A Trusted Execution Platform for Multiparty Computation

by

Sameer Ajmani

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2000

Author .....
Department of Electrical Engineering and Computer Science
July 13, 2000

Certified by .....
Barbara H. Liskov
Ford Professor of Engineering
Thesis Supervisor

Certified by .....
Robert T. Morris
Assistant Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Committee on Graduate Students

A Trusted Execution Platform for Multiparty Computation

by

Sameer Ajmani

Submitted to the Department of Electrical Engineering and Computer Science
on July 13, 2000, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

The value of data used in computation is increasing more rapidly than the security of the computation environment. Users are submitting private personal and financial information to untrusted programs, even though the programs cannot guarantee the privacy of that information. This problem is even more pronounced for programs that are provided through the Internet, such as servlets and applets.

Sandboxing and runtime policy mechanisms are designed to prevent such programs from leaking information, but these techniques are either too weak or too restrictive to support useful information sharing. Myers' *decentralized label model* addresses this problem by tracking privacy policies on individual pieces of data as they flow through a program. This thesis presents a system that enforces these policies and allows mutually-distrusting parties to share data in computation.

The Simple Public Key Infrastructure (SPKI) provides name resolution and authorization services without depending on a central authority. This thesis describes a system that combines SPKI with Myers' label model to connect the names and policies in programs with real-world users and permissions. Users must trust the system with their private data; in return, the system protects their data from release to untrusted parties.

The system is called the Trusted Execution Platform (TEP). This thesis presents the design and implementation of TEP and analyzes its performance. TEP ensures that the applications it runs protect the privacy of classified data used in computation.

Thesis Supervisor: Barbara H. Liskov
Title: Ford Professor of Engineering

Thesis Supervisor: Robert T. Morris
Title: Assistant Professor

Acknowledgments

There are many people I must thank for helping me reach this goal.

First, I thank my advisors, Barbara Liskov and Robert Morris. Barbara continually challenges me and helps me focus my efforts, while Robert inspires me with his enthusiastic debates and his dedication to to improving design. Thanks to both of them for their time and support.

Without the work of Andrew Myers and Ronald Rivest, this thesis would not have been possible. Their systems feature a beauty and balance that I can only hope to capture in my work.

Many parts of this thesis are results of long discussions with my fellow researchers. I thank Miguel Castro, in particular, for helping me with everything from basic concepts to nuances of my design. Chandra Boyapati and Nick Mathewson have also helped me solve several challenging problems.

I'm not sure I would have survived this long without some fun in the office. My main partner in crime has been Ziqiang Tang, who distracts me with all sorts of games, web sites, and random discussions. Long discussions with Kincade Dunn also help by making sure I use the *other* half of my brain.

Of course, without my parents, I would never have had the opportunity to explore a career in research. Their dedicated work ethic and constant love and support help me accomplish anything I set my mind to do. My friends, too numerous to list here, have also provided me with an incredible amount of support. I thank them all for being a part of my life.

Finally, I thank my fiancée, Mandi White, for being the most loving and wonderful person that I have ever known. Mandi has made several sacrifices just to be with me during my graduate research, and I can't thank her enough. I look forward to a lifetime together with her.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the number of broadband network connections to the home and office increases, more and more applications are being delivered over the Internet. With access to these network connections, applications can freely disseminate any information provided to them by the user. The user's privacy is at risk.

Suppose a tax preparation company provides a program called "WebTax" for generating tax forms from users' income data. Currently, such applications are deployed in one of two ways: either the user sends income data to the tax preparer for processing or the user downloads the entire program to his or her local machine. The preparer's database, as well, must be sent to the machine where the program resides.

In Figure 1-1, the user, Bob, sends his private income data directly to the tax preparer and must trust the preparer not to abuse his information. In Figure 1-2, Bob's data is safe as long as WebTax cannot communicate back to the tax preparer. Unfortunately, such programs often need to contact their server for data, so WebTax could send Bob's income data back to the server. In either case, the Bob's private data could be released to an untrusted party.

A number of solutions have been proposed to address this problem, ranging from sandboxing applications to prohibiting application downloads completely. So far, the solutions have either been too restrictive or too permissive, and it has been difficult to find a satisfying balance. Recently, Myers developed a technique based on *information flow control* that allows programmers to decide explicitly when and how to disseminate information [ML97, ML98, Mye99a, Mye99b, ML00]. Combined with static checking, this technique can ensure a program will not leak information without permission.

Returning to our example, consider a tax program that can be checked statically not to leak information provided to it by the user. Bob can download the application, check it, and run it locally. The application can contact its server for data but is guaranteed not to release Bob's income data. This protects Bob's privacy concerns, but introduces a new problem: the server must send its proprietary data to Bob. If Bob were malicious, he could read the server's data and take advantage of the tax preparer.

This is a situation where mutually-distrusting parties wish to share data in computation. Neither party is willing to release their data to the other, but the computa-

tion cannot proceed without both sets of data. To solve this problem, we introduce a *trusted third party.* The third party serves as a mediator between mutually-distrusting parties that can protect their private data while running the computation.

In our tax preparation example, we can introduce a *Trusted Execution Platform* (TEP) between Bob and the tax preparer. WebTax, the user's income data, and the server's proprietary data are all sent to TEP. TEP runs WebTax on the data and returns the result to Bob (see Figure 1-3). By applying the information flow checking techniques mentioned earlier, TEP can allow WebTax to contact other parties without risking information leaks. This makes it possible to share data between mutually-distrusting parties while protecting their privacy.

TEP must perform a number of tasks to protect the privacy of the involved parties. TEP must check that WebTax is authorized by the Tax Preparer to leak information to Bob, since the Tax Form contains a small amount of information from the preparer's database. TEP must ensure that the parties connected to WebTax are indeed Bob and the Tax Preparer, rather than attackers attempting to fool the system into releasing private data. TEP must encrypt any private data transferred over the network, since eavesdroppers may be listening to TEP's transmissions, and TEP must protect the integrity of that data from attackers. This thesis will describe how each of these tasks is implemented.

TEP's execution model is not limited to two-party computation. In fact, TEP can support multiparty computation with an arbitrary number of readers and writers. Consider the online version of a silent auction (Section 4.4.2). Each bidder wants to keep his or her bid and identity secret, only revealing themselves if they win the auction. The auctioneer wants to keep the reserve price secret until it is met. The bidders can trust TEP to protect their privacy concerns, although they each need to authorize TEP to reveal their bid if they win. The auctioneer is assured the reserve price will not be revealed until it is met.

This thesis describes the design and implementation of a Trusted Execution Platform for multiparty computation. The next section describes the structure of TEP in more detail, and Section 1.2 provides additional motivation for the work. Section 1.3 gives an overview of the rest of the thesis.

1.1 System Structure

TEP's main function is to load and execute programs on behalf of its clients. Although the execution model is simple, protecting clients' privacy requires some complexity. Thankfully, we can decompose this task into a number of distinct functions, as described below:

Program Validation TEP must check that each program it runs cannot leak data, except when authorized. We use static information flow checking to enforce this constraint. Only parties whose information is declassified by a program need grant it authorization.

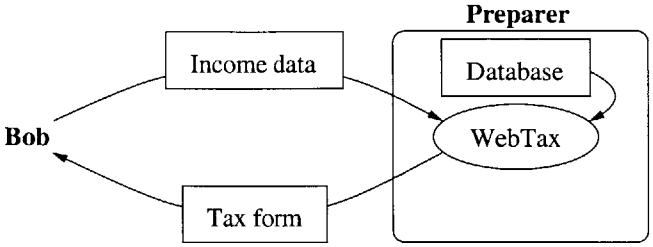

Figure 1-1: Server-side computation

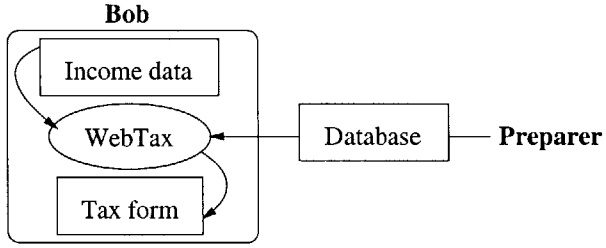

Figure 1-2: User-side computation

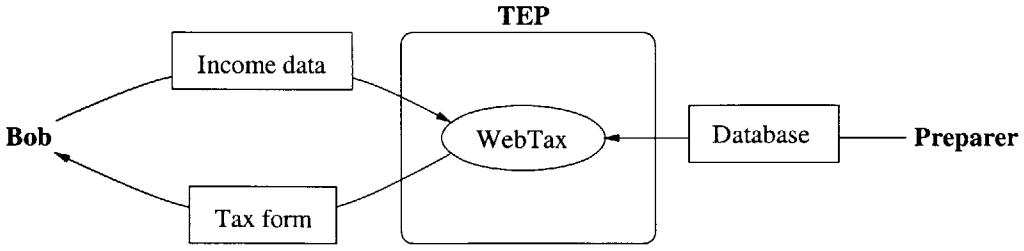

Figure 1-3: Third-party computation

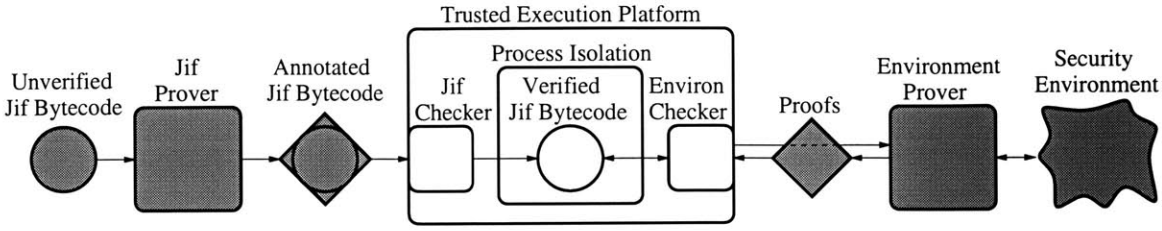

Figure 1-4: System Components: trusted are light; untrusted are dark

Authorization TEP must check that any program that leaks information is authorized by the parties whose information is leaked. This ensures that each party recognizes when its private information is released to other parties.

TEP must also check whether one principal authorizes another to leak its data. This allows programs running with one principal's authority to leak data provided by another principal.

Name Resolution Since programs refer to principals by name, TEP must be able to connect those names with actual users. This ensures that information leaked to a named principal is released only to the intended user.

Process Isolation Since TEP is a shared service, it may run multiple untrusted programs simultaneously. TEP must isolate these programs from one another by separating their heap memory and class data. This ensures that programs cannot access private data handled by other programs.

Each of these tasks is implemented by components in TEP. We find that by decomposing certain components into trusted and untrusted subcomponents, we can reduce the size of TEP's trusted base significantly. By minimizing TEP's trusted base, we increase our confidence in the system and decrease the number of potential attacks. TEP's components are depicted in Figure 1-4.

1.1.1 Program Validation

We implement program validation using Myers' static information flow checking techniques. Myers developed a set of program annotations (labels) that allow programmers to express privacy restrictions on data naturally [ML00]. When combined with Java type declarations, these labels form an extended type system that can be checked statically at compile time. If a labelled program can be type checked, it is guaranteed not to leak information through explicit or implicit data channels. Myers implemented a language that incorporates these labels called *Jif*, for Java Information Flow.

TEP requires that all programs it runs be bytecode annotated by a *Jif Prover*. The Jif Prover handles all the complicated inference necessary to prove that a program does not leak information. The annotated bytecode generated by the prover can then be checked by a small, trusted *Jif Checker*. Since the checker rejects unsafe

programs, the prover can be moved outside the trusted base. We expect the prover will be much larger than the checker, so this separation allows us to reduce the size of TEP's trusted base.

Myers' thesis [Mye99b] provides the full syntax and semantics of the Jif language, while Mathewson's thesis [Mat00] describes the Jif bytecode representation and verification algorithm. This thesis will assume that both the Jif Prover and the Jif Checker have been implemented. We will describe Myers' label system in more detail in the next chapter.

1.1.2 Authorization and Name Resolution

We implement authorization and name resolution using the Simple Public Key Infrastructure (SPKI) [EFL+98, Ell99, EFL+99, EFL+00]. SPKI is a fully-decentralized public key infrastructure that supports authorization and naming with certificates. TEP assumes that each principal controls an asymmetric key pair and uses public keys for identity. Principals can issue certificates – signed statements – that name or authorize other principals and programs. By chaining these certificates, TEP can resolve names into keys and determine whether a program is authorized by a given principal.

We will refer to the set of certificates accessible via the network as the *Security Environment.* An untrusted component called the *Environment Prover* can search for certificate chains that prove authorizations and name resolutions. The small, trusted *Environment Checker* can check these proofs efficiently. By separating security environment access into trusted and untrusted components, we reduce the size of TEP's trusted base. We describe these components in detail in Chapters 3 and 4.

1.1.3 Process Isolation

We implement *process isolation* using Java classloaders and security managers. Each Java classloader defines its own class namespace and static data area [LB98]. TEP ensures that each program thread has access only to classes within its thread. Java's security managers and policy mechanism prevent untrusted code from accessing system resources directly [Jav98]. Since TEP must ensure that programs send information only to authorized principals, programs are not allowed to open network channels without using TEP's API. Since TEP must protect its own disk space and classes, programs are not allowed to access the disk. Instead, TEP provides mechanisms for storing persistent data on other servers. Details on these protection mechanisms and TEP's API are given in Chapter 4.

1.2 Motivation

Given most users' lack of concern about security, it may seem overzealous to develop a service whose sole purpose is to protect the privacy of user data. Yet, as increasingly sensitive data is used in computation, attackers have more motivation to take

advantage of insecure systems. TEP simplifies the task of protecting sensitive data by performing the necessary security checks on behalf of clients. TEP can also allow clients to access a service anonymously, which may be difficult otherwise if clients must contact servers directly.

Distributing a computation increases the number of possible attacks on a system, so we must understand why a trusted third party is necessary. TEP allows mutually-distrusting parties to share private data in computation, and implements the mechanisms required to protect each party's privacy. Though there exist cryptographic techniques for multiparty-computation without a third party, such techniques are inefficient and impractical [Gol98]. Without a third party, some party must sacrifice its privacy to run the computation.

An important feature of TEP is that it removes the burden of implementing most security checks from the client and server sides of a computation. The client need only authenticate TEP before entrusting it with private data. Service providers need only check their code and grant it the appropriate permissions before releasing it to TEP; TEP then uses static information flow checks to ensure their programs won't leak information. Since TEP only releases data when it is explicitly declassified, providers and clients are protected against accidental leaks due to bugs. TEP also allows providers to separate public code and algorithms from private data used in computation.

This model shares the benefits of Sirer et al's distributed VM [SGGB99]: security checks are localized to a single system to ensure that security updates are made consistently for all parties. TEP can apply techniques to reduce the size of its trusted base and improve security without requiring changes from the users. By focusing security efforts on a few trusted systems rather than many untrusted ones, all parties involved in shared computation benefit.

1.3 Overview

The rest of the thesis is organized as follows:

Chapter 2 introduces Myers' decentralized label model and information flow control language, Jif. We describe how programs can be annotated and then checked for information leaks. We also define the services that TEP must implement to support Jif.

Chapter 3 connects language components used in Jif to cryptographic public keys and certificates. We present SPKI, the Simple Public Key Infrastructure, and describe how it is used to implement name resolution and authorization.

Chapter 4 details the design of the Trusted Execution Platform. We consider the trusted and untrusted components in turn, then discuss their interactions. We present additional examples to demonstrate how the system can be applied in practice.

Chapter 5 describes the implementation of TEP and analyzes its performance. We also propose techniques for improving performance.

Chapter 6 concludes the thesis by discussing related work and proposing areas for future research.

Chapter 2

Information Flow Control

This chapter presents Myers' decentralized label model and information flow control language, Jif. We show how labels are used to annotate programs and how such programs can be checked for information leaks. We then describe how TEP implements the runtime environment required by Jif.

2.1 Myers' Decentralized Label Model

Privacy depends on controlling the flow of information in a system. A simple way to guarantee privacy is to prevent any information from leaving the system, as in Java's *sandboxing* model. Unfortunately, this also prevents parties from sharing the results of a computation.

The *Java policy* model supports finer-grained access control, allowing the system administrator to select what channels should be open to specified programs [Jav98]. This model introduces the problem of deciding which programs to trust with private information, yet it provides no automated techniques for checking unknown programs. Java Policies do nothing to protect users against bugs in programs that leak information. Furthermore, trust decisions are centralized with the system administrator, so users cannot directly control the dissemination of their information.

Myers has developed a set of program annotations (labels) that allow individual users in the system to keep track of their own data in a program [ML00]. When combined with Java type declarations, these labels form an extended type system that can be checked statically at compile time. If a labelled program type-checks, it is guaranteed not to leak information through explicit or implicit data channels. When a program needs to release information, it requires explicit authorization from the user whose data is leaked. This system provides flexible and powerful control over the dissemination of private data and avoids the need for centralized authority.

2.1.1 The Program Boundary

Actions such as reading and writing data occur *within* a program and are isolated to the program's memory. A program can only release data to parties outside the

program through special *output channels*, and data can enter a program only through *input channels*. Myers' static checking rules track information flow within a program, up to and including the channel boundary. It is TEP's job to enforce the policies of principals who own that data by ensuring data written to output channels is sent only to authorized principals. TEP must also ensure that data read from input channels is labelled appropriately. Channels will be discussed in more detail later in this section.

2.1.2 Principals and the Acts-For Relation

When discussing information flow control and authorizations, we use the term *principal* to refer to any individual, group, or abstract role. Users like "Alice", groups like "Students", and roles like "Manager" are all principals. Principals can own data and can grant authorizations that allow other principals to read, modify, and disseminate that data.

When principal A grants the right to act on its behalf to principal B, we say that B *acts for* A. This means that any data owned by A is also owned by B: B can read the data, overwrite it, or even release it to other principals. Acts-for is a very powerful and dangerous relation: principals must be careful to allow only trusted principals to act on their behalf.

2.1.3 Label Structure

Myers' label system is *decentralized*: it allows individual principals to add independent *privacy policies* to data. Each policy restricts the set of principals that can read the data, and the label-checking rules ensure all policies are enforced simultaneously. Since each principal controls their own policies on data, they can independently *declassify* their policies without trusting other principals.

For example, consider a label that has two policies, one owned by Alice and the other by Bob. Each policy specifies a set of principals allowed to read the data; let's say Alice allows herself and Bob to read and Bob allows himself, Carol, and Fred. Such a label is written {`Alice: Alice, Bob; Bob: Bob, Carol, Fred`}.

The set of allowed readers in Alice's policy is {`Alice, Bob`} and in Bob's policy is {`Bob, Carol, Fred`}. Since the label system enforces all policies simultaneously, the principals allowed to read the data are those in the *intersection* of the reader sets: {`Bob`}. This makes intuitive sense: Bob is the only principal that both Alice and Bob allow to read in their policies. This means that data with this label can be written to an output channel that Bob can read from, but nobody else. If any other principal could read the data, there would be an information leak.

More formally, a label is a set of components called *policies*; each policy is composed of two parts, an *owner* principal and a set of *reader* principals. A label with two policies, each with two readers, is written $\{o_1 : r_1, r_2;\ o_2 : r_2, r_3\}$. Here, o_1, o_2, r_1, r_2, and r_3 denote principals. Every policy in a label must be obeyed when data exits the system, so only principals who act for a reader in *each* policy can read data with that label. Assuming all the above principals are distinct, only r_2 may read from data with this label.

The next several subsections describe how labels are applied to actual data in code and how they are manipulated by computation.

2.1.4 Relabeling Data

A program stores data in memory slots called *variables*; data is transferred between variables by assigning the value of one variable to another. Labels are incorporated into programs by associating a label with each variable. By transferring data between variables, the program effectively *relabels* the data.

Myers' label-checking rules ensure that a program maintains the privacy policies on data by requiring that each relabeling be a *restriction*. That is, the set readers allowed by the new label is, in all contexts, a subset of the readers allowed by the original label.

Myers defines these *safe relabelings*:

Remove a reader. It is safe to remove a reader from any policy in a label, since the changed policy is no less restrictive than the original.

Add a policy. It is safe to add a policy to a label, since all existing policies are still enforced.

Add a reader. It is safe to add a reader r' to a policy if r' acts for some reader r already allowed by the policy. Since r' has all the privileges of r already, adding r' does not compromise the safety of the policy.

Replace an owner. It is safe to replace the owner of a policy o with o' if o' acts for o. The owner of a policy is the only principal that can grant authority to declassify the policy, so replacing o with the more restrictive authority o' is safe.

Each of these relabelings preserves the safety of each policy in a label. Myers defines a *complete relabeling rule* that specifies precisely when relabeling one label to another is safe [ML98]. We denote the relation "L_1 relabels to L_2" by $L_1 \sqsubseteq L_2$. By requiring that a program make only safe relablings, we can ensure that the program does not compromise the privacy requirements of any principal that owns the data.

2.1.5 Implicit Flows

Myers label-checking rules make it possible to track the transfer of information through *implicit* flows. Implicit flows result from changes in a program's control flow due to classified data. For example:

```
if (a) {
    b = true;
}
```

The value of the boolean variable `a` is transferred to `b` without explicit assignment. If the label on `a` is more restrictive than that `b`, this transfer leaks information. Myers'

rules keep track of these transfers by assigning a label to the program counter itself, pc.

In this example, the label on pc inside the conditional inherits the label on a. Each statement inside the conditional inherits the label on pc, so the assignment to b is legal only if the label on b is as restrictive as that on pc. Since the label on pcis at least as restrictive as that on a, this restriction detects any implicit information flow to b.

Implicit flows must be checked statically since runtime checks can leak information when they fail. Myers' system tracks implicit flows through all types of control flow mechanisms, including conditionals, loops, method calls, and exceptions.

2.1.6 Computing Values

Assignment is not the only way that programs handle data; programs also compute values from pieces of existing data. Since values are stored in labelled variables, it is necessary to derive labels for computed values. To avoid unnecessary restrictiveness, a derived label should be the least restrictive label that enforces the policies of its sources.

Since labels are simply sets of policies, the derived label is simply the *union* of the policies of its sources. We call this least restrictive label the *join* of the source labels; we denote the join of labels L_1 and L_2 by $L_1 \sqcup L_2$.

For example, consider the computation `x = a + b`. Suppose the label on a is {Alice: Alice} and the label on b is {Bob: Bob}. Then the derived label for x is {Alice: Alice; Bob: Bob}. This label enforces all the policies of the source data, so the program cannot leak data to any principal outside intersection of the two policies. In this example, the intersection of the reader sets is the null set, so *no* principal can read the data. Without means for relaxing policies, it is very easy to restrict data to the point that it becomes unreadable. Myers addresses this problem by introducing a second kind of relabeling for declassification.

2.1.7 Declassification

Declassification allows programs to relax the labels on data. More precisely, a program can relabel data to a label with a less restrictive policy if the program has the *authority* of the policy's owner. This means the principal who owns the policy to be declassified must explicitly give the program permission to do the declassification. One of TEP's jobs is to check that any program that declassifies data has the requisite authorizations.

For example, a program can assign data labelled {Alice: Alice; Bob: Bob} to a variable labelled {Alice: Alice, Bob; Bob: Bob} if it has Alice's authority. Alice's authority is required since her policy is being relaxed (to add Bob as a reader). Myers' rules will only allow this statement if the program has been granted authority by Alice's principal. Bob's authority is not required since his policy is unchanged. Notice that this declassification relabels unreadable data to data readable by Bob.

Since it is undesirable to give an entire program authority when only a few statements declassify data, programs can choose to use only a subset of their authorizations at any point in the code. The *current authority* at a particular point in a program is the set of all principals whose data can be declassified at that point. Myers's rules calculate the authority at each point in the program statically, so declassification authority checks incur no runtime overhead.

Myers defines this inference rule for declassification:

$$\frac{L_A = \bigsqcup_{(p \text{ in current authority})} \{p :\} \qquad L_1 \sqsubseteq L_2 \sqcup L_A}{L_1 \text{ may be declassified to } L_2}$$

This rule builds on the rules for relabeling and label join. It says that L_1 may be relabeled to L_2 if it is safe to relabel L_1 to the join of L_2 and L_A. Recall that the join of two labels is the union of their policies, and observe that L_A is defined as the join of all labels {p:}, where p is a principal in the current authority.

The policy {p:} is the most restrictive policy owned by p: no principal can read data whose label includes this policy. Any policy owned by p can be relabeled by restriction to {p:}. Therefore, this inference rule allows any relabeling that weakens policies owned by principals who have granted their authority to the program. Policies owned by principals outside that authority may not be changed. This rule defines a selective, decentralized version of declassification that allows programs to declassify data on behalf of consenting principals while protecting the policies of other principals. We discuss how principals grant their authority to programs in Section 2.2.

2.1.8 Channels

Channels are half-variables that allow data to enter and leave the labelled domain. Input channels are a source of labelled data; all data read from the channel inherits the channel's label. Output channels allow labelled data to exit the system; only data whose label is less restrictive than the channel label may be written to the channel. These mechanisms ensure that no information is leaked when data enters or leaves the system.

For example, data read from an input channel labelled {Bob: Bob} can be assigned to variables with the label {Bob: Bob} or {Bob:}. This way, the principal writing at the other end of the channel is assured that its data can be read only by Bob. Recall that within a program, labels ensure that Bob's policy is preserved as information flows from variable to variable. Channels protect the policy when the labelled data enters or leaves the program.

Data written to an output channel labelled {Alice: Alice} could have the label {Alice: Alice}, {Alice: Alice, Bob}, or even {} (a label with no policies that is readable by everyone). Principals whose data is handled by the program are assured that their private data will not be written to this channel unless they explicitly relabel it with Alice's policy. The policies of all other principals must be removed from data written to this channel, since the channel only accepts data with Alice's policy.

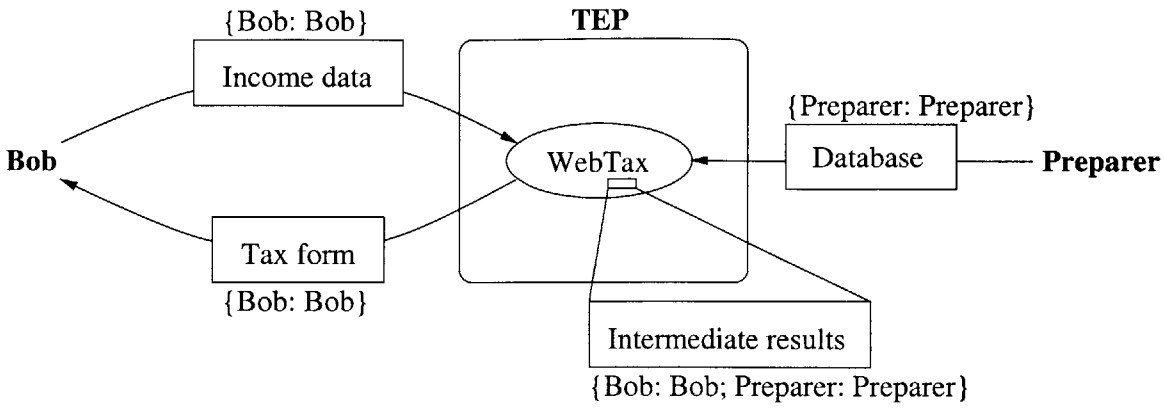

Figure 2-1: Labelled Application Example

Once data is written to a channel, the policies in its label can only be enforced by the receiving principal. TEP must ensure that the principal reading from the channel is a reader in the channel's label. This mechanism is described in detail in Chapter 4.

2.1.9 Example

We return to the tax preparation example from the Introduction to show how labels are used to control the flow of data in an application. Figure 2-1 depicts the WebTax application running on TEP with labelled data. The label {`Bob: Bob`} on the income data and tax form means that only Bob can read that data. The label {`Preparer: Preparer`} on the database means that only the tax preparer can read the database.

When WebTax makes a calculation that uses Bob's income data and the Preparer's database, the result is the union of the policies from both sources. The label on the intermediate result, {`Bob: Bob; Preparer: Preparer`}, contains two policies: one owned by Bob and the other owned by the preparer.

It may seem odd to consider data with two contradicting policies; one claims the data may only be read by Bob and the other, the Preparer. In fact, such data can only be read by the program itself and the Trusted Execution Platform. TEP ensures that the program is unable to send unreadable data out of the system.

For the result to become readable by Bob, the Preparer must remove its privacy policy from the label with an explicit declassification. The declassification means the Preparer must acknowledge that a small amount of its information is leaked from the computation. To allow the program to execute the declassification, the Preparer must grant its authority to WebTax.

Once the data has been declassified, it may be written to the channel connected to Bob. The labels on the data ensure that only data labelled {`Bob: Bob`} will be written to the channel, and TEP ensures that only Bob can read from the channel. This way, Bob's data is kept secret, but he is able to obtain the result of a shared

computation.

2.2 Jif: Java Information Flow

Myers has designed an label-annotated version of the Java language called *Jif*, for Java Information Flow. Jif can be compiled into bytecode that TEP can check statically for information leaks. This section describes the language constructs of Jif and presents some additional restrictions required by TEP.

2.2.1 Language Constructs

To incorporate information flow control into a practical programming language, Jif adds some new constructs to the Java language. One that has already been mentioned is the declassify statement that allows a program to loosen the restrictions on a piece of data. The declassify statement requires that the program have the *static authority* of any principals whose policies are made less restrictive. This means a program that declassifies data must declare the authority it needs explicitly in its code. The declassify statement itself is removed after static analysis, since it requires no runtime checks.

A program can claim the authority of principals in three ways:

- Code itself can be granted authority by principals. Jif classes declare the authority they expect to receive in an authority declaration. The declaration `authority(Eve, Trevor, Fred)` means that the class expects to be authorized by each of those three principals. TEP checks that this class has in fact been granted authority by these principals before it loads the class. The mechanism for this check is described in Chapter 4.

  Methods can request any subset of the authority of their class by declaring an authority constraint in their method header. The body of a method gains the static authority of the principals listed in the constraint. The method constraint `authority(Trevor, Fred)` is legal if the authority declaration of its class contains at least these two principals. The body of this method gains the static authority of Trevor and Fred.

- A method can request the authority of its caller by declaring a caller constraint in its method header. The constraint `caller(Carol, Ted)` means that any code that calls this method must have been granted authority by both Carol and Ted. The method body then gains the static authority of both Carol and Ted.

  A caller constraint can name any number of principals and can also list runtime principal arguments to the method. Consider this example:

```
void foo(principal p) where caller(p)
{
    // use p's authority
}
```

The method `foo` declares a caller constraint for the runtime principal p, which is an argument to the method. This constraint means that any code that calls `foo` must have the static authority of p. Connecting static authority to runtime principals requires Jif's actsfor test, described next.

- A program can increase its authority if a principal in the current authority actsfor another principal whose authority is desired. For example, a program that has Alice's authority can augment its authority to include Bob's if Alice actsfor Bob. Programs can make this check at runtime with the Jif statement `actsfor(Alice, Bob) {...}`. If the check passes, the code inside the braces is executed with Bob's authority added to the current authority. If the check fails, the instructions in the braces are skipped.

  Runtime actsfor tests also allow code to gain the authority of runtime principals. Consider this example:

```
void bar(principal p) where authority(Bob)
{
    actsfor (Bob, p) {
        foo(p);
    } else {
        throw new FatalError();
    }
}
```

The method `bar` inherits the static authority of Bob from its class using an authority constraint. It then tests whether Bob actsfor the runtime principal p. If so, `bar` call the method `foo`, given above. This is legal since, inside the first set of braces, `bar` has increased its *static* authority to include p.

This example shows how Jif determines static authority by analyzing the control flow that leads to each point in a program. This incurs no runtime overhead, since all the analysis is done statically. Jif keeps track of the actsfor relationships that are known at each point in the program and uses them to augment the authority and caller declarations in the code. Principals are referred to by name and runtime principals are referred to by variable name, so the analysis need only consider names defined in the current program scope.

This example also shows the use of an optional `else` clause that can follow an actsfor test. The `else` clause is executed if the test fails; the code inside the second set of braces gains no additional authority. In this example, the `else` clause throws FatalError, Jif's only unchecked exception.

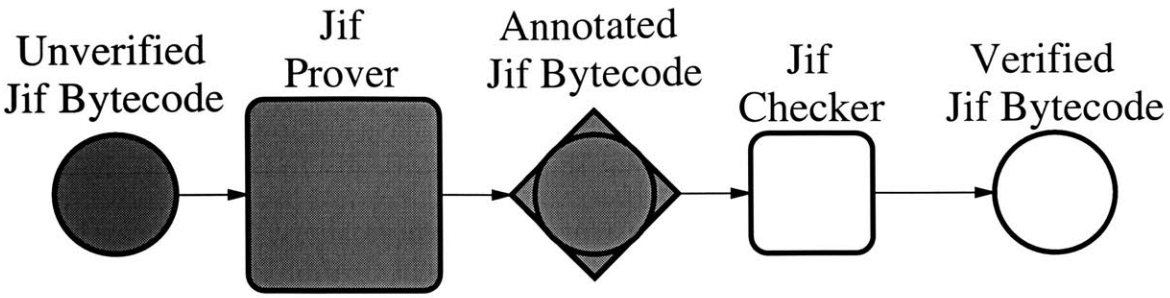

Figure 2-2: Jif Components: trusted are light; untrusted are dark

> A method can also check actsfor statically by putting an actsfor constraint its method header. Jif checks that this actsfor relationship has already been tested at each point where this method is called. For example, the constraint `actsfor(Alice, Bob)` requires that the relationship "Alice acts for Bob" be known at every point the method is called.

Once a program has the authority of a principal, it can declassify policies owned by that principal. This means authorizing unfamiliar code is very dangerous, since the code can leak the authorizing principal's private data. Jif alleviates this concern by forcing programmers to make declassifications explicit. If programmers understand each declassification and are willing to accept the amount of data that is leaked, they can authorize their programs with confidence.

Jif introduces a number of other mechanisms to make programming with labelled data more practical, such as label inference, label and principal polymorphism, and first-class runtime labels and principals [Mye99b]. Most Jif constructs are removed by static analysis, but actsfor checks, authority declarations, and channels must be implemented by TEP. The implementation of these constructs is described in Chapter 4.

2.2.2 Programming for TEP

Figure 2-2 depicts the Jif development process and components. Jif programs are written and compiled into bytecode in an untrusted environment. An untrusted *Jif Prover* annotates Java bytecode with information flow control data. TEP uses a trusted *Jif Checker* to check that incoming bytecode is label-correct and type-safe. Once the code has been verified, it is guaranteed not to leak information without the requisite authority.

To simplify the system and ensure security, TEP requires these additional properties of any program it runs:

- Jif programs must be distributed as JAR files (Java Archive, [Jav97]) that TEP can download. All classes required by the program, except for standard Java libraries and certain TEP classes, must be contained in the JAR file. This

allows TEP to verify the contents of the JAR and properly isolate the program's classes.

- Jif programs must not use Java library classes that circumvent TEP's security mechanisms. TEP enforces this at runtime using Java's standard security manager, although one could implement this check statically to reduce overhead.

- Methods that TEP calls directly (*top-level methods*) must have a particular method signature that allows TEP to simplify its invocation interface and move marshaling code out of the trusted base. We describe this signature in detail in Section 4.3.3.

- Jif classes must not declare the Java package `tep`: this is the package that contains TEP's trusted classes, so foreign classes are forbidden to override them. TEP rejects any such classes at load-time.

- Program and principal authorizations must be available to TEP through its security environment. This allows TEP to work with programs independently of their authorizations. This also allows providers to change authorizations on programs that are already deployed. This mechanism is described in Section 3.3.

TEP provides classes that implement I/O channels, secure data storage, and Jif's actsfor check. Programmers use stubs of these classes when writing their programs, but TEP substitutes its own trusted implementations when it runs the programs. These requirements and their implementations will be described in Chapter 4.

Chapter 3

The Security Environment

This chapter describes the design and implementation of a *security environment* that provides authorization and name resolution services. Figure 3-1 depicts the components of the security environment and their interactions. As with Jif, TEP uses a prover-checker pair to move information from the untrusted domain into the trusted domain.

Programs running on TEP issue authorization and name resolution requests to the environment, where an untrusted *Environment Prover* searches for evidence that supports the request. If it succeeds, the prover returns a proof. The trusted *Environment Checker* then checks the proof and reports whether the request is satisfied.

The security environment associates principal names and authorizations with cryptographic public keys and certificates. By accessing the security environment through a high-level interface, TEP can be designed independently of the implementation of the environment.

We use the Simple Public Key Infrastructure (SPKI, [EFL+98, Ell99, EFL+99, EFL+00]) to implement TEP's security environment and concepts in Jif. The next chapter will describe how TEP uses the security environment at runtime.

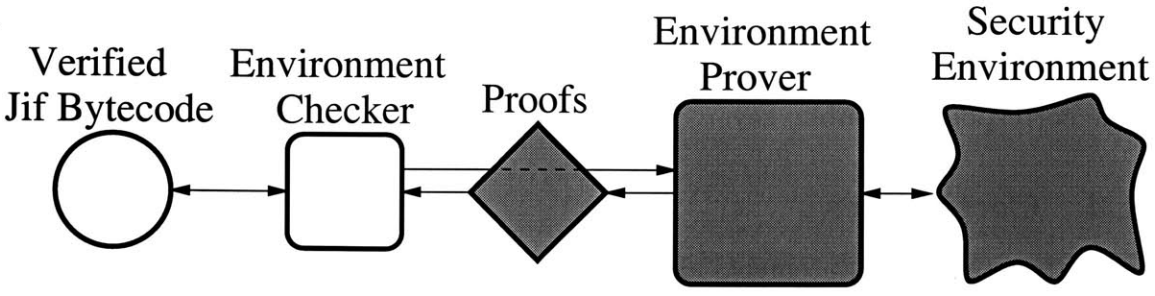

Figure 3-1: Security Components: trusted are light; untrusted are dark

3.1 Definitions

This section describes the basic components upon which the security environment is built. Although these definitions are presented in the context of SPKI, alternate implementations are available using other security infrastructures.

3.1.1 Keys

TEP uses both symmetric and asymmetric key systems. Symmetric keys are used for encrypting data to preserve its privacy on a network or on untrusted storage. They are also used to generate keys for Message Authentication Codes (MACs), which protect the integrity of data on the network.

Asymmetric key pairs (public and private keys) are used as the basis for digital identity. In SPKI, each key pair defines its own local "space" that contains all statements signed by the private key. We assume TEP's own public key is well-known.

3.1.2 Signatures

Digital signatures are special pieces of data that can be created only by a particular private key. Signatures uniquely identify the data they sign by calculating a one-way function on the digest of the data. Signatures can be verified with the corresponding public key to prove that the controller of the private key signed the data.

For example, Alice could write and sign a document saying that she will give all her money to Bob. Mallory could write a similar document saying that Alice will give all her money to Mallory. We choose only to believe the document signed by Alice. Digital signatures serve a similar purpose, but provide the additional guarantees that they are very hard to forge and can only be associated with a single document.

3.1.3 Principals

A principal is an individual, group, or role that has control over one or more asymmetric key pairs. A public key is the only global identifier for a principal in SPKI. Any user can generate an asymmetric key pair and sign statements with the private key. Other users can then associate signed statements with the principal identified by the corresponding public key.

3.1.4 Certificates

Certificates are signed statements about sets of public keys or programs. Each certificate has an issuer, specified by public key. Each certificate has one or more subjects: public keys or programs about which the statement is made. A certificate is valid only if accompanied by the issuer's signature, a special piece of data that only the issuer's private key can create. A signature is specific to that certificate and is evidence that the issuer actually made the statement in the certificate.

Example

Figure 3-2 shows a SPKI certificate that authorizes a class, followed by the signature of the certificate (adapted from [EFL+98]). We have included "//" comments for clarification; they are not part of the certificate or signature.

The `cert` contains the certificate material. The `issuer` specifies the public key of the principal that issued the certificate. The `subject` specifies the class being authorized by digest and URL. The `tag` specifies the type of authorization being granted; in this case, the issuer grants its `authority` to the class. The `not-after` and `not-before` parts are optional and bound the validity period of the `cert`.

The `signature` references the `cert` with the first hash and specifies the `issuer`'s public key in the second hash. The material that follows is the actual signature of the `cert` using the `issuer`'s private key.

Both the `cert` and `signature` are given in their human-readable form. They are structured as Lisp *S-expressions*: sequences grouped by matching parentheses where the type of each sequence is identified by its first element [Riv97]. S-expressions are designed to be simple to parse and easy to read and understand. For network transport and signing, S-expressions can be encoded into machine-readable canonical forms.

Certificate Chaining

Certificates can be *chained* by connecting a public key in the subject of one certificate to the issuer of another certificate. Chains of certificates are used to generate new statements by issuers on the chain about subjects in the chain. A sequence of certificates can be *checked* to determine what statements follow from that chain.

For example, say Alice issues a certificate granting Bob some authorization, and Bob issues a certificate granting that authorization to Carol. By chaining these two certificates, we can deduce that Alice authorizes Carol. Alice can control whether or not Bob can delegate her authority by setting or clearing a "propagation" bit on her certificate. Certificate chains may be arbitrarily long, but each link except the last one must have the propagation bit set.

SPKI defines a format for certificates that emphasizes usability and readability [EFL+00]. This helps TEP reduce the size of its trusted base by simplifying parsing and certificate-handling code. Since SPKI certificates require no central authority; individual principals can issue statements at will. This is in keeping with Jif's spirit of decentralization and eliminates the need for additional trusted authorities. The certificate chain discovery and checking algorithms are described in Appendix A.

3.2 Names

Names bind strings to sets of keys or programs. Principal names in Jif programs are bound to public keys in the security environment. TEP uses the environment to determine if a given principal name is bound to a given key, thus connecting principals in Jif to principals in the environment.

```
(cert                  // a certificate
 (issuer               // the issuer of this certificate
  (public-key          // is specified with a public key
   (rsa-pkcs1-md5      // the public key algorithm
    (e #11#)           // the RSA public exponent
    (n |ALNdAXftavTBG2zHV7BEV59gntNlxtJYqfWIi2kTcFIgIPSjKlHleyi9s
       5dDcQbVNMzjRjF+z8TrICEn9Msy0vXB00WYRtw/7aH2WAZx+x8erOWR+yn
       1CTRLS/68IWB6Wc1x8hiPycMbiICAbSYjHC/ghq2mwCZO7VQXJENzYr45|
     ))))              // the RSA modulus

 (subject              // the subject of the certificate
  (object-hash         // is an object, specified by hash
   (hash md5 |vN6ySKWE9K6T6cP9U5wntA==|  // the MD5 hash
    jar:http://www.webtax.com/webtax.jar!/COM/webtax/TaxPrep.class
    )))                // a URL for the object

 (tag authority)       // the type of authorization: "authority"
 (not-before "1999-12-31_23:59:59") // cert is valid only between
 (not-after "2000-01-01_00:00:00")  // these two times
)

(signature             // the signature of the certificate
 (hash md5 |54LeOBILOUpskE5xRTSmmA==|)  // hash of signed object
 (hash md5 |Ut9m14byPzdbCNZWdDjNQg==|)  // hash of signer's key
 |HU6ptoaEd7v4rTKBiRrpJBqDKWX9fBfLY/MeHyJRryS8iA34+nixf+8Yh/
 buBin9xgcu1lIZ3Gu9UPLnu5bSbiJGDXwKlOuhTRG+lolZWHaAd5YnqmV9h
 Khws7UM4KoenAhfouKshc8Wgb3RmMepi6t80Arcc6vIuAF4PCP+zxc=|
                       // the signature itself
)
```

Figure 3-2: SPKI Certificate Example

For example, the name "Alice" in a Jif program could be bound to a set of public keys, any one of which Alice might use. When TEP opens a channel that's readable only by Alice, it must make sure that the principal reading from the channel controls one of Alice's keys. TEP uses the security environment to check if the remote principal's public key belongs to the set of keys bound to "Alice". If so, TEP can release Alice's data to the remote principal. Section 4.3.5 describes this process in detail.

3.2.1 Linked Names

In SPKI, names resolution is completely decentralized. Each principal defines their own name space by issuing *name certificates*. Principals bind local names to the keys of principals they know directly. Principals can then resolve names not only in their own name space, but in the name spaces of other principals [Aba98].

For example, say Alice, using key K_A, binds the name "Bob" to key K_B and binds the name "Carol" to key K_C. We depict this as:

$$K_A \text{ Bob} \rightarrow K_B \tag{3.1}$$
$$K_A \text{ Carol} \rightarrow K_C \tag{3.2}$$

Suppose Bob, using key K_B, binds the name "DaveJones" to key K_D:

$$K_B \text{ DaveJones} \rightarrow K_D \tag{3.3}$$

Alice can take advantage of Bob's bindings by *linking* to Bob's name space:

$$K_A \text{ Dave} \rightarrow K_B \text{ DaveJones} \tag{3.4}$$

Alice now has a local binding for the name "Dave" that references Bob's binding for Dave Jones. She could have instead used her local name for Bob in the *extended name* "K_A Bob DaveJones":

$$K_A \text{ Dave} \rightarrow K_A \text{ Bob DaveJones} \tag{3.5}$$

We read this last line as, "K_A's Dave is bound to K_A's Bob's DaveJones". The name "K_A's Dave" is resolved to $\{K_D\}$ using both Alice's and Bob's name certificates. Note that non-existent bindings such as "K_C's Dave" resolve to the empty set.

3.2.2 Name Resolution

Names are defined by certificates, and certificates may be chained together to resolve extended names. Consider the name "K_A's Dave" and the set of certificates described above. If we wish to resolve this name into a set of keys, we must find a chain of certificates that demonstrates the binding. We see that both certificates (3.4) and (3.5) provide a binding for "K_A's Dave". Let's use (3.5).

This certificate binds the name "K_A's Dave" to the extended name "K_A's Bob's DaveJones". To resolve "K_A's Dave", we must now resolve "K_A's Bob's DaveJones". Certificate (3.1) binds "K_A's Bob" to K_B. We can now use K_B to resolve the next part of the name, "DaveJones". Certificate (3.3) binds "K_B's DaveJones" to K_D. This is our final result, since there are no other names left to resolve.

We could have instead used the certificate (3.4) in the first step and gotten the same result. In SPKI, when multiple certificates provide different bindings for the same name, the result is the set of all possible bindings. Suppose Alice had also issued this certificate:

$$K_A \text{ Dave} \rightarrow K_E \tag{3.6}$$

The name "K_A's Dave" is now bound to the set of keys $\{K_D, K_E\}$. This rule is applied recursively when resolving extended names. Suppose we add these certificates:

$$K_A \text{ Bob} \rightarrow K_F \tag{3.7}$$

$$K_F \text{ DaveJones} \rightarrow K_G \tag{3.8}$$

The name "K_A's Dave" is resolved directly into K_E by (3.6). It is also bound to the extended name "K_A's Bob's DaveJones" by (3.5). Certificates (3.1) and (3.7) bind "K_A's Bob" to $\{K_B, K_F\}$, and certificates (3.3) and (3.8) bind "K_B's DaveJones" to K_D and "K_B's DaveJones" to K_G, respectively. So the final set of keys bound to "K_A's Dave" is $\{K_E, K_D, K_G\}$.

Although this may seem confusing, both the chain discovery algorithm and the checking algorithm are simple recursive procedures. Since SPKI is fully decentralized, these algorithms can treat all principals uniformly – there are no special-case principals. Appendix A describes both algorithms in detail.

3.2.3 Names on TEP

TEP resolves names in Jif programs using SPKI's linked local name spaces. Since each SPKI name space must start at some public key, TEP requires all principal names start from TEP's own public key. TEP, using its public key K_T, issues bindings for "dns", "verisign", and other trusted name services. Jif programs then name principals starting from one of these names, such as "dns edu mit lcs ajmani" or "verisign microsoft billg". TEP could implement name resolution with an entirely different security environment, such as Secure DNS [DNS99] or QCM [GJ98, GJ99], without requiring changes to existing Jif programs.

3.3 Authorization

Authorization controls access to a principal's privileges. A principal may issue certificates granting its authority to principals or programs. The issuer may also delegate to its subjects, allowing subject principals to authorize others. Chains of authorization certificates can define larger sets of keys or objects authorized by the issuer.

3.3.1 Propagation

In SPKI, *authorization certificates* specify an *issuer* principal; a *subject* principal, name, or program; an authorization *tag* that identifies the type of authorization being granted; and a *propagation* bit which must be set for the subject to delegate the authority. Here are some examples of authorizations:

$$K_A \xrightarrow{read+} K_B \tag{3.9}$$

$$K_B \xrightarrow{read} K_B \text{ Carol} \tag{3.10}$$

$$K_C \xrightarrow{read+} K_D \tag{3.11}$$

$$K_D \xrightarrow{write} \texttt{WebTax.class} \tag{3.12}$$

Authorization (3.9) states that the principal who controls key K_A authorizes the principal who controls key K_B to *read* anything K_A can read. The "+" means that K_A has set the propagate bit, so K_B can pass this authorization to other principals. In (3.10), K_B does exactly that: K_B grants read authorization to the principal that K_B calls Carol. Note however, that K_B does not set the propagate bit, so Carol cannot grant K_B's authorization to other subjects. Also note that names transparently transfer all authorizations to their bound subjects.

Let's assume K_B's Carol is bound to K_C. In (3.11), K_C grants read access, with the right to delegate, to K_D. In (3.12), K_D grants write access to the program `WebTax.class` (in an actual certificate, the class is referenced by digest, as shown in Section 3.1.4).

Given the above set of certificates, we can determine what authorizations are granted to each principal. K_A grants read authorization to the set of principals $\{K_B, K_C\}$. Authorization does not propagate from K_A to K_D because K_B's Carol does not have the right to delegate the authority.

K_C grants read authorization to $\{K_D\}$. Read authorization does not propagate from K_C to `WebTax.class` through K_D because (3.12) grants *write* authorization, and no other certificates grant the class read authorization.

3.3.2 Authorizations on TEP

TEP uses two types of authorization for Jif: `actsfor` and `authority`. Principals grant the `actsfor` authorization to other principals to satisfy Jif's runtime actsfor tests, as described in Section 2.2.1. Principals grant the `authority` authorization to Jif classes to satisfy class authority declarations.

Acts-For Authorization

Since actsfor tests check authorization between two named Jif principals (the "actor" and the "actee"), TEP must resolve the actee name into a key and then check that that key authorizes the actor's name. The actor's *name* must receive the authorization (rather than a key bound to that name), because this allows the actee to restrict the authorization to a particular actor role.

For example, the test `actsfor(Manager, Alice)` is implemented by first resolving the name "Alice" (in TEP's local name space) into a set of keys. We must then find an authorization chain from any one of Alice's keys to the name "Manager", defined in TEP's name space. Consider the following set of certificates:

$$K_{TEP}\ \text{Alice} \rightarrow K_A \tag{3.13}$$
$$K_{TEP}\ \text{Alice} \rightarrow K_B \tag{3.14}$$
$$K_A \xrightarrow{actsfor} K_{TEP}\ \text{Manager} \tag{3.15}$$
$$K_{TEP}\ \text{Manager} \rightarrow K_M \tag{3.16}$$

The Environment Prover can prove that Manager actsfor Alice with the sequence of certificates (3.13), (3.15). The prover finds this chain by starting at TEP's definition for Alice and resolving it into the set of keys $\{K_A, K_B\}$ using (3.13) and (3.14). The prover then executes a depth-first search that traverses all the `actsfor` certificates reachable starting from either of Alice's keys.

If the prover finds a chain that leads to "Manager", it stops and returns the chain to the Environment Checker on TEP. If the prover exhausts all its certificates without finding such a chain, it reports a failure to the checker. The checker, upon receiving a chain of certificates, checks that the chain actually proves the desired property.

The prover *does not* resolve "Manager" into K_M using (3.16): Alice's intent is to authorize the current manager, not the particular public key bound to the name "Manager". If Alice authorized K_M, the principal that controls K_M could use Alice's authorization even when not acting as "Manager".

Program Authorization

Satisfying `authority` declarations is straightforward. The declaration `authority(Bob)` is implemented by first resolving the name "Bob" into a set of keys and then finding an authorization chain from one of Bob's keys to the declaring class. It is important that the class be authorized by digest, since digests uniquely identify a particular class bytecode. Since even small changes to a class can change information flow, principals must re-authorize classes whenever they are changed.

Consider this set of certificates:

$$K_{TEP}\ \text{Bob} \rightarrow K_B \tag{3.17}$$
$$K_B \xrightarrow{authority+} K_B\ \text{Gnu} \tag{3.18}$$
$$K_B\ \text{Gnu} \rightarrow K_G \tag{3.19}$$
$$K_G \xrightarrow{authority} \texttt{WebTax.class} \tag{3.20}$$

This sequence of certificates, in the order presented, proves that Bob authorizes the class `WebTax` (`WebTax.class` denotes the digest of the class). The prover discovers this sequence by first resolving "Bob" in TEP's name space. This yields the key K_B; the prover then searches for `authority` certificates issued by this key.

The second certificate authorizes the principal "Gnu", named in Bob's name space.

The prover must now resolve this name, so it looks for name certificates issued by K_B that bind the name "Gnu". The third certificate binds "Gnu" to the key K_G. Finally, the fourth certificate shows that K_G authorizes the class `WebTax`.

This example demonstrates how principals can defer program authorization to other principals by granting them the `authority` permission. This example also shows how the prover may need to resolve names in the middle of a search. Name resolution can be time consuming and can introduce branches in the authorization graph that slow down the prover. This effect can be reduced by caching intermediate authorizations and resolutions at the prover.

The design of these checks allows TEP to use alternate security environments to implement authorizations without requiring changes to existing Jif programs. However, Jif users will need to re-issue their authorizations if TEP changes the way it implements these checks. The next chapter describes how and when TEP invokes authorization checks.

Chapter 4

Design

This chapter describes the components, interfaces, and interactions that compose the Trusted Execution Platform. TEP separates trusted and untrusted components to make its trusted base as simple as possible. Section 4.1 presents TEP's trusted components and services, then Section 4.2 presents the untrusted ones. Section 4.3 details the interactions between these components.

4.1 Trusted Components

This section describes the *trusted* components of TEP. "Trusted" means that all principals are willing to expose their private data to these components and are willing to believe the output of these components without further verification. Compromise of these components can compromise the *safety* of the system; that is, compromise can cause classified information to be leaked to untrusted parties. We assume these components run together on a physically secure machine with trusted administrators.

4.1.1 Strong Encryption

To ensure the privacy of data on the network, communication channels are encrypted with symmetric-key ciphers. We assume symmetric key encryption is powerful enough that attackers cannot decrypt data before it loses its value. We also assume that public key encryption, private key signatures, and message authentication codes are strong.

Compromise of channel encryption can allow eavesdroppers to read private data from the network. Compromise of digital signatures or message authentication codes can allow attackers to undetectably alter private data, thus destroying the *integrity* of the data. TEP depends on these cryptographic mechanisms to provide privacy and integrity guarantees.

4.1.2 Process Isolation

Each program that runs on TEP resides in its own logical *process*: each process has its own memory, class namespace, and access to system resources. Since TEP is a shared

service, it must provide *process isolation* to ensure that no process can interfere with the memory or namespace of other processes. This is vital for protecting the privacy of data used in computation.

Since all processes run on the same physical machine, TEP must implement process isolation within its trusted computing base. TEP runs each process in a thread on TEP's own Java Virtual Machine. By running processes as threads, rather than as separate virtual machines, TEP improves performance, reduces system load, and simplifies process management. Since threads can use shared variables to communicate values and create covert channels, TEP must ensure that its application threads cannot share data.

The work described in this thesis does not fully implement the process model. Instead, we use Java language constructs, custom Java classloaders, and the Java security manager to provide memory isolation and to control access to system resources. The following sections describe these mechanisms and define how Jif programs access TEP's resources. More advanced systems for isolating processes in Java are described in Chapter 6.

Class Isolation

To separate the class namespaces of different applications, we have implemented a custom Java classloader for TEP applications. Since different applications can have classes with the same name, TEP must ensure that separate applications cannot load each other's classes. Otherwise, an application may attempt to load a class that does not link correctly with its own classes.

Each application that runs on TEP has its own private classloader, known as the *application classloader*. Each Java classloader provides a separate class namespace and static class data [LB98]. This ensures that separate applications running on TEP cannot access each other's classes or static data. Jif does not allow classes to have static variables, so the only static data in a class is the class object itself.

Multiple instances of the same application can run on TEP at the same time and can share a common application classloader. Since Jif classes cannot have static variables, these instances do not share any common class variables. However, these instances can communicate by synchronizing on the class object using class methods. To prevent this, Jif does not allow the use of the `synchronized` keyword.

Each application classloader loads code only from a single JAR file, so each client program must be packaged in a single JAR file. Since classes are stored as separate entries in the JAR file and Java requires that classes have the same name as their file, each application can have at most one class with a given name. This ensures that there is no ambiguity about which class to load for a given name.

Each application classloader defers to the system classloader and TEP's own classloader before loading classes from its JAR file, so foreign classes cannot override Java system classes or TEP's own classes. The application classloader also rejects any classes in the JAR that declare `package tep`: this prevents foreign classes from accessing TEP's package-private data.

Memory Isolation

Memory isolation ensures that one process cannot read from or write to the memory of another process. Although Jif labels protect the privacy of data in these programs, labels do not stop programs from using shared data to communicate. To prevent this, TEP does not allow separate application threads to share mutable variables. Threads may share read-only variables, such as an application classloader or system classes, but mutable variables are stored on each thread's local stack or in heap variables referenced from the thread's stack. If applications need to exchange data, they must do so through channels or persistent objects. The API for these functions is given later in this section.

All of TEP's implementation resides in classes in package `tep`, so client programs can only access TEP's `public` classes and methods. These methods and classes provide various services to the applications, but they do not reveal TEP's private data or data from other applications.

Controlling System Resources

Access to TEP's file system and the network is controlled using the Java security manager. Since the file system contains TEP's private data and implementation, foreign code must not be allowed to read or write files. Since the network is untrusted and open to eavesdroppers, all communications must be controlled by TEP to ensure privacy.

The Java security manager works by enforcing policies for various sets of classes. Each policy grants a set of permissions for accessing system resources to a particular set of classes. Java system classes are implicitly given all permissions, since they control access to the system directly [Jav98]. Jif programs running on TEP do not have direct access to the Java system classes; instead, we provide Jif versions of the standard library classes. These classes allow Jif programmers to use the standard Java APIs while providing appropriate information flow labels.

TEP's security manager grants TEP's own classes permission to access all resources but denies permission to all other classes.[1] If a foreign class attempts to access a restricted method or class, a runtime `AccessControlException` is thrown. It would be more efficient to make these checks statically; this is left for future work.

When foreign classes need to access the network, they must use TEP's privileged `TepChannel` class. This class ensures that private data written to the channel is protected from unauthorized principals (Section 4.3.5). Similarly, persistent objects must be stored using TEP's privileged `TepSealedObject` class. This class protects classified data when stored on untrusted servers (Section 4.3.6).

[1] Following the principle of least privilege, TEP should be granted only the permissions it requires. Determining this set of permissions is left for future work.

Controlling Thread Creation

Since Jif is a single-threaded language, TEP must prevent its applications from creating threads. Unfortunately, the Java security manager does not provide a way to control thread creation. Ideally, the `start()` method in class `java.lang.Thread` should throw a runtime exception if called from foreign code.[2]

Since this mechanism is not available, we assume that the static analysis performed by the Jif Checker rejects classes that attempt to create threads. We feel this assumption is reasonable and, in fact, provides a more efficient mechanism than runtime checks.

4.1.3 Jif Checker

All programs that run on TEP must pass static information flow checks. As described in Chapter 2, TEP uses a trusted Jif Checker to check the annotations on incoming bytecode. The annotations are generated by an untrusted Jif Prover and demonstrate that the code will not leak private data without authorization (see Section 4.2.3). Both the Jif Checker and Prover are described in Mathewson's thesis [Mat00].

4.1.4 Environment Checker

Programs on TEP must be able to establish their authority to declassify data. As described in Chapter 3, TEP uses a trusted Environment Checker to check authorization and authentication proofs. The proofs are generated by an untrusted Environment Prover and demonstrate that a given program or principal is authorized by another principal (Section 4.2.4).

This thesis uses the Simple Public Key Infrastructure (SPKI) to implement TEP's security environment [EFL+99]. However, TEP is not bound to SPKI for authentication and authorization services: any Checker that satisfies the interface described in the next section can be used by TEP.

Environment Checker API

Table 4.1 gives the API provided by the Environment Checker to TEP. This API does not define methods for initializing the checker, since different checker implementations might require different initialization data. We expect that checkers will initialize themselves from static configuration files. Our implementation of the Environment Checker requires TEP's public key for name resolution and the Environment Prover's host and port.

The `hasAuthority` routine returns *true* if the given key has authorized the specified class. If a name is given, the proof must first bind the name to a key, then show that the key authorizes the class. TEP uses this routine to test authority constraints on Jif classes (Section 4.3.3).

[2] Java provides a policy that prevents applications from adding new threads to a given thread group, but our experiments show that this does not stop applications from spawning threads.

```
class Checker {
    boolean hasAuthority(byte[] classHash, String issuerName);
    boolean hasAuthority(byte[] classHash, PublicKey issuer);
    boolean hasName(String subjectName, String issuerName);
    boolean hasName(PublicKey subject, String issuerName);
    boolean actsFor(String actorName, String acteeName);
    boolean actsFor(String actorName, PublicKey actee);
    boolean actsFor(PublicKey actor, String acteeName);
    boolean actsFor(PublicKey actor, PublicKey actee);
}
```

Table 4.1: Environment Checker's API for TEP

The `hasName` routine returns *true* if the issuer's name resolves to the subject key or name. TEP uses this routine to check whether a principal reading from an external channel belongs to the reader set in the channel's label (Section 4.3.5).

The `actsFor` routine returns *true* if the actee has authorized the actor to act on its behalf. TEP uses this routine to check actsFor tests in Jif programs (Section 4.3.4). Jif supports runtime principals that may be bound directly to public keys, so the checker must handle actsFor tests between keys and names.

If both the actor and actee are keys, the checker first checks whether the keys are the same. If so, it returns *true*, since actsfor is reflexive. Otherwise, it requires a proof that the actee authorizes the actor.

If the actor is a name and the actee is a key, the checker requires a proof that the actee authorizes the actor name.

If the actor is a key and the actee is a name, the checker first checks if the actee name resolves to the actor key. If so, it returns *true*, since names pass all authorizations implicitly. Otherwise, it requires a proof that the actee name resolves to a key that authorizes the actor.

If both the actor and actee are names, the checker first checks if the names are the same or if the actee name resolves to the actor name. If neither is true, it requires a proof that the actee name resolves to a key that authorizes the actor name.

These checks are implemented by the untrusted Environment Prover, discussed in Section 4.2.4. If a proof checks successfully, the checker caches the result. The expiration for the cache entry is set to the soonest expiration of all the certificates in the proof. If a proof fails the check or if the prover cannot find a proof for the request, the checker caches the failure for a short time (one minute in our implementation).

4.2 Untrusted Components

This section describes the untrusted components of the system. These components may behave arbitrarily without compromising safety, but must behave normally to

provide *liveness*. That is, untrusted components can cause the system to delay computation while waiting for an answer or reject authorization when it should have been granted.

Untrusted components must run on separate machines from the trusted base to reduce the risk of compromise. By moving complicated processing into the untrusted components, we reduce the size of the trusted base and increase reliability.

4.2.1 Network

The network is asynchronous, unreliable, and publically-visible. The network can drop, repeat, reorder, and modify messages. A given system may be online or offline or may switch between. Online hosts can become temporarily or permanently unreachable. The Domain Name Service is unsecure and IP addresses can be spoofed. Trusted systems must remain fail-safe when required services are unavailable.

4.2.2 Invokers, Providers, Readers, and Writers

Programs on TEP can interact with a variety of untrusted parties. Invokers contact TEP directly to request the execution of a method on a particular class. Providers create Jif applications and post them on the network in JAR files. Readers and Writers wait for connections from programs to read data from or write data to the program, respectively. Invokers themselves are Readers and Writers: they can communicate with the program over their channel to TEP.

One of the simplest systems for TEP requires one Invoker and one Provider: this yields the "Remote Procedure Call" (RPC) model of computation where the invoker provides data to the program and receives return values. Programs can add unbounded numbers of Readers and Writers (a party may be both), but always has exactly one Invoker and Provider. A single Invoker can request programs from different Providers on separate connections to TEP.

TEP need not trust Invokers, Readers, and Writers since TEP authenticates them and requires proofs that they can access classified data (Section 4.3.2). Providers need not be authenticated since program authorizations give the digest of the authorized class explicitly.

We can check an application's authenticity by comparing the digest of the application JAR file against a digest provided by the Invoker. Alternately, we can check a signature on the JAR file using a public key provided by the Invoker. These checks are not required to protect the privacy of data, but they ensure that TEP loads the desired program. Our implementation allows the invoker to check an *MD5* digest of the JAR file.

4.2.3 Jif Prover

The Jif Prover annotates bytecode with information flow controls required by the Jif Checker. The annotations demonstrate that the program will not leak classified information without authorization, as described in Chapter 2. The prover can be

untrusted since the checker will only accept classes that will protect the privacy of data.

4.2.4 Environment Prover

The Environment Prover is an untrusted service used by TEP to obtain authentication and name resolution proofs. The implementation of the prover depends on the specific name, key, and certificate management systems in the environment. As described in Chapter 3, our system uses SPKI to generate these proofs.

The prover resides on a separate machine from the TEP, allowing multiple TEPs to share a single prover and preventing attackers from compromising TEP via the prover. Even if attackers do compromise the prover, they cannot compromise TEP's safety. The prover must provide the signatures and verification keys required to validate each certificate, so an attacker cannot forge an authorization unless it controls the issuer's private key.

A separate prover is not technically necessary, as long as the checker satisfies the API given in Section 4.1.4. However, separating these two systems allows us to reduce the size of TEP's trusted base.

The remainder of this section describes the interface between the Environment Prover and Checker. This interface is specific to our particular implementation and defers as much work as possible to the prover.

Environment Prover API

Table 4.2 gives the API provided by the Environment Prover to the Environment Checker. This API can change with different implementations of the Prover and Checker and is not available to TEP directly.

The Checker sends a public key to the Prover as a SPKI *PublicKey*, which includes the key material and algorithm identifier. A principal name is sent as a SPKI *Name*, which includes the name as a string and the starting public key. Recall that since all names in Jif programs are based in TEP's name space, the starting public key for Jif names is always TEP's public key.

A class is sent as a SPKI *ObjectHash*, a wrapper around a digest of the class bytecode. The Prover returns each proof as a SPKI *Sequence*, a series of certificates and signatures as described in Chapter 3.

The `hasAuthority` routine returns a proof that the given issuer authorizes the given class. If the issuer is given by name, the proof first resolves the name into a key then shows that the key authorizes the class.

The `hasName` routine returns a proof that the given name is bound to the given subject. The subject is a SPKI *Subject*, which may be a *PublicKey*, *Name*, or *ObjectHash*.

The `actsFor` routine returns a proof that the "actee" authorizes the "actor" to act on the actee's behalf. If the actee is given by name, the name is first resolved into a key. However, if the actor is given by name, the proof ends with an authorization of the name itself, as described in Section 4.1.4.

```
Sequence hasAuthority(ObjectHash classHash, PublicKey issuer);
Sequence hasAuthority(ObjectHash classHash, Name issuerName);
Sequence hasName(Subject subject, Name issuerName);
Sequence actsFor(Name actorName, Name acteeName);
Sequence actsFor(PublicKey actor, Name acteeName);
Sequence actsFor(Name actorName, PublicKey actee);
Sequence actsFor(PublicKey actor, PublicKey actee);
```

Table 4.2: Environment Prover's API for Checker

All of these routines return *null* if they fail to find a proof for the request. Our implementation does not cache proofs at the prover, since we cache results at the checker. Caching proofs can improve prover performance and may be useful if the prover is shared by multiple systems.

4.2.5 Security Environment

The security environment is, at its core, a large set of certificates and a standard by which to interpret them. In this implementation, all the certificates in the system will be stored at the Environment Prover. The interpretation of certificates is based on the Simple Public Key Infrastructure (SPKI [EFL+98, Ell99, EFL+99, EFL+00]).

The system is based on the belief that a certificate signed by a particular private key is a statement made by the principal who controls that key. As long as certificates mention only keys, no additional trust is required. However, name resolution introduces additional trusted systems: programmers must take care to resolve names only through naming services they trust. For example, if DNS's private key is compromised by an attacker, the attacker can issue name certificates binding any DNS name to any key.

Revocation

An important mechanism in certificate systems is *revocation*, the ability to invalidate certificates that should not be trusted. When a principal's private key is compromised, certificates issued by that key should not be trusted. Although SPKI supports revocation with online checks, our implementation only supports certificate expiration. Expirations provide coarse-grained control over certificate validity but do not have the responsiveness or flexibility of online checks. Although we do not implement online checks, we will discuss the additional infrastructure and runtime mechanisms they require.

Online checks differ from certificate expirations in that they typically validate certificates only for a short time. This means that TEP must constantly revalidate each certificate used to satisfy name resolutions and authorizations. However, this also allows TEP to avoid requesting new name and authorization proofs whenever the

supporting certificates expire. Our implementation avoids the complexity of revalidating certificates online but suffers the performance impact of needing to regenerate expired proofs.

Long-lived programs running on TEP introduce a problem for authorization. If the authorization for a class in such a program is revoked or expires while the program is running, what should happen to the program? The correct behavior is to stop the program's execution, although this may leak a small amount of information. The main challenge with supporting this behavior is keeping track of what programs are affected when an authorization expires. Even more difficult is keeping track of the actsfor checks that each program depends on and killing a program if its authorization is revoked.

When a principal's private key is compromised, it must revoke all the certificates issued under the old key and change to a new key. More importantly, the principal must contact the issuers of any authorizations granted to the old key. Those issuers must revoke any certificates that authorized or bound names to the old key and reissue them to the new key. Names help simplify this process by adding a level of indirection to keys: if a principal is always referred to by name in the subject of certificates, that principal can change keys simply by changing the name binding.

4.3 Interactions

This section details the interactions between components of the system.

4.3.1 Program Distribution

TEP requires that programs it runs be packaged in Java Archive files (JARs) [Jav97]. A JAR contains all the classes required by an application except standard Java library classes and runtime support classes provided by TEP (Section 4.3). The application classes are linked with the system classes and TEP's classes by the application classloader (Section 4.1.2). When a new JAR is downloaded, TEP checks all the classes in the JAR before running any of that code. Our implementation caches previously-loaded and checked JARs to improve response time.

Including Proofs

An application JAR may contain SPKI authorization proofs that TEP can check and cache. By including proofs with their applications, providers can greatly decrease the time required to satisfy class authority declarations (Section 4.3.3) and runtime actsfor checks (Section 4.3.4).

Proofs included in a JAR are listed in the JAR's *manifest file* [Jav97]. The manifest is stored in the JAR entry `META-INF/MANIFEST.MF`. An example manifest file is given in Figure 4-1.

The first line of the manifest is required by the JAR standard and simply indicates the manifest version number. The remaining lines are divided into *sections* by blank lines. Each section has a name, given by a `Name` field.

```
Manifest-Version: 1.0

Name: proof : dns mit lcs bob : actsfor : dns webtax client
File: bob.actsfor.client.proof

Name: proof : COM.webtax.Tax : authority : dns webtax preparer
File: tax.authority.preparer.proof
```

Figure 4-1: JAR Manifest File with Proofs

Sections that describe proof entries have four-part names, where each part is separated by a colon. The first part of the name is simply the string `proof`, indicating that the entry describes an authorization proof. The second part names the subject of the authorization; the third part gives the authorization tag; and the last part gives the name of the issuer of the authorization.

For `actsfor` proofs, the subject is a Jif principal and the tag is `actsfor`. The first section of the manifest in Figure 4-1 specifies a proof that `dns mit lcs bob` acts for `dns webtax client`. `File` field indicates which entry in the JAR file contains the proof itself.

For `authority` proofs, the subject is a class contained in the JAR file. The second section of the manifest specifies a proof that `dns webtax preparer` authorizes the class `COM.webtax.Tax`. This class is stored in the JAR entry `COM/webtax/Tax.class`.

Whenever TEP loads a JAR file, it also loads the proofs contained in the JAR file. TEP sends each proof to the Environment Checker and, if the proof is correct, caches the result. For efficiency, TEP only attempts to check a proof whose result is not already cached.

We might want to extend this mechanism to handle name resolution proofs and proofs whose issuers or subjects are public keys. Our implementation does not do so, but we expect implementing this extension to be straightforward.

4.3.2 Secure Session Creation

This section describes how two parties establish a secure communication session over an untrusted network. We have developed our own non-standard protocol that avoids certificate exchange between parties. Instead, our protocol forces each party to prove which key it controls and allows each party to check the authorization of their peer's key locally.

This differs from standard protocols, such as SSL, that exchange authorization certificates during key exchange. Since our protocol has not been analyzed fully, future work should explore adapting TEP to use a more proven protocol. We also expect that better protocols can reduce the time required to establish a secure session, as described in Section 5.2.5.

A secure session guarantees these properties:

Privacy Eavesdroppers on the network cannot read session data.

Integrity Attackers on the network cannot change session data without detection.

Authentication Each party knows what private key their peer controls.

To achieve these properties, communicating parties must engage in a *mutual authentication* protocol. The protocol proves to each party what private key the other party controls, and allows parties to reject unrecognized peers. The parties also exchange secret *session keys* for encrypting session data. Finally, the parties include *message authentication codes* (MACs) with their messages to protect the integrity of their data.

The principals Alice (the client) and Bob (the server) run the protocol as follows:

1. Alice sends Bob her public key, a fresh nonce (a 64-bit random number), and her signature over the data:

   $P_A, R_A, S_A(P_A, R_A)$

2. Bob verifies Alice's signature and replies with his public key, a fresh nonce, a fresh secret key encrypted with Alice's public key, and his signature over *all* the data:

   $P_B, R_B, E_A(K_B), S_B(P_A, R_A, P_B, R_B, E_A(K_B))$

3. Alice verifies Bob's signature and decrypts his secret key. She then sends Bob a fresh secret key encrypted with Bob's public key and her signature over *all* the data:

   $E_B(K_A), S_A(P_A, R_A, P_B, R_B, E_A(K_B), E_B(K_A))$

4. Bob verifies Alice's signature and decrypts her secret key.

5. To protect the privacy of their data, Alice and Bob encrypt their output streams. Alice encrypts her output stream with K_A and decrypts her input stream with K_B. Bob encrypts his output stream with K_B and decrypts his input stream with K_A.

6. The protect the integrity of their data, Alice and Bob append a MAC to each message they send. Alice derives her output MAC key from K_A and her input MAC key from K_B. Bob derives his output MAC key from K_B and his input MAC key from K_A.

7. To defeat Man-In-The-Middle attacks, Alice and Bob check their peer's public key against a set of expected public keys. If their peer's key is not a member of the expected set, they close the session.

We discuss each of these steps in turn:

Step 1 provides Bob with Alice's public key and a fresh nonce. For now, Bob accepts the key and uses it to verify the accompanying signature. Notice that this

message could be a replay of an earlier instance of the protocol; this attack will be defeated by later steps in the protocol.

Step 2 provides Alice with Bob's public key, a fresh nonce, and his session key. The session key is encrypted with Alice's public key, so Alice can decrypt it with her private key. The signature is calculated over all the data from the first two steps, so it includes both nonces. This message defeats most replay attacks by Bob, since it is very improbable that those two nonces were used together in a previous instance of the protocol.

Step 3 provides Bob with Alice's session key. Again, the signature includes both nonces, so this message defeats most replay attacks by Alice. By step 4, both parties have each other's public key, session key, and evidence that the other controls the given key (using signatures). The signatures also protect the integrity of each message.

Step 5 protects data sent during the session from eavesdroppers. Symmetric-key encryption has good performance and is difficult to break, and so works well to protect the privacy of data. This implementation uses the *RC4* stream cipher with 128-bit keys.

Step 6 protects data sent during the session from attackers. It is possible for an attacker to overwrite a message in such a way that, when decrypted, it means something else. Alice and Bob protect their messages with MACs: MACs are message digests that include a shared secret (usually derived from the session keys), so they cannot be created by an attacker. MACs are included with each message before it is encrypted. If either party receives a message without a matching MAC, they discard the message. This implementation uses *HMAC-MD5*, a MAC algorithm based on the *MD5* message digest algorithm.

Step 7 protects the session from man-in-the-middle attacks. These attacks occur when an attacker intercepts the keys being sent during the first three steps and substitutes its own keys. The attacker can convince both parties that the protocol has completed successfully while gaining access to all data sent during the session. This attack is very hard to defeat without a shared secret or prior knowledge about the parties.

In TEP, most secure sessions are between parties that know something about one another. We assume TEP's public key is well known, so invokers that contact TEP can reject peers that do not use TEP's key. TEP does not have prior knowledge about the invoker, but Jif programs can check the identity of the invoker with a runtime actsfor test. For example, the test `actsfor(invoker, Bob)`, where `invoker` is the invoker's runtime principal, will succeed only if the invoker's key is bound to the name Bob or if the invoker acts for Bob.

When TEP contacts a principal to open a secure channel, the receiving party can check its peer's public key against its set of known TEP keys. TEP checks that the remote principal's key is bound to one of the readers in the channel label, so it is guaranteed that data is leaked only to authorized principals. Section 4.3.5 presents channel creation in detail.

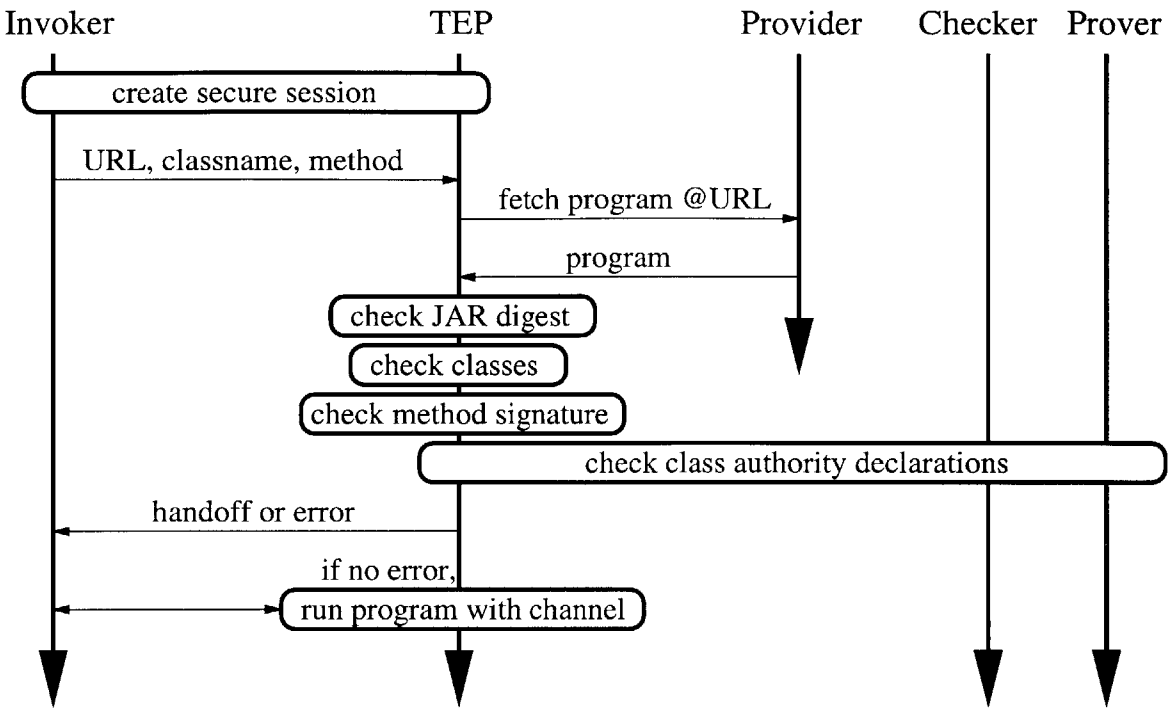

Figure 4-2: Invocation Protocol

4.3.3 Program Invocation

To invoke a method on TEP, a client must first establish a secure channel to TEP, as described in the previous section. The invoker sends TEP the URL for an application JAR file, an optional digest for the expected JAR, a class name, and the name of a method on that class. TEP fetches the JAR from the URL, checks it against the digest, checks its classes with the Jif Checker, checks the authority declarations for each class, and checks that the requested method meets the requirements given in the following sections.

If all the requirements and constraints are satisfied, TEP sends a "handoff" message to the invoker to signal that all the checks have passed and the channel is now connected directly to the program. TEP runs the method, passing it the channel to the invoker. If any of the checks fail, TEP reports the error to the client and closes the session. Figure 4-2 gives a graphical representation of the invocation protocol.

Class Authority Declarations

Class authority declarations specify the principals whose authority the class uses to declassify data, as described in Chapter 2. TEP checks the authority declarations for each class in an application JAR file before running the application. Since authority is a static property of the class, all authority declarations must be satisfied before runtime. If, instead, we deferred these checks until each class was loaded by the

```
Manifest-Version: 1.0
Authority: dns webtax preparer

Name: public static void WebTax.calculate(tep.JifPrincipal)
Authority: dns webtax preparer
Caller:
ActsFor: dns mit bob, dns mit alice; dns mit alice, dns mit carol
```

Figure 4-3: Jif Class Manifest File

application, failed checks could leak information.

Our implementation uses a static class field called `JIF_MANIFEST` to store class and method constraints. Recall that Jif does not allow static class fields, so this field is inaccessible to the Jif programmer and must be generated by the Jif Prover. `JIF_MANIFEST` is a Java manifest file stored in a field of type `java.lang.String`. An example of a class manifest file is given in Figure 4-3.

The top-level `Authority` field in the class manifest gives a comma-separated list of the principals whose authority is required by the class. In Figure 4-3, the class requires the authority of `dns webtax preparer`. TEP checks that each principal in the list has granted its authority to the digest of the class using the Environment Checker's `hasAuthority` routine. If any of these checks fails, the class is rejected and the error is reported to the application invoker.

After the class `Authority` field, there may appear a number of sections corresponding to the methods in the class. Each section is named using the full method signature as defined by `java.lang.reflect.Method.toString()` [Jav99]. Each method section may specify `Authority`, `Caller`, and `ActsFor` fields that correspond to the method authority, caller, and actsfor constraints.

The first two fields give comma-separated lists of principals; the `ActsFor` files lists pairs of principals separated by semicolons. In Figure 4-3, the method `calculateTax` requires the authority of `dns webtax preparer` and requires that Bob actsfor Alice and Alice actsfor Carol.

Method Constraints

Any method that TEP invokes directly must have a specific signature, as described below. This signature ensures that no information is leaked when the method terminates and that the method can communicate with its invoker. This signature also pushes application-specific processing out of trusted code, reducing the size of TEP's trusted base.

Invocable methods must:

1. Be public and static.

2. Return void.

3. Accept exactly two arguments: A runtime principal `invoker` (labelled {}) that wraps the invoker's public key and a channel (labelled {}) that reads and writes data labelled {`invoker: invoker`}. Notice that the label on the `channel` variable is not the same as the label for the data it reads and writes.

4. Have begin and end labels {}: The *begin-label* specifies a restriction on the program counter at the invocation point of the method; the label {} means the method gains no information from its caller. The *end-label* specifies information leaked when the method terminates; the label {} means the method may not leak any information by terminating.

5. Have no caller or actsfor constraints.

6. Reside in a class that has no principal or label parameters. Such parameters allow a single class to work with different principals and labels, but TEP does not support this mechanism for top-level methods.

So, an invocable method must have this signature:

```
public static void ANY_NAME{}(
    principal{} invoker,
    TepChannel[{invoker:invoker},{invoker:invoker}]{} channel
    ){} where authority(...)
```

Requirement 1 ensures that the method is accessible to TEP and does not require an instance of the class. TEP does not support top-level instance methods because it does not preserve class instances between invocations. If a program needs to store data between invocations, it must use the persistent data mechanism described in Section 4.3.6.

Requirements 2 and 3 allow TEP to avoid parsing and marshaling complex data structures. TEP provides a channel to the invoker of the class, so the method can handle arguments and return values. This gives Jif programmers full control over the format of the method parameters, and removes that code from TEP's trusted base. The `TepChannel` class is described in detail in Section 4.3.5.

The label on the data that can pass through the channel is {`invoker: invoker`}, meaning that only the invoker can read that data. If the invoker needs to provide data with a different label to the program, it can either write a program to relabel the data or require that the program open a second channel. For the first option, the invoker must write a program that relabels data read from the channel and invokes the desired method directly. If the relabelling requires declassification, this program must run with the invoker's authority. The second option is simpler but less efficient; the original program must open a second channel to the invoker and request data with a different label.

Requirement 4 ensures that no classified information is leaked when the method starts or terminates. Jif keeps track not only of information flow through data but also implicit transfers through program control flow. The label {} on the begin and

end of the method require that the starting and ending points of the method reveal no classified information.

Top-level methods may not throw exceptions, since TEP does not report any information about method termination to the caller. If the method needs to report an exception, it should send a message directly to the invoker over the given channel. The method may throw FatalError, Jif's only unchecked exception, but it is caught by TEP and ignored.

Requirement 5 ensures that the method does not inherit any caller authority or actsfor results from TEP. The method can gain static authority only through an authority constraint. The constraint can only list principals who also appear in the authority declaration of the method's class.

Top-level methods may not have caller or actsfor constraints, since these constraints allow the method to inherit authority and actsfor results from the calling code. If a principal needs to call a method with a caller constraint, the principal must write a wrapper method that declares the principal's authority. Similarly, if a principal needs to call a method with an actsfor constraint, the principal must write a wrapper method that makes required actsfor test at runtime.

Requirement 6 ensures that TEP does not have to determine parameters for the top-level class at runtime. Since class parameters have no runtime representation, it does not make sense for TEP to parameterize the class.

Although these requirements seem restrictive, we can simulate most common computation models within these constraints. Parsing arguments within a method requires no more work than parsing outside the method, but it allows the programmer, rather than TEP, to determine the format for the arguments. This flexibility not only aids the programmer, but also reduces the size of TEP's trusted base.

We can implement methods that parse arguments, marshal return values, report exceptions, determine class parameters, and instantiate classes. Class instances can be saved to and loaded from persistent objects. By packaging such methods in a library, we can provide remote procedure call semantics to the programmer while freeing TEP from these responsibilities.

4.3.4 Runtime Authorization

Jif programs can increase their authority at runtime with actsfor statements, as described in Chapter 2. TEP implements actsfor statements by translating them into calls to a `TepRuntime` object. The `TepRuntime` class encapsulates the code required for checking authorization between two principals and contains data required by other privileged TEP classes. Its API is given in Table 4.3.

Each application running on TEP has a reference to its own instance of the `TepRuntime` class. Each instance contains a reference to TEP's public and private keys, a reference to the Environment Checker, a connection to the Environment Prover. This connection is used to issue requests for authorization proofs at runtime. `TepRuntime` objects share a cache so they can take advantage of common authorization results.

Jif actsfor statements are translated into calls to the `actsfor` method on the application's `TepRuntime` object. The block following an actsfor statement is executed

```
public class TepRuntime {
    // public methods accessible by Jif classes
    public static TepRuntime get();
    public boolean actsFor(JifPrincipal actor, JifPrincipal actee);

    // package-private methods accessible only by TEP's classes
    static void set(TepRuntime runtime);
    PublicKey getPublicKey();
    PrivateKey getPrivateKey();
    boolean sameAs(PublicKey key, JifPrincipal principal);
}
```

Table 4.3: TEP's Runtime API

if the check succeeds, otherwise, the optional `else` block is executed. An example of this translation is given in Figure 4-4.

`TepRuntime`'s static `get` routine returns the instance of the runtime associated with the current application thread. If `get` is called by a thread that is not running on TEP, it throws the unchecked exception `JifFatalError` and forces the calling Jif application to abort. `JifFatalError` is translated from Jif's only unchecked exception, FatalError.

The `actsFor` routine returns *true* if the "actor" (the first principal) is authorized by the "actee" (the second principal). This routine is implemented by calling the appropriate `actsfor` routine on the Environment Checker (Section 4.1.4).

The `JifPrincipal` class is an abstract superclass of the two classes `JifName` and `JifKey`. These classes represent principal names (strings) and runtime principals (public keys), respectively. These classes are invisible to the Jif programmer, who uses principal names and variables directly.

The remaining `TepRuntime` routines are privileged and only accessible by TEP's classes. The `set` routine allows TEP to set the `TepRuntime` object for the current application thread. The `getPublicKey` and `getPrivateKey` routines return TEP's public and private keys respectively, for use in establishing secure channels.

The `sameAs` routine returns *true* if the given key corresponds to the given `JifPrincipal`. If the principal is a `JifName`, the routine checks whether the name is bound to the given key using the Environment Checker's `hasName` routine. If the principal is a `JifKey`, this routine simply compares the keys directly. The `sameAs` routine is used in channel creation to determine if the remote principal is one of the readers in the channel label, as described in the next section.

4.3.5 Channels

Jif is designed to allow multiple parties to share data in computation, so Jif programs must be able to exchange data with external principals. This functionality is provided

```
// Given Bob's authority,

actsfor(Bob, Alice) {
    // use Alice's authority
} else {
    throw new AccessDenied("Bob must act for Alice");
}

// translates to...

if (TepRuntime.get().actsFor(new JifName("Bob"),
                             new JifName("Alice"))) {
    // use Alice's authority
} else {
   throw new AccessDenied("Bob must act for Alice");
}
```

Figure 4-4: Translating Jif's actsfor statement

by the `TepChannel` class.

The `TepChannel` class implements a bi-directional, secure network channel that supports different privacy labels on the input and output streams. Although some channels are implicitly uni-directional, TEP requires that channels be bi-directional to support the secure session protocol presented in Section 4.3.2. We expect that input and output devices such as keyboards, mice, displays, and printers will communicate with applications on TEP through these secure network connections.

Jif Channel API

The Jif API for `TepChannel` is given in Table 4.4. The `TepChannel` class is parameterized over two labels, `IN` and `OUT`. Data read from the channel's input stream may only be assigned to variables whose labels are equal to or more restrictive than `IN`. This label protects the privacy of data written by the remote principal to the program. Data written to the channel's output stream may only come from variables whose label is equal to or less restrictive than `OUT`. This label protects the privacy of data written by the program to the remote principal.

The `TepChannel` constructor requires that the given host and port have the label {}, because the host and port must be exposed to the untrusted network. This means that any code that creates a channel must remove all privacy policies from the host and port using declassification. Channel creation itself is a publically-visible event and can leak data through an implicit flow (Section 2.1.5). To prevent this, we could require that the *begin label* of the constructor be {}. This would ensure that no information is leaked when the channel is created, but it may be too restrictive for

```
class TepChannel[label IN, label OUT] {
    TepChannel[IN, OUT](String{} host, int{} port)
        throws (AccessControlException,    // illegal reader
                GeneralSecurityException,  // cryptography error
                IOException);              // input/output error
    jif.io.InputStream[IN] getInputStream();
    jif.io.OutputStream[OUT] getOutputStream();
}
```

Table 4.4: Jif's Channel API

some programs.

After creating a connection, the `TepChannel` constructor runs the channel creation protocol described in the next section. The constructor throws an `AccessControlException` if the remote principal does not belong to the reader set of the label `OUT`. This protects programs from revealing classified data to unauthorized principals. The constructor throws a `GeneralSecurityException` if there is an error with the secure session protocol and throws an `IOException` if there is an error communicating with the network. The constructor may also throw an unchecked `JifFatalError` if it cannot get an instance of `TepRuntime`. The parentheses around the checked exception list are required by Jif to disambiguate its syntax.

The `getInputStream` and `getOutputStream` routines provide access to the input and output streams of the channel. We chose to build channels around the standard Java `InputStream` and `OutputStream` classes so that programmers can use familiar library classes for stream manipulation and object serialization. Jif programmers will need to use Jif versions of the stream classes that restrict the labels of data that can be read from or written to a stream. In this thesis, we assume these classes are available in the `jif.io` package.

TEP Channel API

When the Jif API is translated into Java for TEP, the `IN` and `OUT` labels must be preserved for use at runtime. Table 4.5 gives the `TepChannel` API in Java. The constructor accepts the `IN` and `OUT` labels as two additional arguments of type `JifLabel`.

The `JifLabel` class is a runtime representation of a Jif label. Each `JifLabel` contains a set of objects of type `JifPolicy`. Each `JifPolicy` has an owner and a set of readers of type `JifPrincipal`. The reader set for a `JifLabel` is calculated as the intersection of the reader sets for each `JifPolicy`. `JifLabel` also has a `relabelsTo` method that returns *true* if the label is less restrictive than its argument.

Channel Creation Protocol

Figure 4-5 gives a graphical representation of the channel creation protocol. The first step in the protocol is establishing a secure connection to the requested host and port.

```
class TepChannel {
    TepChannel(JifLabel in, JifLabel out, String host, int port)
        throws AccessControlException,
               GeneralSecurityException,
               IOException;
    InputStream getInputStream();
    OutputStream getOutputStream();
}
```

Table 4.5: TEP's Channel API

TEP then checks whether the remote principal is one of the readers in the channel's `OUT` label.

This check implements the semantics of a *trusted reader*, since the reader need not enforce the policies on data sent over the channel. For example, data labelled {`Bob: Alice`} can be written to a channel connected to Alice, but Alice can then disseminate the data arbitrarily. Bob must trust that Alice will honor his policy. We discuss alternate output channel semantics in Section 6.2.1.

For each reader in the channel's `OUT` label, TEP calls `TepRuntime`'s `sameAs` routine on the peer's public key, K_P, and the reader. The `sameAs` routine uses the Environment Checker's `hasName` routine to determine if the reader's name is bound to K_P. If the peer is not an allowed reader, the protocol aborts and throws an `AccessControlException`.

If the reader check succeeds, TEP sends the channel's `IN` label and `OUT` label to the remote principal. The `IN` label is provided to allow the peer to check the privacy policies on data TEP reads from the channel. The `OUT` label is provided to allow the peer to preserve the privacy policies on data it reads from the channel.

If the remote principal does not agree that the `IN` label adequately protects its privacy, it closes the channel and TEP aborts the protocol with an `IOException`. If the remote principal approves the `IN` label, both parties may start using the channel.

Protecting Privacy and Integrity

The channel protects privacy by encrypting and decrypting data using the *RC4* stream cipher. The channel protects integrity by calculating MACs for data sent over the streams. Since MACs must be calculated for individual messages, rather than stream, we automatically break the stream data into blocks and insert a MAC for each block.

On output, we create a new block whenever the stream is flushed or the stream buffer reaches a maximum size. Each block is prefixed by its length, appended with its MAC, and encrypted with the output session key. On input, data is decrypted with the session input key, read into blocks, and checked against the accompanying MACs. If a MAC check fails, an `IOException` is thrown. These mechanisms protect

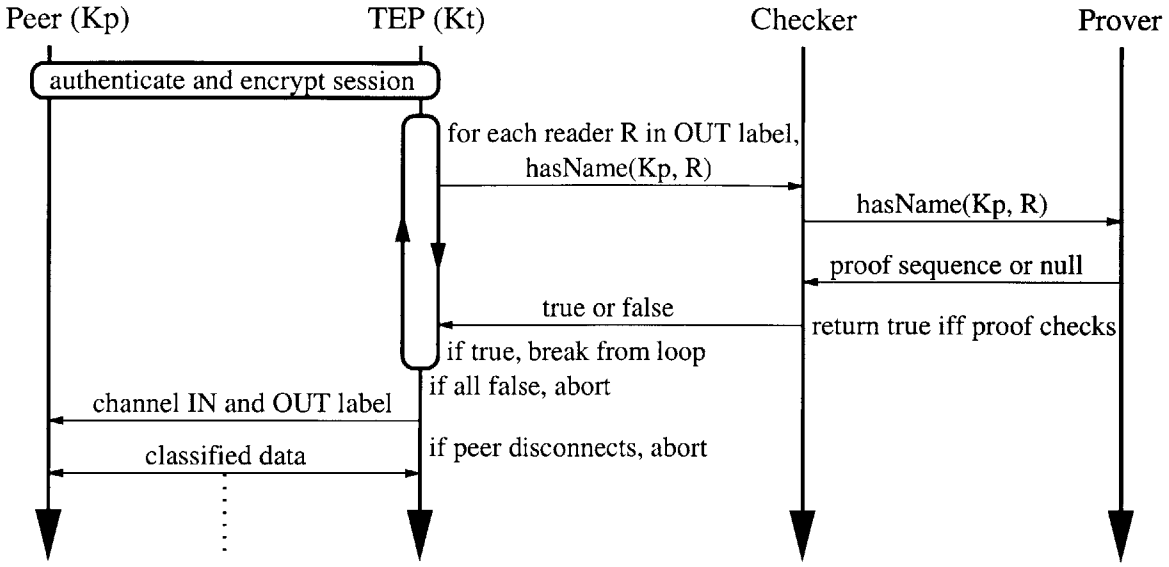

Figure 4-5: Channel Creation Protocol

both the privacy and the integrity of data sent over channels while providing the programmer with the standard Java stream abstractions.

4.3.6 Persistent Objects

Many applications require the ability to remember information between invocations. However, TEP does not allow programs to store persistent data on the trusted platform, since this opens the system to denial-of-service attacks.

Instead, TEP can cryptographically "seal" objects to declassify them for storage on other systems. TEP uses its authority to declassify the data and simultaneously encrypts the data to preserve its privacy [Aba97]. The program can then store the data on any server. When the program retrieves persistent data, TEP decrypts it and restores the label. If the program attempts to restore the object using a label less restrictive than the original, TEP throws an `AccessControlException`.

Persistent Object API

A persistent object can be created in two ways: it can be constructed from a labelled object or it can be read from an input channel. Both forms must ensure the original label of the object is preserved. TEP implements these objects with the `TepSealedObject` class, whose API is presented in Table 4.6.

To construct a persistent object from a labelled object, the programmer must parameterize the `TepSealedObject` with the same label as the data it stores: {L}. The constructor accepts any `Serializable` object with a label equal to or less restrictive than {L} and creates a declassified object. The constructor throws an exception if

```
class TepSealedObject[label L] {
    TepSealedObject[L](Serializable{L} object)
        throws (GeneralSecurityException,  // cryptography error
                IOException);              // input/output error
    void writeTo(DataOutput output)
        throws (IOException);              // input/output error
    TepSealedObject[L](DataInput input)
        throws (AccessControlException,    // label mismatch
                GeneralSecurityException,  // cryptography error
                IOException,               // input/output error
                ClassNotFoundException);   // unknown class type
    Serializable{L} getObject();
}
```

Table 4.6: Jif's Persistent Object API

there is an error serializing or encrypting the object.

The program can write a `TepSealedObject` to an output stream with the `writeTo` method and can read an object from an input stream with the second constructor.[3] The stream constructor throws an `AccessControlException` if its label is less restrictive than the label of the object on the stream. Jif programs can retrieve the original object from the sealed object with the `getObject` method.

As with channels, the Jif persistent object API is translated into a Java API for TEP. This API is given in Table 4.7. The translation changes the class label parameters into runtime label arguments, so that TEP can store and check the labels at runtime.

Serialized Form

The serialized form of a `TepSealedObject` is a 4-tuple:

$$\langle E_{TEP}(K),\ E_K(L),\ E_K(O),\ S_{TEP}(E_{TEP}(K), E_K(L), E_K(O))\rangle$$

$E_{TEP}(K)$ is a secret key, K, encrypted with TEP's public key. K is used to encrypt the object label, L, in $E_K(L)$ and the object itself, O, in $E_K(O)$. Finally, TEP signs the encrypted data in $S_{TEP}(\ldots)$. The signature protects the integrity of the object, while the encryption protects its privacy.

[3] TEP does not use Java's built-in serialization mechanism because it cannot support the necessary checks.

```
class TepSealedObject {
    TepSealedObject(JifLabel label, Serializable object);
        throws GeneralSecurityException,
               IOException;
    TepSealedObject(JifLabel label, DataInput input);
        throws AccessControlException,
               GeneralSecurityException,
               IOException,
               ClassNotFoundException;
    Serializable getObject();
}
```

Table 4.7: TEP's Persistent Object API

External Storage

Systems outside of TEP cannot use the `TepSealedObject` class, since it requires access to TEP's private key. Instead, these systems use the `SealedObject` class. Like `TepSealedObject`, `SealedObject` has a constructor that reads from an input stream and a `writeTo` method that writes to an output stream. This allows external systems to store, index, and retrieve objects for Jif programs while protecting the privacy of the sealed data.

4.4 Examples

This section presents a few example applications. The first two applications show how Jif's information flow controls interact with TEP's runtime mechanisms to allow programs to work with classified data. The latter two describe how TEP's mechanisms can be applied on the server-side and client-side to protect privacy.

4.4.1 Tax Preparer

This example returns to the application presented in the first two chapters: a tax preparation service. WebTax, the application provider, implements the service with the `WebTax` class, given in Figure 4-6. The application runs on TEP when a client invokes the `calculateTax` method on the `WebTax` class.

The `WebTax` class has an `authority` declaration that specifies the principal "dns com webtax preparer". This means methods in this class can run with the preparer's authority and can declassify the preparer's data. The class has two methods, `calculateTax` and `getTaxRate`.

The `calculateTax` method has the signature specified in Section 4.3.3 and claims the authority of "dns com webtax preparer". This method implements the tax preparation service itself with these three steps:

1. The method reads the client's `income` as a double-precision number through the provided channel.

2. The method reads the current `taxRate` from the WebTax server using `getTaxRate` and uses it to calculate the client's `taxAmount`.

3. If the method gets an `IOException` or a `GeneralSecurityException`, it reports the exception to the client and quits.

4. Otherwise, the method declassifies the `taxAmount` to remove WebTax's policy and sends the result back to the client.

This final step requires the authority of "dns com webtax preparer" to declassify `taxAmount`, since the original label on `taxAmount` makes is unreadable by the invoker. This declassification represents a leak of the preparer's information. In this example, the client can reconstruct the preparer's data from the final result, so this leak might be unacceptable. However, most calculations are complex enough that declassifying the final result leaks a negligible amount of data.

The `getTaxRate` method opens a channel to the WebTax server, "data.webtax.com", and reads the `taxRate` as a double-precision number. This method does not claim the authority of the preparer since it does not declassify data. This method also takes advantage of Jif's implicit label polymorphism to avoid explicitly labelling its return value.

Clients use the tax preparation service by connecting to TEP and requesting that it execute the `calculateTax` method. TEP runs the method, passing it a channel to the client and the client's runtime principal. The client then sends its income data over the channel and waits for the result. Jif's information flow controls protect the privacy of both the client's and the preparer's data.

4.4.2 Auction/Election

Auctions and elections are computations that combine data from a number of sources. An auction calculates a maximum bid among the bids submitted by the bidders. An election determines which candidate from a set of candidates receives the most votes submitted by voters. Since these applications are similar, we will focus our discussion on the auction.

Currently, an auction uses a single server to keep track of the bids and report the result when the auction is done. Bidders trust the server to record their bids and calculate the high bid correctly. We would like to move this trust to TEP instead and, additionally, protect the privacy of the bidders.

The ideal model would be to run the auction server as a TEP application. Unfortunately, programs running on TEP cannot accept connections, so bidders would not be able to submit their bids to the server. We discuss extending TEP to support incoming connections in Section 6.2.1.

Another model is to use a TEP application to submit bids to an untrusted server. TEP adds a level of indirection between the bidder and the server and can allow the

```
public class WebTax authority (dns com webtax preparer) {
    public static void calculateTax{}(
        TepChannel[{invoker:invoker},{invoker:invoker}]{} channel,
        principal{} invoker){}
        where authority (dns com webtax preparer)
    {
        try {
            // read the invoker's income amount
            jif.io.DataInputStream[{invoker:invoker}] input =
                new jif.io.DataInputStream[{invoker:invoker}](
                    channel.getInputStream());
            double{invoker:invoker} income = input.readDouble();

            // read the tax rate from WebTax and do the calculation
            double{dns com webtax preparer:} taxRate = getTaxRate();
            double{invoker:invoker; dns com webtax preparer:}
                taxAmount = income * taxRate;

            // send the declassified result to the invoker
            jif.io.DataOutputStream[{invoker:invoker}] output =
                new jif.io.DataOutputStream[{invoker:invoker}](
                    channel.getOutputStream());
            output.writeDouble(declassify(taxAmount,
                                          {invoker:invoker}));
            output.flush();
        } catch (IOException e) {/* report to client */}
        catch (GeneralSecurityException e) {/* report to client */}
    }
    private static double getTaxRate()
        throws (IOException, GeneralSecurityException)
    {
        // open a channel to the WebTax data server
        TepChannel channel = new TepChannel
            [{dns com webtax preparer:}, {dns com webtax preparer:}]
            ("data.webtax.com", WEBTAX_PORT);

        // read the tax rate from the server
        jif.io.DataInputStream[{dns com webtax preparer:}] input =
            new jif.io.DataInputStream[{dns com webtax preparer:}]
                (channel.getInputStream());
        return input.readDouble();
    }
}
```

Figure 4-6: Tax Preparer Jif Code

bidder to bid anonymously. However, this model does not stop the untrusted server from calculating the maximum bid incorrectly.

To ensure the calculation is done correctly, bidders invoke applications on TEP to submit their bids and calculate the high bid. TEP stores the bids in a sealed array stored on an untrusted server. When a bidder submits a bid, TEP reads the array from the server and appends the new bid with information that identifies the bidder. When a bidder or the auctioneer wants to know the high bid, TEP reads the array, unseals it, and calculates the maximum bid in the array. TEP also reveals the identity of the high bidder to allow that bidder to claim the item up for auction.

TEP must store the bids in a single array, rather than separate sealed objects, to ensure that the untrusted server cannot remove bids without destroying the entire auction. The bidding application must also ensure that other applications on TEP cannot read or overwrite the bid array. These concerns deal with the *integrity* of the auction data – restricting the principals and programs that can write the data. We discuss enhancements that protect integrity in Section 6.2.

4.4.3 Bank Accounts/Medical Database

Both banks and hospitals deal with classified data from a number of clients. Instead of combining this data in computation, these institutions need to ensure that data does not leak between client records. Myers' label model can protect programs from such mistakes by revealing places where data inherits too many policies and must be declassified.

These systems do not need to load code dynamically and often need to reveal client data to highly-authorized parties, such as doctors and bank executives. So, these applications would run better on servers local to the banks and hospitals than on TEP. However, these application *do* need to provide remote access to their databases, which requires establishing secure channels, checking authorizations, and protecting persistent data. These mechanisms are all part of TEP, so institutions that need to provide the same guarantees can incorporate much of the work described in this thesis.

4.4.4 Browser Applet

Perhaps the most common example of dynamically-loaded, untrusted code is the browser applet. Web browsers download bytecode to the client machine and execute it locally. Current protection mechanisms, including sandboxing and the Java policy mechanism, are often too coarse-grained and restrictive to protect privacy while allowing useful information sharing. In contrast, Myers' label system provides fine-grained information flow tracking and allows users to control when their information is leaked to other parties.

As we discussed in Chapter 1, client-side computation compromises the privacy of the service provider's data. When this is acceptable to the provider, TEP is unnecessary. However, TEP's mechanisms are useful for protecting the client from accidental

information leaks and for checking whether the provider is authorized to access classified data. The decentralized authorization model described in this thesis allows clients and providers to manage authorizations without trusting other organizations, such as VeriSign.

Chapter 5

Analysis

This chapter discusses TEP's implementation and performance. The primary goal of our work is to create a practical system that supports multiparty computation, so we have developed an implementation that allows us to evaluate the power, usability, and performance of our design. Section 5.1 describes TEP's implementation and provides information about its distribution. Section 5.2 presents performance results for several test applications and suggests ways to improve system performance.

5.1 Implementation

We have developed an implementation of TEP that supports the design described in this thesis. The implementation reveals the complexity required to protect private data throughout a system and allows us to evaluate the techniques we use for reducing the size of TEP's trusted base.

5.1.1 Distribution

TEP is implemented in Java and is available online at:

`http://www.pmg.lcs.mit.edu/~ajmani/tep/`

This distribution provides the code required to run the TEP server and clients, an Environment Prover server, and a "data" server that accepts connections from programs running on TEP. The implementation uses a number of third-party packages to implement various features:

SDSI/SPKI This package provides the certificate and public key management tools required by TEP. It was developed as part of TEP's implementation effort and is available at:

`http://www.pmg.lcs.mit.edu/~ajmani/sdsi/`

Jif Compiler and Libraries This package allows programmers to develop programs in Jif and is currently being implemented at Cornell University.

Component	Lines of Java	Trusted Lines	% Reduction
TEP Client/Servers	4600	4200	9%
SPKI Prover/Checker	6000	5000	17%
Jif Prover/Checker	†	‡	
Cryptography	2000	2000	0%
Total	12600 + †	11200 + ‡	11%

†,‡These quantities unavailable at this writing.

Table 5.1: Implementation Size

Jif Prover and Checker This package allows programmers to annotate their program bytecodes automatically and allows TEP to check the annotations on incoming programs. This package is being implemented at MIT.

Open JCE This package provides some the cryptographic algorithms used by TEP.

Cryptix JCE This package provides some of the cryptographic algorithms used by SDSI/SPKI and TEP.

The current distribution provides example classes for users who wish to develop applications for TEP. As the packages under development become available, TEP will be updated to support them. TEP's final APIs may differ from those presented in this thesis, but we expect to support all the functionality described herein.

5.1.2 Complexity

The number of packages used by TEP suggests that protecting privacy while supporting practical programs requires significant complexity. However, this observation can be misleading, since TEP uses only certain parts of each package in its trusted base. Table 5.1 lists the amount of code TEP uses from each package.

The second column in the table gives the number of lines used to implement both the trusted and untrusted parts of the corresponding packages. The third column gives the subset of those lines used in the trusted base. The cryptography entry in the table omits lines for algorithms that are not used by the system. We have not included the Java libraries, since we were unable to estimate how many lines are used by the system. The numbers are approximate and include comment and empty lines.

The table shows that separating untrusted components from trusted ones reduces the size of the trusted base by over ten percent. The largest reduction results from separating the SPKI Environment Prover from the Environment Checker, reducing the size of that package by one sixth. We expect the separation of the Jif Prover and Checker to provide similar benefits.

5.2 Performance

We have measured TEP's performance on several test applications and present the results in this section. We find that the vast majority of application time is spent doing the cryptography for establishing secure channels, checking authorizations, and creating persistent objects. We detail these findings and suggest optimizations for improving application performance.

5.2.1 Testing Platform

Our tests use four machines:

TEP This machine runs the Trusted Execution Platform itself. The Jif application JAR files are stored on this machine's local disk.

Environment Prover This machine runs the Environment Prover. All the name and authorization certificates are pre-loaded into the prover's cache, but authorization chains are generated at runtime.

Data Server The Data Server accepts connections from Jif programs and provides data requested by the programs.

TEP Client This machine runs the TEP client. The client invokes methods on TEP by issuing requests to the TEP server.

Each machine is a 600MHz Intel Pentium III with 512k off-die L2 cache and 512MB RAM. Each runs RedHat Linux 6.0 and Sun's Java Runtime Environment 1.2.2. The machines communicate over 100Mb local Ethernet. 1024-bit RSA keys are used for public-key cryptography.

5.2.2 Session Creation

Establishing a secure channel to TEP is the first step in any client session. The client connects to TEP's socket and initiates the three-round protocol described in Section 4.3.2. The protocol involves several CPU-intensive cryptographic operations. These operations are listed in Table 5.2 in the order they occur on the client side of the protocol.

Decryptions and signatures account for 85% of the time in the client protocol, so optimizing these operations can greatly improve performance (Section 5.2.5). The *Other* amount accounts for the time for connecting the socket and sending and receiving the messages for each round. These times do not include the additional decryption and verification that the server must execute to complete the protocol.

The cost for establishing a secure session with TEP is amortized over the requests made by the client, so clients benefit from issuing multiple requests to TEP each session. The following sections do not include the cost of establishing the initial sessions in the application times.

Task	Time (ms)
Client Sign	88
Server Verify	4.3
Server Encrypt	3.4
Server Sign	88
Client Verify	4.3
Client Decrypt	87
Client Encrypt	3.4
Client Sign	88
Other	44
Total	410

Table 5.2: Times for Session Creation at Client

5.2.3 Test Suite and Overall Results

We test five applications. The first three exercise various features of TEP and demonstrate simple operations. The latter two implement the tax preparation example to demonstrate a real-world application.

Empty This application contains no code and simply provides a baseline for invocation time.

Echo This application reads an object from the invoker and immediately writes it back. It demonstrates the use of standard Java stream classes and object serialization over TEP channels. Our test sends the string "Hello" to the program.

Count This application demonstrates the use of persistent objects and channel creation by incrementing a persistent counter. The counter is stored on disk by the Data Server as a sealed object. The program requests the counter, unseals it, increments it, reseals it, and writes it back to the Data Server. Much of the time in this application (over 600ms) is spent creating a secure channel to the Data Server and checking the channel label.

Tax-1 This application calculates the tax for a given income value. The program reads the invoker's income value, retrieves the tax rate from the Data Server, calculates the tax, and writes the result back to the invoker. This application also packages a proof in the application JAR file. Much of the time in this program is spent reading the proof from the JAR file (over 300ms) and connecting to the Data Server (over 600ms).

Tax-2 This application adds two authority declarations (over 600ms each) and two actsfor statements (over 250ms each) to the Tax-1 application.

Each application is run eight times, each time by a separate client. The result of the first run of each round is thrown out to allow Java's just-in-time compiler to

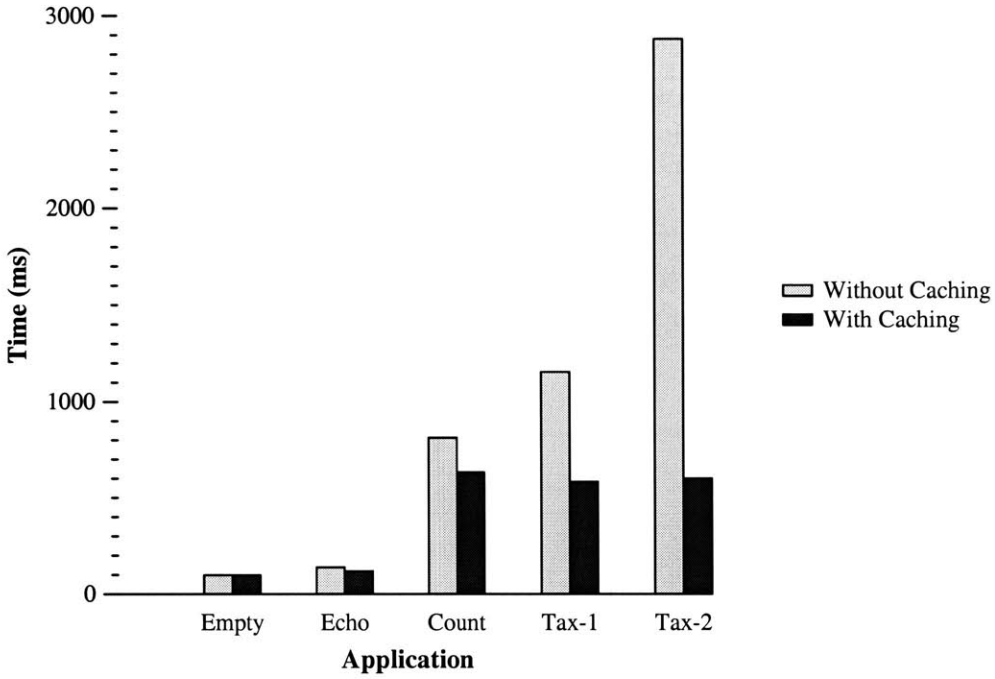

Figure 5-1: Total Times for Applications at Client

optimize the code. The applications are run both with and without caching at TEP. When caching is enabled, TEP saves loaded applications and authorization results between invocations.

Figure 5-1 plots the the invocation times for each of the test applications, both with and without caching. These times do not include the time required to establish the initial secure session between the client and TEP. The figure shows that application and authorization caching can dramatically improve performance for some applications. In the next section, we will analyze each of these applications and explain how their time is used.

5.2.4 Detailed Results

The Empty Application

Table 5.3 gives the execution time for the Empty application at the server. The time at the server is approximately 30ms less than that at the client because the client must wait for the server to complete its part of the session protocol.

At the server, most of the time (87%) is spent creating a new connection to the prover. This connection is used to request authorization proofs, but it is wasted for this application. A possible optimization is to check whether an application contains any actsfor statements or authority declarations before creating this connection. A better optimization is to allow applications to share connections asynchronously, as we discuss in Section 5.2.5.

Task	Time (ms)	
	Without Caching	With Caching
Connect to Prover	55	55
Load Application	6.4	0
Invoke Method	0	0
Other	2	2
Total	63	57

Table 5.3: Times for the Empty Application at Server

Task	Time (ms)	
	Without Caching	With Caching
Connect to Prover	55	55
Load Application	7.7	0.1
Invoke Method	32	30
Other	2	2
Total	97	87

Table 5.4: Times for the Echo Application at Server

A small amount of time is spent loading the application from the JAR file. Since the file is stored on TEP's local disk, most of this time is spent reading the disk. Application caching eliminates this expense, but the difference is too small to be reflected at the client. The *Other* amount accounts for the handoff message from TEP to the client before the application is invoked.

The Echo Application

Table 5.4 gives the execution time for the Echo application at the server. As with the Empty application, much of the time is spent connecting to the prover and loading the application, although caching removes the latter. The application itself spends 30ms reading the argument from the client, deserializing it, reserializing it, and writing it back to the client.

The Count Application

Table 5.5 gives the execution time for the Count application at the server. The time at the client is *less* than the time at the server because the client does not wait for the application to complete – it simply initiates the call and quits. To prevent reads and writes of the persistent counter from overlapping, we must serialize the client requests. We accomplish this by running these tests from a single client, rather than separate ones.

Task	Time (ms)	
	Without Caching	With Caching
Connect to Prover	57	55
Load Application	11	0.1
Open Channel	409	422
Check Reader	206	3.1
Create Sealed	96	95
Read Sealed	94	95
Other	52	47
Total	927	717

Table 5.5: Times for the Count Application at Server

Most of the time in this application is spent opening a secure channel to the Data Server. Creating the session requires approximately 410ms, as described in Section 5.2.2. Strangely, this time *increases* by 2% when caching is enabled. We expect this noise caused by changes in memory allocation behavior and variations in network traffic. The *Other* amount accounts for the time TEP must wait for the Data Server to complete its part of the session protocol.

After creating the session, TEP must check that the Data Server's public key is bound to the name of a reader in the channel label (Section 4.3.5). To check that the data server is an allowed reader, TEP requests a proof from the Environment Prover and checks it locally with the Environment Checker. In this application, the proof contains two name certificates. Approximately 130ms is used to contact the prover and receive the proof; the checker then checks the proof in 75ms, averaging 38ms per certificate. By caching authorization results, TEP greatly reduces the time required the check the reader.

The Count application reads a sealed counter from the Data Server, unseals it, increments it, reseals it, and writes it back. Both reading the sealed object and resealing it use 95ms. When sealing an object, 88ms are used to sign it and 3ms are used to encrypt it. When reading the object, 87ms are used to decrypt it, 4ms are used to verify the signature, and the remaining time is used to read the object from the channel.

The Tax-1 Application

Table 5.6 gives the execution time for the Tax-1 application at the server. Tax-1 does not include any authorization checks, but it does open a channel to the Data Server. As in the Counter application, TEP must check that the Data Server is an allowed reader of the channel. Caching greatly reduces the time needed for this check.

Tax-1 also packages a proof in its JAR file. This proof contains five certificates and requires 190ms to check. The remaining 180ms is used to read the proof from the JAR file and parse it into a SDSI sequence. Again, caching allows the application

Task	Time (ms)	
	Without Caching	With Caching
Connect to Prover	55	55
Load Proof	374	0
Open Channel	408	421
Check Reader	218	3
Other	74	70
Total	1129	549

Table 5.6: Times for the Tax-1 Application at Server

to eliminate this expense.

The *Other* amount accounts for the time TEP waits for the Data Server to complete the session protocol, the time required to request and receive the tax rate from the Data Server, and the time required to read the invoker's argument and write the return value.

The Tax-2 Application

Table 5.7 gives the execution time for the Tax-2 application at the server. Tax-2 adds two authority declarations and two actsfor checks to Tax-1. The results are similar to those for Tax-1, except a large amount of time is spent satisfying the four authorization requests. However, if these results are cached, the time for Tax-2 is almost equal to the time for Tax-1.

The two authority proofs have a total length of fifteen certificates and require 670ms to check. The remaining 570ms from is used to request and receive the proofs from the Environment Prover.

The two actsfor proofs are somewhat more skewed. One is five certificates long and the other is just a single certificate. The short proof requires only 37ms to check and 70ms to prove. The long proof requires 190ms to check and 260ms to prove, for a total time of 450ms. Reading and checking the same proof from the JAR file takes 342ms. In this case, including the proof in the JAR file can reduce total application time by over 100ms.

5.2.5 Improving Performance

There are a number of techniques we can apply to improve performance. One technique is to implement expensive tasks in native code or hardware. Another is to re-use of the results of previous computation. In this section, we discuss how these techniques can be applied to our system.

Task	Time (ms)	
	Without Caching	With Caching
Connect to Prover	56	55
Load Proof	342	0
Check Authority×2	1237	0
Check ActsFor×2	555	1.5
Open Channel	411	420
Check Reader	157	3
Other	73	79
Total	2831	559

Table 5.7: Times for the Tax-2 Application at Server

Native Code

Since TEP is implemented in Java, its bytecode must be interpreted by the Java Virtual Machine. Just-in-time compilation improves the performance of Java by compiling bytecode into machine code, but the system still incurs overhead from Java's runtime mechanisms. By implementing the system in native code, we can remove most of the overhead of the JVM.

To demonstrate the improvement provided by native code, we consider the effect of replacing our Java cryptography implementation with a C++ implementation [Dai99]. Figure 5-2 compares the times for the public-key cryptographic operations in each implementation. The Java operations were run on a 600MHz Pentium III running RedHat Linux 6.0; the C++ operations were run on a 450MHz Celeron running Microsoft Windows 2000 beta 3. The results show that the C++ implementation requires less than one-third the time of the Java implementation.

Figure 5-3 compares the times to establish a secure session using Java and native code cryptography. These times are larger than those in Table 5.2 because they include the cryptography for both the client and server sides of the protocol. Assuming communication time stays the same, the native code implementation is over 60% faster than the original Java implementation. This not only decreases the time required to connect to TEP, but also improves the performance of applications that connect to the Data Server.

Native cryptography also reduces the time required to verify signatures by 3ms, allowing the Environment Checker to check certificates slightly faster. By implementing the Environment Prover and Checker in native code, we can further improve performance. The main drawback of native code is that it does not incorporate Java's protection mechanisms.

Prefetching

Prefetching is a technique that allows TEP to reduce latency for client requests. By caching the results of expensive operations, we can respond to client requests more

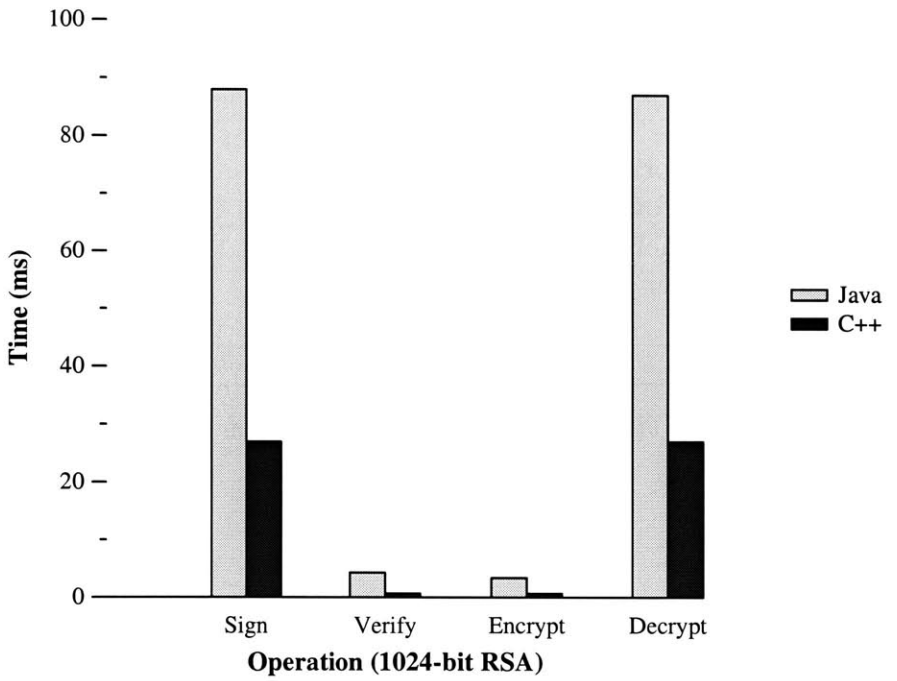

Figure 5-2: Public Key Cryptography Operations

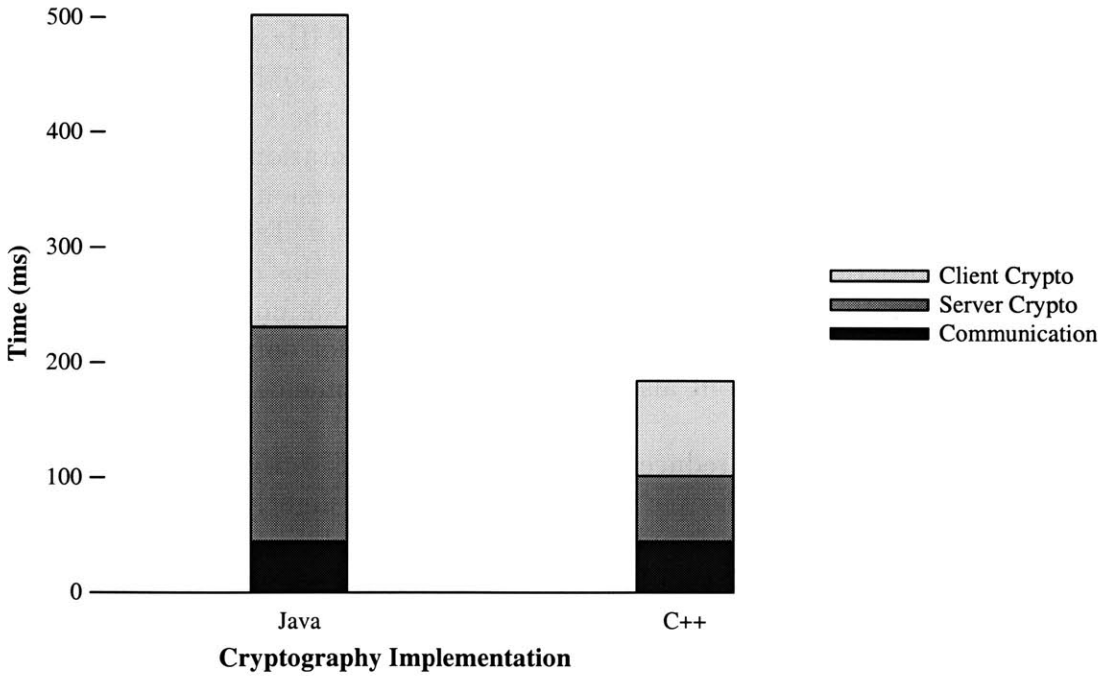

Figure 5-3: Times for Java and Native-Code Session Creation

quickly than if the operations are executed during the requests. Prefetching does not reduce the amount of computation done by TEP, but rather allows TEP to respond to requests with cached results.

Currently, TEP caches both applications and authorization proofs. TEP could prefetch applications and proofs if it could learn which ones would be required by future clients. Such prior knowledge would need to be provided by an outside source.

Another potential target for prefetching is channels. Establishing secure channels is expensive, so TEP could reduce latency by creating channels before client requests. Since it is likely that a given application will contact the same server each invocation, TEP can prefetch channels for the most popular applications. When an application requests a channel, TEP can check the channel pool for any that match the requested host, port, and labels.

Unfortunately, this technique still requires expensive computation by TEP. The next section describes a technique for reducing the number of secure channels that TEP must create.

Channel Sharing

Our implementation uses synchronous streams for communication between processes and provides each process with its own connections to other systems. This provides programmers with a standard stream abstraction and ensures that processes do not block each other when accessing their channels. Unfortunately, creating separate connections for each process is expensive. If, instead, we use *asynchronous* messages for communication, we can implement several optimizations for channels.

The most important optimization is the ability to multiplex existing secure channels for many sets of peers. Separate Jif applications that contact the same external host, port, and principal can share a channel. TEP can mark messages from each application with a unique identifier and use the identifier to sort incoming messages. TEP can also cache channels for use by later application invocations.

Channel sharing need not be limited to secure channels; TEP can also multiplex its connections to the prover so that it need not establish a new connection for each application invocation. This can reduce application invocation time by up to 55ms. However, without asynchronous communication, each application must have its own connection to avoid conflicting with other applications.

The main challenge for sharing channels is ensuring that the labels of the data sent over secure channels are protected. The design must also be modified to use asynchronous messages for all communication.

Parallel Protocol

Another way to reduce session creation time is to allow both parties to work in parallel. Consider this protocol between Alice and Bob:

1. Alice sends Bob her public key and a fresh random nonce, unsigned. At the same time, Bob sends his public key and nonce to Alice.

   Alice to Bob: P_A, R_A

2. Upon receiving the Bob's public key and nonce, Alice encrypts a fresh session key with Bob's public key and then signs the encrypted key and Bob's nonce with her private key. She sends the encrypted key and the signature to Bob. Bob does the same and sends his data to Alice.

   Alice to Bob: $E_B(K_A), S_A(R_B, E_B(K_A))$

3. Alice verifies Bob's signature and decrypts Bob's session key, K_B, with her private key. She encrypts her output stream with K_A and decrypts her input stream with K_B. Bob does the same, using the opposite keys for encryption and decryption.

This protocol can run in one-third the time of our original protocol, since the two signatures are executed in parallel. We did not have the opportunity to implement and benchmark this protocol, but we expect that it can improve the performance of the system significantly.

Chapter 6

Conclusions

In this chapter, we discuss work related to TEP and describe future avenues for research. We conclude with a summary of our contributions.

6.1 Related Work

6.1.1 Secure Multiparty Computation

Secure Multiparty Computation is a set of techniques that allow mutually-distrusting parties to share data in computation without using a trusted third party [Gol98]. The techniques combine cryptography with distributed computation to emulate third-party computation without revealing any party's private data to any other party. Techniques have been developed that support general computation as well as specialized operations, such as distributed key agreement.

If secure multiparty computation provided a practical programming model and reasonable performance, TEP would be unnecessary. Currently, however, these techniques provide neither. Although the building-blocks for general computation exist, they have not been incorporated into a practical programming system. Jif, in contrast, provides familiar Java syntax, low overhead, and intuitive control over information flow.

Each computation in secure multiparty computation can require several rounds of communication between parties, and each round can require several cryptographic operations. Since programs can be built of hundreds of these basic operations, performance is too slow to be practical. Unless these techniques can be made practical, TEP provides a better alternative for multiparty computation.

6.1.2 Alternate Security Environments

A number of other Public Key Infrastructure (PKI) and certificate systems have been developed that provide interesting alternatives to SPKI [SPK98]. The most common is the X.509 PKI [PKI00]. This system is currently used by companies for trusted authentication services and could be adapted for TEP. However, X.509 uses distinguished Certification Authorities, while SPKI is fully decentralized allows any

principal to issue certificates. We prefer SPKI's decentralization because it gives principals more flexibility in building authorization and authentication structures.

Another system that could be adapted for TEP is Secure DNS [DNS99]. This system is currently under development and endeavors to provide a widely-deployed, secure distribution system. Unfortunately, DNS is plagued by performance challenges and is even more centralized than X.509, so SPKI seems to be the better model.

A final system to consider is the Query Certificate Manager (QCM) system [GJ98, GJ99]. QCM is similar to SPKI in that it is decentralized and supports linked local name spaces. QCM accepts powerful set-based queries that provide more flexibility than the current SPKI implementation. However, QCM's certificate discovery and verification algorithms are less efficient than those for SPKI and QCM does not support threshold subjects (Section A.3).

6.2 Future Work

Jif provides a natural programming model that protects the privacy of data in its programs. TEP endeavors to make using such programs equally natural. However, the current design must restrict the programs it runs to preserve the information flow guarantees provided by Jif. In this section, we discuss how some of these restrictions can be removed by extending TEP's design.

6.2.1 Extending Channels

TEP's channel model requires that programs initiate connections to servers with known hosts and ports. This model supports input and output to any device that can be connected to a network server, but it does not support incoming connections to TEP programs or multicast output channels. The model also releases data to any reader in a channel's output label, even if the reader cannot be trusted to honor the policies on that data. In this section, we describe extensions to address these problems.

Untrusted Readers

Our design treats the readers on output channels as *trusted peers*. That is, we expect a reader will preserve any policies on data they read from the channel. This model puts the burden of trust on the owners of the policies that allow the peer to read the data.

This model will not work if the peer is untrustworthy. For example, data labelled {`Bob: Alice`} can be written to a channel connected to Alice, but Alice can then disseminate the data arbitrarily. There is no way to force Alice to honor Bob's policy, since she is not operating on a trusted platform.

One solution to this problem is to model *untrusted peers* for output channels. To do this, we must assume that the peer gains total control over the data and will not honor any policies but its own. We can implement this model with this Jif API:

```
class UntrustedOutputChannel[principal P] {
    UntrustedOutputChannel[P](String{} host, int{} port);
    jif.io.OutputStream[{P:}] getOutputStream();
}
```

This API specifies the Jif class `UntrustedOutputChannel` that models a channel connected to an untrusted peer. The peer is specified using the class principal parameter `P`; TEP requires that the peer's key is the `sameAs` the principal `P`. The API forces the Jif programmer to relabel all data written to the channel with the peer's policy, `{P:}`. Although this API does not change the power of the application, it does force the programmer to declassify data released to an untrusted principal.

The untrusted model does not allow trusted peers to preserve the original policies on data written to the channel. Our model, in contrast, sends the output label of the channel to the peer. The trusted peer can then use the label in Jif programs that run on their local machine. For example, a tax preparer might release a tax form to its client with the label `{client: client; preparer; client}` and hold the client liable if the information leaks from the client's machine. The client can use local Jif programs to ensure the labelled data is not accidentally leaked from its system.

Acts-For Readers

An important difference between TEP's output channel model and Jif's semantics is that TEP does not allow a principal that actsfor an allowed reader to read from the channel. TEP's check is more restrictive: it only accepts principals that appear explicitly in the reader set of the label. The main reason for this difference is to avoid implicit runtime actsfor checks. We may be able to make these checks explicit by recording information about the static principal hierarchy in the channel code, but we must be careful to avoid adding unnecessary overhead.

Incoming Connections

Incoming connections could be supported by implementing "server sockets" for Jif programs. Such sockets must be wrapped in a class that creates a labelled channel for each new client. These channels can have either *static labels* or *dynamic labels.*

Static labels require that each client key be bound to the name of a reader in the channel label. Consider a socket that creates channels labelled `{student: student}`. If the principals with keys K_A, K_B, and K_C connect to the socket, TEP must be able to find proofs that the name `student` is bound to each key.

Dynamic labels allow the label on each channel to be determined at runtime. The Jif program must then use runtime label discrimination to handle data read from or written to the channel. Jif implements runtime label discrimination with the switch label construct, described in [ML00].

The main challenge in implementing server sockets is determining how to multiplex incoming connections to the programs running on TEP. We are unwilling to allow programs to listen on an arbitrary port on TEP, since this consumes resources on the trusted base and can cause conflicts if different programs request the same port.

Incoming connections can connect to TEP's port, but must then also identify the program to which to connect. Since multiple instances of the same program can be running on TEP at the same time, disambiguating these connections can add substantial complexity to the trusted base. We may be able to move this complexity to untrusted code, since even if a channel connects to the wrong program, that program cannot leak any information.

Certain applications, such as auctions and elections, combine classified data from multiple incoming connections and could take advantage of TEP's mechanisms to protect the privacy of bidders and voters. For these applications, we can deploy specialized TEPs that can use identifiers for individual auctions and elections to route incoming connections. These identifiers can be assigned by an untrusted system (such as the auction web page) and associated with the program when it is invoked by the auctioneer. Care must be taken to ensure no two auctions have the same identifier.

Multicast Output Channels

The channel model can be extended to support multicast and anycast output channels. Such channels are useful for disseminating classified data to a group of authorized readers. TEP would need to encrypt the output session key with each of the readers public keys and send all the keys over the channel, although a better solution would use a single public key for the reader group. Currently, we can simulate this behavior by using a trusted relay that reads data from the program and multicasts it to the recipients.

6.2.2 Increasing Trust

TEP depends on the trust of its clients and, therefore, must make every effort to protect that trust. This section describes mechanisms that increase our confidence in TEP's implementation and reduce its complexity.

Implementing TEP in Jif

Even with our efforts to reduce the size of the trusted base, TEP's implementation exceeds ten thousand lines of Java code. This amount does not include the code in the Java libraries, the Java runtime system, or the underlying operating system. A single bug in this code can open the system to attack and can compromise its ability to protect client data.

We do not have techniques for identifying arbitrary bugs, but we can identify mistakes that leak information using Myers' information flow analysis. The Jif language and the analysis it supports can be used for any program, not just ones that run on TEP. By writing TEP in Jif, we can use Jif's static analysis to check TEP itself for information leaks.

Static analysis will reveal the places where TEP must use its authority to declassify client data. Declassification will be necessary when TEP creates labelled channels over raw sockets and when TEP seals persistent objects, and may be needed elsewhere.

We expect these declassifications to occur rarely, so most of TEP's code can preserve the labels on client data.

Static analysis will also reveal any implicit flows that leak information. This analysis allows us to identify necessary declassifications and fix any accidental information leaks. TEP must use its authority implicitly when it authorizes a program or satisfies an actsfor test; these mechanisms may not be identified by Jif.

Splitting the Checker

The Environment Prover-Checker split reduces the size of TEP's trusted base. It is possible to improve this further by moving the certificate parsing functionality of the checker into a new component, the translator. The translator reformats incoming proofs to a canonical form and then passes the proof to the checker. When the implementation of certificates or the security environment changes, only the translator need change to handle new certificate formats. Although this split does not reduce the size of the trusted base, it reduces the amount of trusted code that needs to change when the security environment changes.

The translator also allows the checker to work with multiple types of systems: names could be implemented with X.509 certificates while authorizations could use SPKI certificates. It may even be possible to combine certificates of different types, if such reasoning were secure.

To implement this feature, we must determine if it is possible to translate different certificate formats into a single canonical form. Both the translator and the checker must still be trusted, since it may not be possible to verify certificates after translation.

Thin Operating Systems

TEP's current implementation must trust the Java Virtual Machine and the underlying operating system. Ideally, a trusted system would be implemented with as little trusted code as possible, while still providing useful functionality. Systems like Exokernel that provide low-level access to trusted hardware can allow us to reduce the size of the trusted base by implementing TEP with simpler primitives [MK97]. However, this benefit must be balanced against the risk of increased coding errors due to using a lower level of abstraction.

6.2.3 Protecting Integrity

TEP is designed to protect the privacy of its clients' data. Unfortunately, an attacker can easily corrupt an application if it can destroy the integrity of data used by the application. Users often want integrity guarantees on data they receive from a trusted source. This section explores mechanisms with which TEP can protect the integrity of client data, in addition to its privacy.

Integrity Labels

Myers' label model is designed to protect the privacy of data in a distributed environment. The model ensures that only authorized parties can read classified data. However, it does not protect *integrity*: there are no restrictions on which principals can write to a piece of data.

Myers addresses this problem by introducing the logical dual of privacy labels: *integrity labels* [ML00]. These labels contain policies that specify which principals may *write* to a piece of data. For example, the client of a tax preparation service wants the guarantee that only its income data and the tax preparer's data have been incorporated into the resulting tax form. The tax preparer can provide this guarantee by annotating the data with the appropriate integrity labels.

Input channels can be labelled with integrity policies that identify the allowed writers to the channel. Output channels send their integrity labels to their peers to provide them with integrity guarantees. Remote parties can use this integrity information to determine whether they trust the data sent by the program running on TEP. For example, election servers can check that all votes are cast using a particular voting application and thus ensure the votes are recorded correctly.

Jif would need to be modified to statically check integrity labels along with privacy labels. Myers describes how decentralized labels can enforce integrity and privacy independently and simultaneously. Since programs can affect the integrity of data, TEP would have to support a new type of certificate that guarantees a specified class preserves the integrity of data it uses.

In a sense, integrity labels "complete the loop" in trusted computation. A number of mechanisms would need to be developed to support integrity labels, but we expect this could be done while still providing practical service.

Byzantine-Fault-Tolerant Replication

Byzantine-fault-tolerance protects systems against attacks that affect the their correctness. Castro's BFT algorithm defines how to replicate a service efficiently to withstand arbitrary attacks against up to a third of the *replicas* (machines that replicate the service) [CL99]. The algorithm does not protect the privacy of data in the service, since even a single compromised replica can be used to observe application data, but it can protect integrity by rejecting replicas whose computation is corrupted.

TEP can be replicated using Castro's BFT algorithm to reduce the threat of malicious attacks against the system. Each replica runs the program requested by the client; the client then waits until it received $2f + 1$ matching replies from the programs. Other parties involved in the computation must coordinate their systems with each TEP replica.

BFT's recovery mechanism can also protect TEP against crash-stop failures, making TEP highly-available. The main deterrent against BFT is that it adds a substantial amount of complexity to the trusted base and may increase the risk of attacks against private data.

6.2.4 Resource Control

TEP's current design does not protect TEP's network or CPU resources. Jif applications can flood TEP's network connections and can use arbitrary amounts of CPU time, effectively launching denial-of-service attacks against TEP from within. To counteract this risk, TEP must incorporate resource control and scheduling.

Several projects have developed Java operating systems that provide process isolation and resource control [BTS+00]. They allow TEP to prevent foreign programs from consuming too much CPU time or network bandwidth, making TEP resilient against internal denial-of-service attacks. Using a Java process model, TEP can ensure that well-behaved programs receive a fair share of TEP's resources.

6.3 Summary

As information becomes increasingly valuable, protecting its privacy becomes more and more important. Cryptographic techniques have been employed to protect data in storage and in transit, but they have not been able to protect data in computation. Until practical systems are developed that allow mutually-distrusting parties to share data in computation, trusted third parties are needed to protect the privacy interests of those parties.

This thesis describes a system that provides a trusted computation service to its clients. The system incorporates fully-decentralized models for tracking information flow and authorizations, allowing clients to develop their own trust hierarchies without using a centralized authority. The system uses static checking to ensure that the programs it runs cannot leak classified information without authorization. The system encrypts data that exits the system to ensure that it is released only to the intended recipients. The system relies on the trust of its clients and uses that trust to protect their privacy.

We have described how a system for information flow between abstract principals can be grounded in a real-world environment of networks, certificates, keys, and machines. We have used this combination to implement the Trusted Execution Platform, a service that provides secure multiparty computation for practical programs. We have detailed TEP's design, analyzed its performance, and demonstrated its flexibility with several applications.

Providing trusted and secure service is a challenging problem that will become increasingly important as systems become more complex. We hope this thesis has demonstrated how a few powerful techniques can be combined create a useful service.

Appendix A

Certificate Chain Algorithms

This appendix describes our algorithms for creating and checking certificate chains. As described in Chapter 3, TEP uses certificate chains to check name resolutions and authorizations. These chains are created by the Environment Prover and are checked at TEP by the Environment Checker.

Section A.1 presents the Checker's algorithm, and Section A.2 presents the Prover's algorithm. Section A.3 describes how we extend these algorithms to support threshold subjects.

A.1 Checking Certificate Chains

Security environments are based on simple components – public keys and certificates – but require significant complexity to manage their large distributed databases. TEP moves this complexity out of the trusted base by using a simple checker to check proofs of security properties. All the difficulties involved in searching for certificate chains are handled by an untrusted component called the *Environment Prover.*

The Environment Prover encodes certificate chains as SPKI *Sequences.* Sequences are lists of certificates and their signatures. A simple algorithm can parse a sequence to check if it proves a desired property. TEP implements this algorithm in a trusted component called the *Environment Checker.* Figures A-1 and A-2 give the checker's algorithm in pseudocode.

A.1.1 Checking Name Resolutions

The *Name* datatype is a tuple containing the name's *issuer* and a list of the name's *subnames.* The issuer is the public key in which the name is based (recall that Jif names are resolved starting with TEP's public key). Subnames are simply the character strings that compose an extended name. In the name "dns edu mit lcs ajmani", the subname list is ["dns", "edu", "mit", "lcs", "ajmani"].

SPKI resolves names using chains of name certificates, or *defs.* A def binds a subname to a *Subject*; a Subject is a Name, a PublicKey, or the digest of a class. To resolve a Name, a sequence of defs resolves each subname into a key. The key bound

to one subname must issue the def that binds the next (Section 3.2). The def that resolves the last subname may be bound to any Subject.

The Environment Checker resolves a name by requesting a proof from the Environment Prover. The checker then calls RESOLVE-NAME on the proof and the name. RESOLVE-NAME returns the Subject to which the name is bound. In most cases, TEP need only check if the returned subject matches an expected subject (the implementation optimizes for this case).

RESOLVE-NAME works by setting its starting point as the name's issuer key and breaking the name into subnames. It then checks that the next item in the certificate proof is a def that binds the current subname to a public key. RESOLVE-NAME repeats the loop until each subname is resolved, then returns the final subject.

CHECK-DEF implements the inner loop of RESOLVE-NAME by reading a certificate from the proof, checking that it binds the current subname, and recursively resolving the subject if it is a name.

This last step, implemented by MAYBE-RESOLVE-NAME, reflects an ambiguity in the proof grammar. Since the goal of a name resolution or authorization may be a name, the checker does not know whether to resolve a name or return it directly. If the checker happens to try to resolve a name when it should have returned it, it backs up to the previous position. This case is rare, so the performance impact is minimal. Disambiguating the proof grammar is left for future work.

A.1.2 Checking Certificates

The *Certificate* datatype is a 5-tuple. Each certificate specifies an *issuer* key, a *tag* string, a *subject*, a *propagate* bit, and a *validity* period. The *tag* identifies either what subname the certificate binds or what authorization it grants.[1] Authorization certificates use the propagate bit to denote whether they allow their subjects to delegate authority (name certificates do not use the propagate bit).

READ-CERT reads a certificate from the proof and checks that it is followed by its issuer's cryptographic signature. READ-CERT also checks if the certificate has expired. The current implementation does not support online checks for certificates (such as revocations or revalidations), but this can be added easily by requiring a validation certificate for each certificate in the sequence.

A.1.3 Checking Authorizations

SPKI proves authorizations with chains of authorization certificates, or *auths*. Each auth grants an authorization to a Subject and specifies if the subject is allowed to propagate the authorization. In TEP, the final target of the authorization is always known and is provided to the prover and the checker.

CHECK-AUTH reads a certificate from the proof, checks that it grants the proper authorization, and resolves the subject if it is a name. CHECK-AUTH then checks if

[1] In actual SPKI certificates, name definition certificates and authorization certificates are different types, so there is no ambiguity.

the subject of the certificate matches the target subject; failing that, it attempts to propagate the authorization through the current subject.

A.1.4 Implementation

TEP's checker algorithm is simple to implement and runs efficiently on long certificate chains ($O(n)$ expected case). The most expensive task is signature verification, which can be improved using native-code or hardware-based cryptography. This algorithm is specialized for SPKI certificate sequences, but similar algorithms can work with any system that uses certificates. Certificate chain checking is encapsulated in the TEP's trusted Environment Checker; TEP's interface to the checker is given in Section 4.1.4.

A.2 Certificate Chain Discovery

The complement to TEP's Environment Checker is the Environment Prover, which must generate authorization and name resolution proofs. Rivest et al give an algorithm for proof discovery based on certificate closure [CEE+99]. This algorithm is complete – it finds all possible proofs – but it is impractical in distributed systems. We have developed a certificate chain discovery algorithm that works in distributed systems, but is incomplete.

The prover's algorithm is essentially a depth-first search that branches at name resolutions [Aur97, Aur98]. Authorization searches traverse a graph of certificates starting from the issuer, seeking a specified target. At each step, the current search path is checked to avoid cycles. Certificates without the propagation bit set cause the current branch to stop searching.

Names are resolved as soon as they are encountered. They are resolved by calculating the set of keys bound to the current subname and using the resulting set to resolve the next subname. The final set of keys introduces branches in the certificate graph for each bound key.

This algorithm can be extremely slow if certificates are spread widely across a network, though work on an efficient certificate distribution scheme is currently underway. This implementation searches for proofs on a certificate cache stored at the Environment Prover, so the search runs in the prover's local memory. This section describes the prover's search algorithm in detail.

A.2.1 Proving Name Resolutions

To prove that a given name is bound to a given public key, the prover must find a certificate chain that resolves the name into that key. TEP uses name resolution to check that the principal reading from an output channel is the same as some reader in the channel's reader set (Section 4.3.5). The algorithm for name resolution is given in Figures A-3 and A-4 .

To resolve a name into a key (or any Subject), the prover calls PROVE-NAME on the requested name, the target subject, and an empty proof sequence. PROVE-NAME

```
Sequence: sequence of SequenceItem
SequenceItem: Certificate | Signature
Certificate: ⟨issuer: PublicKey, tag: String, subject: Subject,
              propagate: Boolean, validity: DateRange⟩
Subject: PublicKey | Name | ClassDigest
Name: ⟨issuer: PublicKey, subNames: sequence of String⟩

Subject ← CHECK-DEF(proof: Sequence, issuer: PublicKey, subName: String)
1       cert ← READ-CERT(proof, issuer)
2       if cert is null or cert.tag does not equal subName, return null
3       return MAYBE-RESOLVE-NAME (proof, cert.subject)

Subject ← MAYBE-RESOLVE-NAME(proof: Sequence, subject: Subject)
1       savedSubject ← subject, savedIndex ← proof.index
2       if subject is a Name,
3               subject ← RESOLVE-NAME(proof, subject)
4               if subject is null,
5                       proof.index ← savedIndex, subject ← savedSubject
6       return subject

Subject ← RESOLVE-NAME (proof: Sequence, name: Name)
1       subject ← name.issuer
2       for each subName in name.subNames do
3               if subject is not a PublicKey, return null
4               subject ← CHECK-DEF (proof, subject, subName)
5       return subject

Certificate ← READ-CERT(proof: Sequence, issuer: PublicKey)
1       if proof has ended, return null
2       cert ← next item in proof
3       if cert is not a Certificate or cert.issuer does not equal issuer
4         or cert.validity is not current, return null
5       sig ← next item in proof
6       if sig is not issuer's signature of cert, return null
7       return cert
```

Figure A-1: Checker Algorithm: Name Resolution and Certificates

```
CHECK-AUTH(proof: Sequence, issuer: PublicKey, tag: String, target: Subject)
1     cert ← READ-CERT(proof, issuer)
2     if cert is null or cert.tag does not equal tag, reject
3     subject ← MAYBE-RESOLVE-NAME (proof, cert.subject)
4     if subject equals target, accept
5     if subject is a PublicKey and cert.propagate is true,
6           CHECK-AUTH(proof, subject, tag, target)
7     else reject
```

Figure A-2: Checker Algorithm: Authorizations

attempts to write a proof to the sequence and returns *true* if it succeeds.

PROVE-NAME works by calculating all possible resolutions of the given name. PROVE-NAME calls PROVE-NAME-SET on the name and an empty proof sequence. We must be careful to avoid infinite loops in the certificate graph which may occur when names are bound to other names. At the same time, we must allow proofs for *different* names to use the same certificates. By giving each name an empty sequence, we resolve names in separate spaces while still avoiding loops.

PROVE-NAME-SET returns its resolutions as a set of *ProofSubjects*: each *ProofSubject* is a tuple containing a subject and a proof that the name resolves into that subject. PROVE-NAME then checks each proof-subject pair to see if any of the subjects match the target. If so, the associated proof is written to the sequence and PROVE-NAME returns *true*.

PROVE-NAME-SET returns the set of subjects bound to the given name, and a proof for each subject. PROVE-NAME-SET works by calculating the set of principals bound to each subname and using those principals to resolve the next subname. The set of subjects bound to the last subname is returned.

To calculate the principals bound to a subname, PROVE-NAME-SET calls PROVE-DEFS on the current subname for each current issuer. Each call to PROVE-DEFS returns a set of subjects that is added to the issuers for the next iteration. Once PROVE-DEFS has been called for each of the current issuers, the next set of issuers is used to resolve the next subname.

PROVE-DEFS calculates the set of subjects bound by a given issuer to a given subname. First, the set of defs issued by the issuer for that subname is retrieved using GET-DEFS. PROVE-DEFS then calculates the union of the sets of subjects bound by each def.

GET-DEFS implements the retrieval of name certificates that match a given issuer and subname. In our implementation, GET-DEFS retrieves the defs from a certificate cache local to the prover. In SPKI, certificates can be deployed widely across the network, although certificates by the same issuer are usually available from a single server. This design supports various distribution models easily by allowing the implementor to redefine GET-DEFS without changing the rest of the prover.

PROVE-DEF calculates the set of subjects bound to a given def (name certificate). PROVE-DEF first checks that the def is either bound to a key or has not already been

used in the proof sequence. This restriction ensures that this certificate will not cause the prover to enter an infinite loop (see the section on name loops below).

PROVE-DEF then saves the current proof index and attempts to write the certificate to the proof using WRITE-CERT. If this fails, the proof index is restored and the empty set is returned. If WRITE-CERT succeeds, PROVE-DEF calls PROVE-DEF-INNER to calculate the set of subjects bound to the def. The proof index is then rolled back to the saved index, since the returned set contains the proofs for each subject.

PROVE-DEF-INNER creates a new set of *ProofSubjects* and adds a copy of the current proof with the subject of the def. If the subject is itself a name, the set of subjects bound to that name is also returned. Recall that the target of the proof may be the name itself, which is why the subject is returned in a separate *ProofSubject*.

Name Loops and Incompleteness

The prover can enter a name resolution loop if a def binds a subname to the same name or to another name that causes the prover to return to the same certificate. For example, if Alice issues the certificate K_A Alice $\rightarrow$ K_A Alice, resolving the name "K_A's Alice" causes the prover to return to the same certificate.

We can avoid loops by checking that each def used in a proof is either bound to a key or is used at most once in the current proof branch. If the def is bound to a key, this branch of the proof search terminates immediately. Otherwise, we must ensure that each def is used at most once so this branch of the proof will terminate.

Unfortunately, these restrictions mean the prover cannot find proofs for names that require multiple applications of a name certificate whose subject is a name. Although this makes the prover incomplete, this situation is not expected to occur in typical naming graphs.

A.2.2 Proving Certificates

The procedures for writing certificates to the proof are given in Figure A-4. WRITE-CERT attempts to write the given certificate and its signature to the proof and returns *true* if it succeeds. WRITE-CERT fails either if the certificate is invalid or if it cannot find the signature for the certificate using GET-SIGNATURE.

GET-SIGNATURE, like GET-DEFS, is implemented according to the certificate distribution used by the prover. Our implementation stores signatures in a local cache, but we expect most networked deployments to keep signatures with their certificates.

A.2.3 Proving Authorizations

Figures A-5 and A-6 give the algorithm for finding chains of authorization certificates. To prove an authorization, the prover calls PROVE-AUTHS on the issuer of the authorization, the authorization identifier *tag*, and the target subject of the authorization. PROVE-AUTHS attempts to write the proof to the given proof sequence and returns *true* if it succeeds.

PROVE-AUTHS calls GET-AUTHS to retreive all the authorization certificates (*auths*) that match the given issuer and tag. It then calls PROVE-AUTH on each auth to check if it authorizes to the target subject.

GET-AUTHS returns the set of auths issued by the given issuer for the given authorization tag. Like GET-DEFS, GET-AUTHS is implemented for the particular certificate distribution used by the prover. In our implementation, auths are stored in a local certificate cache.

PROVE-AUTH starts by checking if the given auth has already been used in the proof. If so, this branch has entered a cycle in the authorization graph and must stop to avoid looping. Otherwise, PROVE-AUTH saves the proof index and attempts to write the certificate to the proof using WRITE-CERT. If it succeeds, PROVE-AUTH calls PROVE-AUTH-INNER to check whether the given auth authorizes the target subject and, if so, returns *true*. If either of these steps fail, the proof is rolled back to the saved index and PROVE-AUTH returns *false*.

PROVE-AUTH-INNER starts by checking if the subject of the given auth is the target subject. If so, it returns *true*. Otherwise, PROVE-AUTH-INNER saves the proof index, checks if the subject is a name and, if so, attempts to resolve the name into the target. If the resolution succeeds, it returns *true*. Otherwise, PROVE-AUTH-INNER restores the proof index and checks if the auth's propagate bit is set. If it is not, this branch of the proof fails and PROVE-AUTH returns *false*.

If the propagate bit is set, then the given auth allows delegation. PROVE-AUTH-INNER attempts to propagate the authorization through the subject. If the subject is a key, PROVE-AUTH-INNER calls PROVE-AUTHS with the subject as the new issuer. If the subject is a name, PROVE-AUTH-INNER calls PROVE-NAME-SET to resolve the name and calls PROVE-AUTH-SET on the resulting set of subjects.

PROVE-AUTH-SET accepts a set of *ProofSubjects* and attempts to find an authorization from any of those subjects to the target. PROVE-AUTH-SET iterates through the set looking for subjects that are keys (any names in the set will have been resolved into keys). For each key, PROVE-AUTH-SET writes the accomanying proof and calls PROVE-AUTHS with the key as issuer. If the call succeeds, PROVE-AUTH-SET returns *true*; otherwise, it rolls back the proof and continues to the next key. If no branch succeeds, PROVE-AUTH-SET returns *false*.

A.2.4 Implementation

Our implementation puts the prover and the certificate cache on a single server, separate from TEP. It is important that TEP be on a separate machine, since the prover is untrusted and could compromise TEP if they shared a machine. The prover has direct access to the certificates, so searching the cache is fast.

Nonetheless, the prover's algorithm is slow because it must attempt to resolve each name it encounters when proving a property. The prover also does not cache the intermediate sequences it creates, so it often repeats work. We are currently researching new algorithms and certificate distribution schemes that may provide better performance in distributed systems.

```
ProofSubjectSet: set of ProofSubject
ProofSubject: ⟨proof: Sequence, subject: Subject⟩

Boolean ← PROVE-NAME( name: Name, target: Subject, proof: Sequence)
1       set ← PROVE-NAME-SET(name, new Sequence)
2       if set is null, return false
3       for each pair in set do
4               if pair.subject equals target,
5                       add all items in pair.proof to proof,
6                       return true
7       return false

ProofSubjectSet ← PROVE-NAME-SET( name: Name, proof: Sequence)
1       set ← { new ProofSubject ⟨copy of proof, name.issuer⟩ }
2       for each subName in name.subNames do
3               nextSet ← {}
4               for each pair in set do
5                       if pair.subject is not a PublicKey, continue
6                       nextSet ← nextSet ∪
7                                       PROVE-DEFS(pair.subject, subName, pair.proof)
8               set ← nextSet
9       return set
```

Figure A-3: Prover Algorithm: Name Resolution

```
ProofSubjectSet ← PROVE-DEFS( issuer: PublicKey, subName: String,
                                proof: Sequence)
1       defSet ← GET-DEFS(issuer, subName)
2       set ← {}
3       for each cert in defSet do
4               set ← set ∪ PROVE-DEF(cert, proof)
5       return set

set of Certificate ← GET-DEFS( issuer: PublicKey, subName: String)
        [implementation specific: returns all matching defs]

ProofSubjectSet ← PROVE-DEF( cert: Certificate, proof: Sequence)
1       if cert.subject is not a PublicKey
2         and cert is already in proof, return {}
3       savedIndex ← proof.index
4       if ¬WRITE-CERT(cert, proof),
5               proof.index ← savedIndex
6               return {}
7       set ← PROVE-DEF-INNER(cert, proof)
8       proof.index ← savedIndex
9       return set

ProofSubjectSet ← PROVE-DEF-INNER( cert: Certificate, proof: Sequence)
1       set ← { new ProofSubject ⟨copy of proof, cert.subject⟩}
2       if cert.subject is a Name,
3               set ← set ∪ PROVE-NAME(cert.subject, proof)
4       return set

Boolean ← WRITE-CERT( cert: Certificate, proof: Sequence)
1       if cert.validity is not current, return false
2       append cert to proof
3       sig ← GET-SIGNATURE(cert)
4       if sig is null, return false
5       append sig to proof
6       return true

Signature ← GET-SIGNATURE( cert: Certificate)
        [implementation specific: returns signature for certificate]
```

Figure A-4: Prover Algorithm: Name Definitions and Certificates

```
Boolean ← PROVE-AUTHS(issuer: PublicKey, tag: String,
                      target: Subject, proof: Sequence)
1    authSet ← GET-AUTHS(issuer, tag)
2    for each cert in authSet do
3        if PROVE-AUTH(cert, target, proof), return true
4    return false

set of Certificate ← GET-AUTHS( issuer: PublicKey, tag: String)
    [implementation specific: returns all matching auths]

Boolean ← PROVE-AUTH( cert: Certificate, target: Subject, proof: Sequence)
1    if cert is already in proof, return false
2    savedIndex ← proof.index
3    if ¬WRITE-CERT(cert, proof),
4        proof.index ← savedIndex
5        return false
6    if PROVE-AUTH-INNER (cert, target, proof), return true
7    else proof.index ← savedIndex, return false
```

Figure A-5: Prover Algorithm: Authorizations, Part 1

A.3 Threshold Subjects

In addition to boolean delegation (the propagation bit), SPKI supports *threshold subjects* [EFL+99]. Threshold subjects allow the issuer of a certificate to require k of n specified subjects to grant a permission before the issuer grants permission. We have implemented support for threshold subjects in SPKI and incorporated it into the certificate chain discovery and checker algorithms. We have not described it in the previous sections since it does not relate directly to authorizations in TEP, but it does add flexibility to the system.

Consider this certificate:

```
(cert
  (issuer <Alice>)
  (subject (k-of-n 2 3 Bob Carol Dave))
  (propagate)
  (tag actsfor)
)
```

This certificate specifies a threshold subject that requires that two out of three of Alice's friends grant `actsfor` to another principal before that principal can act for Alice. If, for example, Bob and Dave both authorize Eve, Eve can act for Alice. This certificate itself does not grant `actsfor` to Bob, Carol, or Dave – just the right to collectively propagate it. Threshold subjects are also useful for program authorization: a principal can require that some number of trusted organizations

```
Boolean ← PROVE-AUTH-INNER(cert: Certificate, target: Subject,
                                    proof: Sequence)
1     if cert.subject equals target, return true
2     if cert.subject is a Name,
3           savedIndex ← proof.index
4           if PROVE-NAME(cert.subject, target, proof), return true
5           proof.index ← savedIndex
6     if cert.propagate is true,
7           if cert.subject is a Public Key,
8                 return PROVE-AUTHS (cert.subject, cert.tag, target, proof)
9           else if cert.subject is a Name,
10                return PROVE-AUTH-SET(
11                      PROVE-NAME-SET(cert.subject, new Sequence),
12                      cert.tag, target, proof)
13    return false

Boolean ← PROVE-AUTH-SET( set: ProofSubjectSet, tag: String,
                                target: Subject, proof: Sequence)
1     savedIndex ← proof.index
2     for each pair in set do
3           if pair.subject is not a PublicKey, continue
4           append all items in pair.proof to proof
5           if PROVE-AUTHS(pair.subject, tag, target, proof), return true
6           proof.index ← savedIndex
7     return false
```

Figure A-6: Prover Algorithm: Authorizations, Part 2

approve a program before granting it authorization.

Our prover algorithm attempts to satisfy a threshold subject by finding separate proofs for each subject, stopping after the threshold is met. The separate proofs are appended to the final proof sequence in the order the subjects appear in the certificate. If a subject cannot be satisfied, a "skip" opcode is added to the sequence.

Consider this certificate:

$$K_A \xrightarrow{actsfor} (2/3)K_B, K_C, K_D \tag{A.1}$$

(A.1) grants `actsfor` from K_A to any principal authorized by two out of the three principals K_B, K_C, and K_D. Assume these certificates also exist:

$$K_B \xrightarrow{actsfor} K_E \tag{A.2}$$

$$K_D \xrightarrow{actsfor} K_E \tag{A.3}$$

The prover can now prove that K_A grants `actsfor` to K_E with the proof (A.1), (A.2), (*skip*), (A.3). The checker works in the expected fashion: when it encounters a threshold subject, it recursively checks the proof for each subject in turn, skipping when specified. The checker keeps track of the number of valid subject proofs given; if it meets the threshold, the certificate is satisfied.

This implementation of threshold subjects can increase proof size unnecessarily if different branches of a threshold subject use the same sequence of certificates. For example, a threshold subject that requires both "dns mit bob" and "dns mit alice" will repeat the certificates that resolve "dns mit" in both branches of the proof. If a threshold subject is resolved using another certificate with a threshold subject, the proof can grow exponentially. Fredette has developed a new structure for SPKI proofs that eliminates this blowup in proof size [Fre98].

Another problem is that the prover may repeat redundant searches for each branch of the threshold subject. This can be fixed by caching intermediate sequences of certificates by the result they prove. For example, the sequence that resolves "dns mit" can be cached and used to resolve both "dns mit bob" and "dns mit alice". In effect, this calculates the *closure* of the certificates traversed by the prover, as defined in [CEE+99].

Bibliography

[Aba97] Martín Abadi. Secrecy by typing in security protocols. In *Proc. Theoretical Aspects of Computer Software: Third International Conference*, September 1997.

[Aba98] Martín Abadi. On SDSI's linked local name spaces. *Journal of Computer Security*, 6(1–2):3–21, October 1998.

[Aur97] Tuomas Aura. Comparison of graph-search algorithms for authorization verification in delegation networks. In *2nd Nordic Workshop on Secure Computer Systems*, Espoo, Finland, November 1997.

[Aur98] Tuomas Aura. On the structure of delegation networks. In *10th IEEE Computer Security Foundations Workshop*, Rockport, MA, June 1998.

[BTS+00] Godmar Back, Patrick Tullmann, Leigh Stoller, Wilson C. Hsieh, and Jay Lepreau. Techniques for the design of java operating systems. In *Summer Usenix Technical Conference*, June 2000.

[CEE+99] Dwaine Clarke, Jean-Emile Elien, Carl Ellison, Matt Fredette, Alexander Morcos, and Ronald L. Rivest. Certificate chain discovery in SPKI/SDSI. To be published, November 1999.

[CL99] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, LA, February 1999.

[Dai99] Wei Dai. Speed comparison of popular crypto algorithms. Web Page, April 1999. http://www.eskimo.com/~weidai/benchmarks.html.

[DNS99] Domain name system security (dnssec). Web Page, March 1999. http://www.ietf.org/html.charters/dnssec-charter.html.

[EFL+98] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI examples. http://world.std.com/~cme/examples.txt, September 1998.

[EFL+99] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. Request for Comments 2693, Network Working

Group, http://www.ietf.org/html.charters/spki-charter.html, September 1999. ftp://ftp.isi.edu/in-notes/rfc2693.txt.

[EFL+00] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. Simple public key certificate. http://world.std.com/~cme/spki.txt, January 2000.

[Ell99] C. Ellison. SPKI requirements. Request for Comments 2692, Network Working Group, http://www.ietf.org/html.charters/spki-charter.html, September 1999. ftp://ftp.isi.edu/in-notes/rfc2692.txt.

[Fre98] Matt Fredette. A new sequence structure. Web Page, April 1998. http://theory.lcs.mit.edu/~cis/sdsi/fredette-ietf-41-slides.html.

[GJ98] Carl A. Gunter and Trevor Jim. Policy-directed certificate retreival. Technical Report MS-CIS-99-07, University of Pennsylvania, September 1998.

[GJ99] Carl A. Gunter and Trevor Jim. What is QCM?, August 1999.

[Gol98] Oded Goldreich. Secure multi-party computation. Working Draft, June 1998.

[Jav97] The JAR guide. Web Page, 1997. http://java.sun.com/products/jdk/1.2/docs/guide/jar/jarGuide.html.

[Jav98] JDK 1.2 security documentation. Web Page, April 1998. http://java.sun.com/products/jdk/1.2/docs/guide/security/.

[Jav99] Java 2 platform, standard edition, v1.2.2 API specification. Web Page, 1999. http://java.sun.com/products/jdk/1.2/docs/api/.

[LABW91] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. In *Proc. 13th ACM Symp. on Operating System Principles (SOSP)*, pages 165–182, October 1991. *Operating System Review*, 253(5).

[LB98] S. Liang and G. Bracha. Dynamic class loading in the Java Virtual Machine. In *Proc. OOPSLA '98*, volume 33, pages 36–44, Vancouver BC, Canada, October 1998. Proceedings of the ACM Special Interest Group on Programming Languages, ACM Press.

[Mat00] Nick Mathewson. Information flow in Java bytecodes. Master's thesis, Massachusetts Institute of Technology, 2000. Work in progress.

[MK97] D. Maziéres and M. Kaashoek. Secure applications need flexible operating systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, May 1997.

[ML97] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proc. 16th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, Saint-Malo, France, 1997.

[ML98] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 1998.

[ML00] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 2000. To appear. http://www.cs.cornell.edu/andru/pubs.html.

[Mye99a] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, San Antonio, TX, USA, January 1999. To appear.

[Mye99b] Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, January 1999.

[PKI00] Public-key infrastructure (X.509) (pkix). Web Page, February 2000. http://www.ietf.org/html.charters/pkix-charter.html.

[Riv97] Ronald R. Rivest. SEXP–(S-expressions). Web Page, May 1997. http://theory.lcs.mit.edu/~rivest/sexp.html.

[RL96] Ronald R. Rivest and Butler Lampson. SDSI – A simple distributed security infrastructure. http://theory.lcs.mit.edu/~cis/sdsi.html, April 1996.

[SGGB99] Emin Gün Sirer, Robert Grimm, Arthur J. Gregory, and Brian N. Bershad. Design and implementation of a distributed virtual machine for networked computers. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, volume 33, Charleston, SC, December 1999. ACM Operating Systems Review, ACM Press.

[SPK98] Simple public key infrastructure (spki). Web Page, February 1998. http://www.ietf.org/html.charters/spki-charter.html.