic $\mathfrak{L}'$ from logic $\mathfrak{L}$ by removing inference rules (AnyPrin) and (Mon), and replacing them by the Any* and Mon* rules described above. We have shown that for goals other than Self : $\text{Bind}(X,Y)$ and Self : $\text{Delegatef}(X,P)$, a proof in $\mathfrak{L}$ exists if and only if a proof exists in $\mathfrak{L}'$, *i.e.* the two logics are equivalent for those goals.

Incidentally, in the Placeless Documents System, the prover never has to prove goals of the form Self : $\text{Bind}(X,Y)$ or Self : $\text{Delegatef}(X,P)$ (only goals of the form Self : *Perm*, where *Perm* is a primitive permission), so as far as Placeless is concerned, $\mathfrak{L}$ and $\mathfrak{L}'$ are equivalent.

B.2 A Proof-Finding Algorithm

We will now sketch an algorithm that implements a decision procedure for $\mathfrak{L}'$.

The algorithm is a simple forward-chaining rule-based theorem prover. Its main loop iterates over a set of statements that are known to be true. This set is initialized with statements describing the local policy, the local name space, and the statement describing the desired access. Then, subprocedures implementing the various inference rules of $\mathfrak{L}'$ are invoked, and the conclusions of the inference rules are added to the set of true statements.

There is exactly one subprocedure for every rule in $\mathfrak{L}'$, with the following exceptions:

- There is no subprocedure for rule (Impl), (AnyImpl), and (MonImpl).
- There is only one subprocedure that handles all of MonTrans1, MonMonTrans1, MonMonMonTrans1, *etc.*
- Likewise, there is only one subprocedure for all the Mon*Trans2 rules, and for all the Mon*Cont rules.

Because of the way the set of true statements is initialized, and because there are no subprocedures for the three Impl rules, the set of true statements will never contain a statement of the form $P \Rightarrow Q$. It will contain only statements of the form $Prin : Perm$.

The way the algorithm implements the rules (Impl), (AnyImpl), and (MonImpl) is by *backward chaining*. When the subprocedure implementing rule (Del) needs to establish $P \Rightarrow P'$, it does not expect to find that statement in the set of true statements. Rather, it will try to find the premises of the various Impl rules in that set (or — in the case where the premise is also of the type $P \Rightarrow Q$ — it recurses on itself).

The main loop continues to call subprocedures, as long as they add new statements to the set of true statements. If none of the subprocedures could add another statement, we break out of the main loop and check whether the goal is in the set of true statements. If it is, the algorithm has found a proof of the goal. If it is not, the algorithm could not find a proof.

Lemma 7. *The algorithm as described above always terminates.*

Proof. Let us define the weight W for every statement of $\mathfrak{L}'$ to be the positive integer that equals the number of nodes in the parse tree if the statement is parsed according to the EBNF definition on page 16. The weight W is a measure for the (syntactic) complexity of a statement.

We note that none of the subprocedures introduces statements that are heavier than all of its premises. In fact, the only rules in $\mathfrak{L}'$ that conclude heavy statements are the Impl rules, which do not have corresponding subprocedures in our algorithm.

We also note that none of the subprocedures' conclusions use "building blocks" (*i.e.* principals, strings, primitive permissions, *etc.*) that did not already exist in the subprocedures' premises. Therefore, all the building blocks that make up statements introduced by subprocedures already exist in the initial state of the set of true statements (axioms).

There exist only a finite number of axioms. Therefore, there exist only a finite number of building blocks that can show up in conclusions introduced by subprocedures. Since these conclusions are bounded in syntactical complexity, there can be only a finite number of conclusions introduced by all the subprocedures.

Therefore, the algorithm must terminate. □

As is apparent from the description of the algorithm so far, it can prove only statements of the form $Prin : Perm$, but not statements of the form $P \Rightarrow Q$. However, because of

Lemma 7 we can extend the algorithm to find proofs for any statement of the form $P \Rightarrow Q$ as follows:

- Run the main loop of the algorithm as above to completion.
- If the goal is of the form $Prin : Perm$, check whether that goal is in the set of true statements. If it is, return `true`; otherwise return `false`
- Otherwise (*i.e.*, the goal must be of the form $P \Rightarrow Q$), check whether the premises of one of the Impl rules concluding $P \Rightarrow Q$ are in the set of true statements. If the premises include a statement of the form $P' \Rightarrow Q'$, recurse. If premises supporting $P \Rightarrow Q$ can be found this way, return `true`, otherwise return `false`.

B.3 Soundness and Completeness of the Algorithm

Obviously, the algorithm as presented in the previous section is sound with respect to $\mathfrak{L}'$. If the algorithm signals finding of a proof, it has actually constructed a proof according to the inference rules of $\mathfrak{L}'$, so there must exist one. Therefore, the algorithm presented above is sound with respect to the logic.

Lemma 8. *The algorithm always enumerates all statements of the form $Prin : Perm$ that are provable from a given set of axioms.*

Proof. Let us define a non-negative integer "proof-length" for every provable statement in $\mathfrak{L}'$. Axioms have proof-length zero, and statements that follow, by some inference rule, from premises with proof-lengths $l_1, l_2, \ldots, L_n$, have proof-length $\max(l_1, \ldots, l_n) + 1$.

Let us assume that our algorithm enumerated all provable statements of form $Prin : Perm$ up to proof-length $n-1$, but that there exists a provable statement of form $Prin : Perm$ of proof-length n that our algorithm cannot find.

That statement must be the conclusion of inference rule (Del), because otherwise a straightforward application of one of the other subprocedures would have yielded our statement $Prin : Perm$.

So, in fact, our statement must be of the form $\text{Self} : P'$, and the corresponding premises of rule (Del) have at most proof-length $n-1$. They are: $\text{Self} : \text{Delegate}(X,P)$, $X : P'$, and $P \Rightarrow P'$. Since both $\text{Self} : \text{Delegate}(X,P)$ and $X : P'$ are of the form $Prin : Perm$ and have proof-length of at most $n-1$, they are in the set of already known-to-be-true statements.

$P \Rightarrow P'$ also has proof-length of at most $n-1$. It is either an axiom, or directly or indirectly relies on statements of the kind $\text{Self} : \text{Bind}(X,Y)$ (by way of the various Impl rules). However, those Bind statements must have proof length smaller than $n-1$, so they must be in the set of known-to-be-true statements. Therefore, the backward-chaining implementation of the Impl rules will find them, the Del subprocedure will fire, and our statement $\text{Self} : P'$ will be added to the set of true statements. Therefore, our assumption that there exists a provable statement of proof-length n that our algorithm cannot find must be false. $\square$

Lemma 8 shows that our algorithm can find proofs for all provable statements of form $Prin : Perm$. We are left to show that our algorithm (as augmented above) can also find a proof for all provable statements of form $P \Rightarrow Q$.

Our reasoning is very similar to that in the proof of Lemma 8. Let us consider a statement of form $P \Rightarrow Q$. This statement will either be an axiom or it will, directly or indirectly, rely on statements of the form $\text{Self} : \text{Bind}(X,Y)$. As shown in Lemma 8, that Bind statement will be in the set of true statements after the main loop concludes, and therefore the backward-chaining decision procedure outlined above will establish that $P \Rightarrow Q$ is provable.

Therefore, our algorithm is complete with respect to $\mathfrak{L}'$.

This concludes the proof that $\mathfrak{L}'$ is decidable. As mentioned above, the prover in Placeless only ever needs to prove goals of the form Self : *Perm*, where *Perm* is a primitive permission. Therefore, the algorithm as presented above is a suitable candidate for a Placeless prover.

B.4 A Decision Procedure for $\mathfrak{L}$

We will now proceed and show that $\mathfrak{L}$ itself is decidable by giving a decision procedure that is sound and complete with respect to $\mathfrak{L}$. Let us call the decision procedure from Section B.2 A'. Here is a decision procedure for $\mathfrak{L}$. We are given a set of axioms and a goal G.

1. If the goal G is of form Self : Delegate(X,P), then

   (a) Make up a new primitive permission object Aux that is not mentioned in the axioms.

   (b) Change the $Perm1 \Rightarrow Perm2$ test in A' such that it is satisfied either if the backward chaining as explained above succeeds, or $Perm1$ is P and $Perm2$ is Aux. (We are essentially adding the implicit axiom $P \Rightarrow$ Aux).

   (c) Add the statement X : Aux to the axioms.

   (d) Run A' to find a proof for Self : Aux. If a proof is found, return `true`, otherwise, return `false`.

2. If the goal G is of the form Self : Bind(A,B), then

   (a) Make up a new primitive permission object Aux that is not mentioned in the axioms.

   (b) Run A' to find a proof for Delegate$(B,\text{Aux}) \Rightarrow$ Delegate(A,Aux). If a proof is found, return `true`, otherwise return `false`.

3. Otherwise, run A' to try to find a proof for G. Return `true` if a proof is found, and `false` otherwise.

Let us call this decision procedure A. Obviously, A always terminates. We will now prove that for each of these three cases, A finds a proof for G if and only if there exists a proof for G in $\mathfrak{L}$.

We start with the last case. G is not of the form $\text{Self} : \text{Bind}(A,B)$ or $\text{Self} : \text{Delegate}(A,B)$. Therefore, a proof for G in $\mathfrak{L}$ exists if and only if a proof in $\mathfrak{L}'$ exists, which is if and only if A' returns `true`, which is if and only if A returns true.

Let us now look at the second case. If there is a proof for $\text{Bind}(A,B)$ in $\mathfrak{L}$, then there is a proof for $\text{Delegate}(B,\text{Aux}) \Rightarrow \text{Delegate}(A,\text{Aux})$ in $\mathfrak{L}$ (use rule (Impl)). Since $\text{Delegate}(B,\text{Aux}) \Rightarrow \text{Delegate}(A,\text{Aux})$ is not of form $\text{Self} : \text{Bind}(X,Y)$ or $\text{Self} : \text{Delegate}(X,P)$, there must then exist a proof for $\text{Delegate}(B,\text{Aux}) \Rightarrow \text{Delegate}(A,\text{Aux})$ in $\mathfrak{L}'$. Since A' is complete with respect to $\mathfrak{L}'$, it will find that proof. Therefore, A will return `true`.

On the other hand, if A returns `true`, that means that A' has found a proof for $\text{Delegate}(B,\text{Aux}) \Rightarrow \text{Delegate}(A,\text{Aux})$. That means there exists a proof for $\text{Delegate}(B,\text{Aux}) \Rightarrow \text{Delegate}(A,\text{Aux})$ in $\mathfrak{L}'$ and, hence, in $\mathfrak{L}$. The only inference rule that could yield $\text{Delegate}(B,\text{Aux}) \Rightarrow \text{Delegate}(A,\text{Aux})$ in $\mathfrak{L}$ is (Impl) (remember that Aux is not mentioned in the axioms). Therefore, there must exist a proof in $\mathfrak{L}$ for its premises, among which is $\text{Self} : \text{Bind}(A,B)$.

Last, let us consider the first case. If there is a proof for $\text{Delegate}(X,P)$ in $\mathfrak{L}$, and we add $P \Rightarrow \text{Aux}$ and $X : \text{Aux}$ to the axioms, then there is a proof for $\text{Self} : \text{Aux}$ in $\mathfrak{L}$ (use rule (Del)). Since $\text{Self} : \text{Aux}$ is not of form $\text{Self} : \text{Bind}(X,Y)$ or $\text{Self} : \text{Delegate}(X,P)$, there must then exist a proof for $\text{Self} : \text{Aux}$ in $\mathfrak{L}'$. Since A' is complete with respect to $\mathfrak{L}'$, it will find that proof. Therefore, A will return `true`.

On the other hand, if A returns `true`, that means that A' has found a proof for Self : Aux. That means that there exists a proof for Self : Aux in $\mathfrak{L}'$ (given the two added axioms) and, hence, in $\mathfrak{L}$ (again, assuming that $P \Rightarrow$ Aux and X : Aux have been added to the axioms). Since Aux is a primitive permission, the only rule that could yield Self : Aux is (Del). Let us look at the premises of that rule, which must all be provable in $\mathfrak{L}$. First, there must be a statement of the form Y : Aux. No inference rule can create a statement of that form, so it must be among the axioms. Since Aux is not mentioned in the axioms other than in the statement X : Aux, Y must indeed be X. So we know that one of the premises is X : Aux. Now there must exist a permission Q such that Self : Delegate(X,Q) and $Q \Rightarrow$ Aux is true. The only way $Q \Rightarrow$ `Aux` can be established is by using the axiom $P \Rightarrow$ Aux, which means that Q is indeed P, and we know that Self : Delegate(X,P) is provable in $\mathfrak{L}$.

This concludes the proof that A is sound and complete with respect to $\mathfrak{L}$, which means that $\mathfrak{L}$ is decidable.

Bibliography

[1] M. Abadi. On SDSI's linked local name spaces. *Journal of Computer Security*, 6(1-2):3–21, October 1998.

[2] M. Abadi, M. Burrows, B. Lampson, and G. D. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[3] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Computer Science and Applied Mathematics. Academic Press, Orlando, FL, 1986.

[4] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, Singapore, November 1999.

[5] D. Balfanz, D. Dean, and M. Spreitzer. A security infrastructure for distributed Java applications. In *21th IEEE Computer Society Symposium on Research in Security and Privacy*, Oakland, CA, May 2000.

[6] D. Balfanz and E. Felten. Hand-held computers can be better smart cards. In *Proceedings of USENIX Security '99*, Washington, DC, August 1999.

[7] D. Balfanz and D. Simon. WindowBox: A simple security model for the connected desktop. In *Proceedings of 4th USENIX Windows Systems Symposium*, Seattle, WA, August 2000.

[8] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical report, MITRE Corporation, March 1976.

[9] Blaze, Feigenbaum, and Naor. A formal treatment of remotely keyed encryption. In *EUROCRYPT: Advances in Cryptology: Proceedings of EUROCRYPT*, 1998.

[10] M. Blaze. High-bandwidth encryption with low-bandwidth smartcards. *Lecture Notes in Computer Science*, 1039:33–??, 1996.

[11] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance checking in the PolicyMaker trust-management system. In *Proceedings of the 2nd Financial Crypto Conference*, volume 1465 of *Lecture Notes in Computer Science*, Berlin, 1998. Springer.

[12] D. Boneh, N. Modadugu, and M. Kim. Generating RSA keys on the PalmPilot with the help of an untrusted server. In *Proceedings of the 2000 RSA Data Security Conference*, San Jose, CA, January 2000.

[13] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[14] N. Daswani and D. Boneh. Experimenting with electronic commerce on the PalmPilot. In M. Franklin, editor, *Financial cryptography: Third International Conference, FC '99, Anguilla, British West Indies, February 22–25, 1999: proceedings*, volume 1648 of *Lecture Notes in Computer Science*, pages 1–16, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1999. Springer-Verlag.

[15] P. Dourish, K. Edwards, A. LaMarca, J. Lamping, K. Petersen, M. Salisbury, D. Terry, and J. Thornton. Extending document management systems with user-specific active properties. *ACM Transactions on Information Systems*, 1999. scheduled for publication.

[16] S. Dusse, P. Hoffman, B. Ramsdell, L. Lundblade, and L. Repka. *S/MIME Version 2 Message Specification*. IETF - Network Working Group, The Internet Society, March 1998. RFC 2311.

[17] S. Dusse, P. Hoffman, B. Ramsdell, and J. Weinstein. *S/MIME Version 2 Certificate Handling*. IETF - Network Working Group, The Internet Society, RFC 2312 edition, March 1998.

[18] W. K. Edwards and A. LaMarca. Balancing generality and specificity in document management systems. In *Proceedings of the IFIP Interact '99 Conference*. IFIP, 1999.

[19] J.-E. Elien. Certificate discovery using SPKI/SDSI 2.0 certificates. Master's thesis, Massachusetts Institute of Technology, May 1998.

[20] C. M. Ellison. Establishing identity without certification authorities. In *Proceedings of the 6th USENIX Security Symposium*, San Jose, July 1996.

[21] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen. *SPKI Certificate Theory*, September 1999. RFC2693.

[22] E. W. Felten, D. Balfanz, D. Dean, and D. S. Wallach. Web spoofing: An internet con game. In *Proceedings of 20th National Information Systems Security Conference*, Baltimore, MD, October 1997.

[23] A. O. Freier, P. Karlton, and P. C. Kocher. *The SSL Protocol Version 3.0*. IETF - Transport Layer Security Working Group, The Internet Society, November 1996. Internet Draft (work in progress).

[24] H. Gobioff, S. Smith, J. D. Tygar, and B. Yee. Smart cards in hostile environments. In *Proceedings of The Second USENIX Workshop on Electronic Commerce*, Oakland, CA, November 1996.

[25] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the Sixth USENIX Security Symposium*, 1996.

[26] L. Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, 1999.

[27] 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`, 1997.

[28] J. Y. Halpern and R. van der Meyden. A logic for SDSI's linked local name spaces. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pages 111–122, Mordano, Italy, June 1999.

[29] J. Howell and D. Kotz. An Access-Control Calculus for Spanning Administrative Domains. Technical Report PCS-TR99-361, Dartmouth College, Computer Science, Hanover, NH, Nov. 1999.

[30] Institute for Applied Information Processing and Communications, Graz University of Technology. *IAIK JCE*, 1999. `http://jcewww.iaik.tu-graz.ac.at/index.htm`.

[31] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, volume 21, pages 105–106, 1987.

[32] B. Lampson. Protection. In *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*, pages 437–443, Princeton University, March 1971.

[33] H. M. Levy. Digital Press, Bedford, MA, 1984.

[34] National Institute of Standards and Technology, U.S. Department of Commerce. *Data Encryption Standard*, December 1993. NIST FIPS PUB 46-2.

[35] National Institute of Standards and Technology, U.S. Department of Commerce. *Digital Signature Standard*, May 1994. NIST FIPS PUB 186.

[36] Netscape Communications Corporation, Mountain View, California. *Implementing PKCS#11 for the Netscape Security Library*, 1998. `http://developer.netscape.com:80/docs/manuals/security/pkcs/pkcs.htm`.

[37] R. Rivest. *A Description of the RC2(R) Encryption Algorithm*. IETF - Network Working Group, The Internet Society, March 1998. RFC 2268.

[38] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[39] RSA Laboratories. *PKCS#11: Cryptographic Token Interface Standard, Version 2.0*, July 1997.

[40] T. Sander and C. F. Tschudin. On software protection via function hiding. In D. Aucsmith, editor, *Information Hiding: Second International Workshop*, volume 1525 of *Lecture Notes in Computer Science*, pages 111–123, Portland, Oregon, U.S.A., 1998. Springer-Verlag, Berlin, Germany.

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

[42] B. Schneier. *Applied Cryptography*, chapter 15.2 Triple Encryption, pages 358–363. John Wiley, 1996.

[43] D. A. Solomon and M. E. Russinovich. *Inside Microsoft Windows 2000, Third Edition*. Microsoft Press, 2000.

[44] J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In USENIX Association, editor, *USENIX Conference Proceedings (Dallas, TX, USA)*, pages 191–202, Berkeley, CA, USA, Winter 1988. USENIX Association.

[45] A. S. Tanenbaum, S. J. Mullender, and R. van Renesse. Using sparse capabilities in a distributed operating system. In *6th International Conference on Distributed Computing Systems*, pages 558–563, Los Angeles, CA, May 1986. IEEE Computer Society. 63-68 in Amoeba book.

[46] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium on Operating System Principles*, 1993.

[47] K. M. Walker, D. F. Sterne, M. L. Badger, M. J. Petkac, D. L. Sherman, and K. A. Oostendorp. Confining root programs with domain and type enforcement. In *Proceedings of the Sixth USENIX Security Symposium*, 1996.

[48] J. E. White. Telescript technology: The foundation for the electronic marketplace. White paper, General Magic, Inc., 2465 Latham Street, Mountain View, CA 94040, 1994.

[49] J. E. White. Mobile agents. In J. Bradshaw, editor, *Software Agents*. AAAI Press and MIT Press, 1996.

[50] A. Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *Proceedings of the 8th USENIX Security Symposium*, Washington, DC, August 1999.

[51] B. S. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994.

[52] E. Young et al. SSLeay and SSLapps. `http://psych.psy.uq.oz.au/~ftp/Crypto/`.

[53] Q. Zhong. Providing secure environments for untrusted network applications - with case studies using virtual vault and trusted sendmail proxy. In *Proceedings of Second IEEE International Workshop on Enterprise Security*, pages 277–283, Los Alamitos, CA, 1997. IEEE CS Press.