Active Certificates: A Framework for Delegation

by

Nikita Borisov

B.Math (University of Waterloo) 1998

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Master of Science

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Eric A. Brewer, Chair
Professor David A. Wagner

Fall 2002

The dissertation of Nikita Borisov is approved:

Chair Date

Date

University of California, Berkeley

Fall 2002

Active Certificates: A Framework for Delegation

Abstract

Active Certificates: A Framework for Delegation

by

Nikita Borisov
Master of Science in Computer Science

University of California, Berkeley

Professor Eric A. Brewer, Chair

In this thesis, we present a novel approach to delegation in computer systems. We exploit mobile code capabilities of today's systems to build active certificates: cryptographically signed mobile agents that implement delegation policy. Active certificates arrive at a new combination of properties, including expressivity, transparency, and offline operation, that is not available in existing systems. These properties make active certificates powerful tools to express delegation. Active certificates can also be used as a mechanism to implement complex policy systems, such as public key infrastructures; systems built in this way are easily extensible and interoperable. A prototype implementation of active certificates has been built as part of the Ninja [17] project.

Contents

List of Figures

Acknowledgments

I would like to thank all the people whose numerous suggestions have been responsible for great improvements to this thesis, including Tal Garfinkel, Oleg Kolesnikov, Mark Miller, Adrian Perrig, Dawn Song, as well as several anonymous reviewers. Special thanks to David Wagner for being a constant source of helpful ideas and comments. Finally, I want to thank my advisor, Eric Brewer, for his continued insight, direction, and support.

Chapter 1

Introduction

Delegation is an essential tool of cooperation. In computer systems, just as in real life, both responsibilities and rights are delegated to other entities in the process of cooperation. Sometimes the delegation of rights is implicit; other times it may be supported by an explicit framework. In either case, delegation of rights carries with it a risk of misuse. To minimize exposure, it is important to restrict the scope of rights transfer to only those necessary to perform the task at hand. Therefore, a secure delegation framework must be able to express highly specific and limited delegation policies. The need for such a framework is especially relevant in view of current trends, as cooperating components are distributed widely over the Internet among many mutually untrusting systems [26, 27].

Several public key infrastructures have developed a framework for secure delegation of rights in the form of delegation certificates. A principal that wishes to delegate rights to another principal issues a delegation certificate, which acts as a signed statement of policy describing what rights should be delegated and to whom. When the use of rights is requested, the access monitor interprets the certificate to make sure that the request is within the bounds of the delegation policy. It also verifies other available certificates or internal policies to ensure that the signing principal has the right to delegate the access rights in the certificate, and produces an authorization decision.

A challenge in designing such systems is the choice of policy language — it must be simple enough to be uniformly interpreted by all access monitors, and rich enough to specify a highly restrictive delegation policy. Consequently, such systems often encounter long delays in standardization and deployment, differing implementations interpret standards in incompatible ways [19], and resulting systems are frequently less flexible than many users would desire (and are difficult to change).

In the absence of a suitable framework for delegation of rights, delegation is often performed by a proxy. A proxy is a daemon, endowed with sufficient credentials to perform access as the original rights owner. The proxy performs delegation by accepting requests from others and then carrying them out on the owner's behalf. This approach is highly general, since the proxy completely mediates access. Proxies can enforce a wide range of policies and highly restrict the types of access they allow. Proxies can also be readily deployed and upgraded without any changes to the infrastructure. However, the lack of infrastructure support means that the proxy must have its own internal mechanisms to authenticate the requesters, and that the owner must find a way to keep the proxy available for use. Furthermore, lending important credentials to a daemon makes it an attractive attack target.

Active certificates are a new approach to delegation that arrives at a combination of these two solutions. An active certificate is similar to a delegation certificate, but it contains program code instead of a coded policy. The program code implements a mobile agent that acts as a delegation proxy, mediating requests and responses. However, this proxy agent is not run continuously; instead, it is instantiated by the access monitor on demand whenever access is required. The signature on the certificate allows the monitor to implicitly authenticate requests coming from the agent; thus the agent can successfully proxy the original owner's rights.

Active certificates use mobile code to bring much of the generality of proxies to a certificate-based system. The only restrictions on the possible types of policy result from any limitations of the mobile execution platform. On the other hand, since the certificate is run at the access monitor, it is able to avoid the availability requirements and security concerns of proxy-based solutions. Active certificates enjoy several properties of certificate systems that have been responsible for their popularity, such as offline operation and ease of certificate distribution.

The programmatic nature of active certificates allows them to express concepts such as composition and modularity. They can therefore be a useful policy tool even in circumstances where delegation is not required. They are sufficiently powerful to build systems such as a hierarchical public key infrastructure. The use of a general purpose language coupled with the interposition architecture make systems built with active certificates easily extensible and interoperable.

Chapter 2

Active Certificate Framework

2.1 Operation

In order to discuss the operation of active certificates, we will define an informal model of authentication. We will formalize the concepts later. The entities in our model are *Alice*, *Bob*, and a *Resource*. Alice has some certain rights to access the Resource, and she wishes to delegate some of those rights to Bob. We model interactions between Alice (or Bob) and the Resource as a flow of requests and responses; this abstraction is both natural for many types of resources and sufficiently general to represents most kinds of access. In our model, requests arrive at the Resource over some sort of authenticated channel, and are labeled with the authenticated sender; we shall represent this as, for example, "Alice: Request". The Resource applies a local policy to decide whether to authorize a request based on the sender and the contents of the request. Hence there is some X such that the request "Alice: X" will be honored by the Resource, and at the same time, "Bob: X" will be denied.

When Alice wants to delegate some of her rights to Bob using active certificates, she first needs to create a proxy that will interact with Bob and the Resource. The proxy receives Bob's requests to use the Resource and applies Alice's delegation policy to decide whether to forward the requests onto the Resource. However, instead of running the proxy herself, Alice implements the proxy in the form of a mobile agent. She then adds a digital signature to the code for the mobile agent with her private key, producing an *active certificate*. Finally, she distributes this certificate to Bob; the distribution channel is irrelevant to the operation of the certificate.

When Bob needs to use the Resource, he must present it with the active certificate. The Resource verifies the signature and then creates an instance of the mobile agent, setting it up to

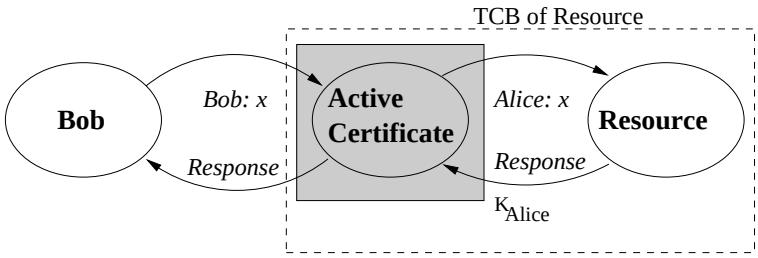

Figure 2.1: Active certificate in operation.

proxy requests from Bob. The agent receives requests from Bob and then forwards them to the Resource, assuming that they pass the delegation conditions encoded in the agent. The signature on the agent certifies its right to act on Alice's behalf; therefore, the runtime system of the Resource implicitly authenticates all requests coming from the proxy as coming from Alice. The operation of active certificates is summarized in Figure 2.1.

To illustrate the operation of active certificates, consider an example where Alice wishes to delegate some of her right to read the file `"foo"` on some file system to Bob. She first creates an agent program, which looks something like the code in Figure 2.2. She then signs the code for the agent program, creating an active certificate, and hands the certificate to Bob. When Bob wishes to access `"foo"`, he presents the certificate to the file system, which verifies Alice's signature and instantiates the agent program. Bob then sends a request "read foo" over the channel to the file system. This request is passed to the agent, authenticated as coming from Bob: "Bob: read foo". The agent will perform all the necessary checks, which in this case will succeed (note that the certificate verifies both the form of Bob's request as well as the identity of the resource) and passed it on to the file system. At this point, the request is coming from the certificate, and so it is labeled as "Alice: read foo". Hence, the request will be honored, as long as Alice has the right to access `"foo"` in the first place. In this way, the local policy of the file system is combined with Alice's delegation policy, encoded in the restrictions that are enforced by the certificate program.

2.1.1 Virtual Resources

Since Bob must send the active certificate to the Resource before he can successfully access it, a new API is needed for him to specify the certificate to use. There are several choices for this API; worth discussing is one we will call *virtual resources*. A virtual resource is an abstraction that may refer to either a standalone resource, or a resource together with an active certificate; Bob

```
processRequest(request):
 IF getCurrentDate() < "Dec 31, 2001" AND
    request.requester = "Bob" AND
    request.type = READ AND
    request.filename = "/some/pathname/foo" AND
    resource == "FileSystem"
 THEN
   return resource.processRequest(request)
 ELSE
   return Error
```

Figure 2.2: Example Active Certificate.

would use a unified way of accessing both. An active certificate would become part of a name of a virtual resource, along with fields such as resource name and network location. So, for example, virtual resources could be named as:

$$
\begin{aligned}
R_1 &:= \text{ResourceA at HostX:123} \\
R_2 &:= \text{ResourceB at HostY:456} \\
R_3 &:= R_1 \text{ with CertX}
\end{aligned}
$$

Here, R_1 and R_2 are standalone resources and R_3 is a resource accessed with an active certificate. Each name contains enough information for Bob to successfully communicate and use a Resource; at the same time, Bob can treat each name as an opaque reference, letting the infrastructure take care of the communication and certificate details. In essence, virtual resources let Bob treat the contents of the dashed box in Figure 2.1 as a black box.

This approach greatly reduces the management overhead for Bob, as he need not worry about which certificate to use at which time. Instead, upon creating an active certificate, Alice would hand him a name or a handle to a new virtual resource that uses this certificate. A virtual resource is similar to a capability, in that it combines naming and authority. However, unlike capabilities, virtual resources are not unforgeable, as Bob is free to extract a certificate from one virtual resource and use it to access a different resource, hence they are not unforgeable references. An active certificate that verifies the identity of the resource can be used as a distributed capability, although in general it may behave differently since it can also perform checks on the identity of the requester.

2.2 Properties

Active certificates share many similarities with both proxy-based and certificate-based approaches to delegation. This allows them to combine important properties of both systems. Active certificates inherit much of the expressivity and transparency of proxies. On the other hand, they can be created and distributed offline with the ease of conventional certificates. We shall discuss these properties in detail in this section.

Expressivity. Expressivity is of paramount importance in a delegation system. Delegation of rights involves weakening access control restrictions that would normally be in place; a specific, fine-grained delegation policy is needed to avoid weakening these restrictions more than necessary. Delegation proxies are highly general in the collection of policies that they are able to express, since they are interposed between Bob and the Resource and they can employ a powerful implementation language. The former allows proxies to affect the entirety of communication between Bob and the Resource; the latter allows for higher complexity of policies.

Active certificates inherit much of this expressivity. Like proxies, they are interposed on the request and response path; however, the limitations of the mobile execution platform may restrict the types of policies that are possible. Nonetheless, there exist mobile platforms that support powerful languages (e.g. Java [16]), allowing for a wide range of policies. The main source of limitations will be deliberate restrictions on the platform imposed out of concern for the security of executing potentially untrusted code.

An example of expressivity that are afforded by active certificates is their ability to understand application semantics. A language such as Java is sufficiently general to interpret the underlying meaning of requests and responses in terms of the application and perform authorization decisions on this basis. The certificate in the above example is able to understand requests for the file system well enough to identify both the file name and the operation that is being performed. In conventional certificate systems, the notion of a file name would need to be integrated with the policy language before certificates could reason about them. Active certificates, on the other hand, can easily support new kinds of applications and new kinds of policies without modifying the runtime system that interprets them.

Transparency. Active certificates retain much of the transparency of proxy-based delegation. Although the Resource does need to be aware of and process active certificates, this function can be

restricted to a small component of the runtime system. An authentication library can process active certificates and mark requests as if they were coming from Alice; the rest of the system need not know that delegation is taking place. This is important because it means that delegation can proceed without explicit support from an application running at the Resource.

Since applications are frequently used in ways that are not originally intended, interfaces provided by the application are likely to become insufficient for users' evolving needs, and upgrading such interfaces can be a slow and difficult process. Notice that although authorization policy is typically chosen by the owner of the Resource, delegation policy is chosen by Alice, who frequently doesn't have direct control over the resource. Therefore, she may have a hard time convincing the owner to invest the time and effort required to adapt the application in order to support her policy. Active certificates allow Alice to implement her delegation policy without changing the application.

Of course, help from the application can greatly simplify the task of implementing security policies. To support this, active certificates define a mechanism to let the application communicate with the certificate program; see Section 2.3.3.

Active certificates also provide transparency to the user of a resource. Bob's interactions with the Resource are the same, whether he accesses it using an active certificate or directly, with the exception that he must send the certificate to the Resource. If Bob uses the virtual resource API, that step can be made transparent as well. This makes active certificates easier to use.

Offline Delegation. The ability to perform delegation offline gives Alice more flexibility, since otherwise she must either remain online and participate in every transaction, or leave an agent with her private key to do the same. The former option limits the scope of delegation, and the latter introduces resource constraints and security concerns. Active certificates allow delegation to occur without Alice being online; indeed, Alice can create the certificates offline without ever storing her private key on a network-connected computer.

An active certificate can be seen as an offline expression of Alice's intentions; i.e. what she *would* have done had she been an online participant. To allow Alice to change her policy at a later time, it is important to associate with each certificate an expiration date, after which it is no longer valid. If more immediate revocation is desired, certificate revocation schemes (e.g. [25, 24, 28]) can be used; however, they add the requirement that either the Resource or Bob must have (at least intermittent) access to an online server.

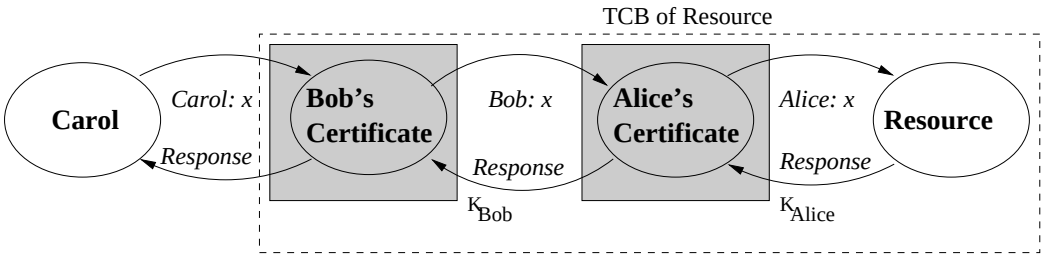

Figure 2.3: Chained Active Certificates.

2.3 Composition and Abstraction

In this section we extend the basic active certificate framework to support *composition* of certificates. Composition is motivated by the ability to re-delegate rights granted by an active certificate. The design of active certificates makes composition a natural metaphor, easily achieved with only minimal support from the infrastructure. At the same time, this simple extension enables many new uses of active certificates, going beyond simple delegation of rights. Each certificate can become a policy module, a component in a larger policy system. Most importantly, policies controlled by different entities can be combined, resulting in new expressive power. The narrow support from the infrastructure allows for greater flexibility and extensibility than existing policy systems with comparable power.

2.3.1 Re-delegation

In its simplest form, composition is a mechanism to re-delegate rights. In some cases, Bob will want to delegate rights he acquired from Alice further, for precisely the same types of reasons that made Alice want to delegate her rights in the first place. The transparency properties of active certificates make the cases of accessing resources through delegation or using them directly nearly identical from Bob's point of view, and it's natural to assume that he will want to delegate rights in both such instances.

So, to re-delegate his rights, Bob would create an active certificate, just as Alice had before. The certificate would accept requests from Carol, enforce any restrictions that Bob may wish, and forward those requests that are acceptable. For this certificate to work, Carol would need to use both Bob's and Alice's certificate when using the Resource. The Resource instantiates both certificates in a chain, as shown in Figure 2.3. All that remains necessary is to properly authenticate

the requests as they pass through the chain. Bob's certificate will see Carol's request as properly authenticated by Carol. However, when it is passed to Alice's certificate, the request will be coming from Bob's certificate and will be authenticated as coming from Bob. Hence Alice's certificate will be able to make the correct policy decision and pass the request on to the Resource, which will see it as coming from Alice. Responses will be passed back up the chain of certificates.

Notice that the operation of each certificate in the chain, viewed in isolation, is indistinguishable from a single-certificate case. The transparency afforded by interposition allows each certificate to operate the same way in both scenarios. When Bob is creating the certificate, he does not need to specify whether he himself has rights to use the Resource, or whether he acquired them via delegation. The only component that needs to be aware of composition is the certificate infrastructure at the Resource, and its task is simple — to forward requests between certificates and properly manage the authentication credentials.

It is easy to see how this composition example can be extended to chains of arbitrary length. Notice that although Alice may not wish Bob to be able to re-delegate her rights, in general she cannot prevent him from doing so. Even if the runtime system allowed Alice's certificate to differentiate between chained and non-chained operation, Bob could simply create a delegation proxy that is completely transparent to the system. Hence restrictions on re-delegation are only a minor hindrance and do not introduce extra security.

Managing chains of active certificates can be complicated, but the virtual resource API can simplify this task. Recall that we can encode an active certificate in a virtual resource name as follows:

$$R_1 = \text{Resource with CertX}$$

To re-delegate access to R_1, we can create a new virtual resource:

$$R_2 = R_1 \text{ with CertY}$$

So in the above example, Bob can give Carol a virtual resource that will include both his and Alice's certificate. Notice that the syntax we use allows Bob to delegate his rights the same way, regardless of whether R_1 is a standalone resource or one accessed through a certificate; Bob need not be aware of which is the case.

2.3.2 Policy Attributes

The re-delegation example in the previous section can be seen as combining two *policies*, expressed by different entities (Alice and Bob), into a composite policy, and using it to produce an

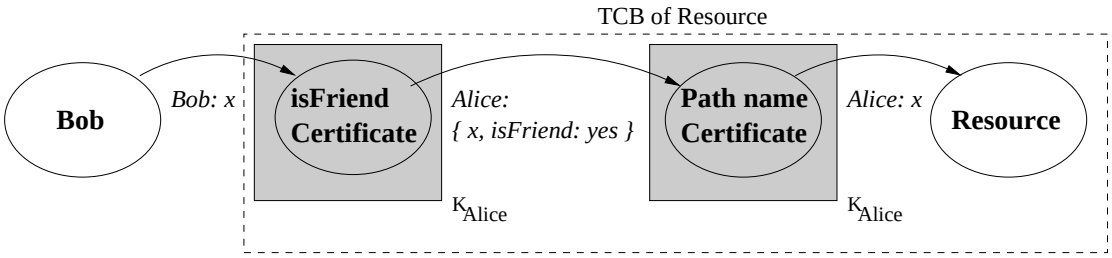

Figure 2.4: Policy Attribute Example.

authorization decision. This composition mechanism enables the opposite task of decomposing a complex policy into component policy "modules". The generality of the active certificate framework suggests applications reaching beyond simple re-delegation of rights. To consider such applications, we need to introduce the concept of *policy attributes*.

Policy attributes define a mechanism for communicating policy abstractions between certificates in a chain. They exist in the form of new fields added to requests and responses sent between certificates that are not intended to be seen by the Resource, but rather consumed by other certificates in the chain.

Consider the following: suppose Alice wants to delegate access to file `"foo"` to a group of her friends. An active certificate to enforce this policy would need to perform two checks: that the request is coming from a member of the group of friends, and that the request is of the appropriate form, i.e. accesses `"foo"`. Policy attributes allow these checks to be separated into two certificates: one that verifies membership in the "friends" group and one that verifies the request type. The former certificate would check the originator of a request, and then add an *isFriend* attribute if the membership is correct. The latter would verify that the *isFriend* attribute is present, and then proceed with the path name checks, as shown in Figure 2.4

Such decomposition allows policy components to be reused. Alice could create many policies that rely on delegating some rights to her friends, each of which could make use of the *isFriend* attribute. She can then change her set of friends without modifying any of these policies by issuing new certificates that generate the *isFriend* attribute. Decomposition also allows distribution of trust. In our example, certificates that consume the *isFriend* attribute must ensure that attributed requests are authenticated as coming from Alice, since presumably only Alice should be allowed to decide who are her friends. However, for other kinds of policies, Alice may trust someone else to define those abstractions; for example, she might want to delegate some rights to Bob's friends.

Notice that policy attributes are not part of the active certificate framework, *per se*, as they don't require infrastructure support beyond simple chaining described in the previous section. Note also that by introducing policy attributes into requests, some of the transparency present in the first composition example is lost. The "friends" certificate must be used in a chain with another certificate that understands the *isFriend* attribute. This coupling is the cost of closer and more deliberate cooperation of certificates; in effect, a new interface for communicating the *isFriend* decision is created.

2.3.3 Application Policy Adapters

Although typically policy attributes are consumed by chained active certificates and are not passed onto the application, some applications may wish to accept attributed requests in order to facilitate policy implementation. For example, it may be easier for the file system to identify requests that are read-only internally; in this case, it may choose to accept attributed requests with a *readOnly* attribute, and refuse to carry out any modification operations for such requests. Then the example certificate from Figure 2.2 could be rewritten to allow any requests but add a *readOnly* attribute.

This example is another demonstration of the trade-off between transparency and ease of policy implementation. The *readOnly* attribute obviates the need for a certificate to fully understand the semantics of the requests sent to the file system. However, it is only possible to use this attribute in a file system that anticipated the need for read-only access, coupling the certificate and application implementation. For policies based on attributes not explicitly supported by the application, it is necessary to identify such attributes within the certificate itself and make the policy decision transparent to the application. The important point is that active certificates allow for systems at different points in this trade-off.

2.3.4 Hierarchical PKI

To illustrate the power of composition and abstraction on a more complex example, we proceed to build a hierarchical public key infrastructure (PKI) using active certificates. A PKI uses certificates to create associations between names, or principals, and public keys. Applications can then use such associations to make policy decisions based on principals and not public keys.

To represent principals in the active certificate setting, we use a policy attribute — a field called *name*. A request with the field *name* set to "X" signifies that the request originates from the

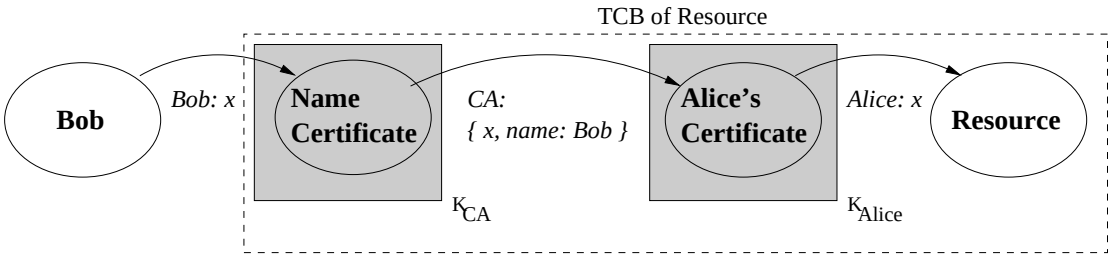

Figure 2.5: Active PKI.

principal with that name. Active certificates (or applications, for that matter) receiving such requests can then make appropriate policy decisions based on the *name* field rather than on the public key of the requester.

Of course the authenticity of the principal name must be verified for this to be useful. In a traditional PKI, a certificate authority (CA) is trusted to make assertions about what keys belong to which principals. We can express the same trust using active certificates; the contents of the *name* field are only trusted if the request is authenticated by the CA. So, a certificate that wants to delegate some rights to the principal ”Bob” would perform the following checks:

```
IF request.requester == CA-key AND
   request.name == "Bob"
THEN
   // do something that Bob is authorized to do
```

But how is Bob able to issue requests authenticated with the CA's key? To enable him to do so, the CA issues an active certificate that accepts any request authenticated by the key K_B that the CA knows belongs to Bob and adds the *name* field of ”Bob” to the request before passing it on. Adding this “name certificate” to the front of a chain will allow Bob to send requests that will be authenticated with his principal name by the CA, as shown in Figure 2.5.

The above example describes a flat hierarchy, but it is easy to introduce subauthorities. Instead of issuing a “name certificate” to Bob, the root CA can issue a delegation certificate to a CA lower down in the hierarchy. The delegation certificate will accept any request authenticated by the lower CA, as long as the *name* attribute is within a domain of power for the subauthority. The sub-CA would then be the one issuing a “name certificate” to Bob. Chaining the name and delegation certificates together once again allows Bob to authenticate himself with his principal name.

Notice that in this case there is a one-to-one correspondence between conventional certifi-

cates used in a hierarchical PKI to create the name-key bindings for “Bob” and active certificates. In this case, each active certificate encodes the operational semantics of its passive counterpart. The important difference is that the interpretation of the name-binding policy is performed by the certificates themselves rather than by a certificate engine at the Resource.

2.3.5 Discussion

The fact that it was easy to build a hierarchical PKI out of active certificates speaks to their generality. The resulting system not only duplicates many of the features of conventional PKIs, it also has interesting new properties, such as ease of interoperation, extensibility, and the potential for a more secure TCB.

Interoperation is an important requirement of PKIs: companies frequently use cross-certification [2] to connect their corporate infrastructures. However, both systems must be able to understand each other’s certificate format, namespaces, etc. The use of active certificates provides an easier way to connect two hierarchies; all that is necessary is an active certificate chaining trust from a node on the first hierarchy to the root (or some other node) on the other. The certificate acts as a “bridge” between the two systems, performing any necessary namespace translations and other modifications to make the systems compatible. Its role here can be compared to an active proxy [11] protocol adapter.

Active certificates also leave plenty of room for extension. A general purpose language allows any computable function to be used as a policy, and the interposition architecture avoids any limitations of an explicit interface with the Resource. As a result, it is possible to create certificates expressing new types of policies and integrate them with an existing system. It is even possible to evolve policy abstractions over time, using adapter certificates to provide backward compatibility. In contrast, conventional certificate systems are difficult to upgrade, since all the libraries that interpret certificates must be replaced, and back-wards compatibility may be difficult to achieve.

The active certificate architecture may also help to make the trusting computing base more secure. Complex certificate libraries can be removed from the TCB and replaced by a general-purpose language interpreter. An interpreter for an established language is likely to be more mature than any given certificate library. Further, there may be incentive for commercial vendors to offer the core of their system as open-source, since most of a given solution’s value lies in management subcomponents, which, while essential to operation, are not security critical. In this way, they can provide their customers with a higher assurance of security than is possible today.

Unfortunately, active certificates cannot duplicate all of the features of modern PKIs. Since it is undesirable to allow a mobile agent to open new network connections, it is difficult to implement certificate revocation lists using active certificates (although the "bill-of-health" certificates proposed by Rivest [31] could be supported). It is still possible to implement revocation lists in the runtime system, but that solution lacks the advantages of active certificates such as the easy ability to change algorithms. Automated certificate management is also complicated by the fact that it is difficult to tell whether a certificate will be helpful in an authentication transaction without executing the program code contained within. And despite a smaller TCB, running untrusted mobile code, even in a restricted environment, is still considered a risk today. Nonetheless, active certificates present an interesting, if not yet practical, new direction for implementing PKIs and other policy systems.

2.3.6 Further Modularity

Abstractly, a chain of certificates acting to produce a composite policy can be viewed as an instance of a particular design pattern — a set of modules arranged linearly, with communication restricted to adjacent modules only. One can envision other configurations of policy modules producing useful results, just like other design patterns are useful to structure software. A fully general compositional model would call for sending a collection of certificates along with a configuration graph representing the communication links between them to the Resource. Such a model would be able to produce more powerful composite policies and allow for greater modularity.

However, the are a few good reasons for the simpler point in the design space — a chain of certificates. Chained composition is natural given the filter-like nature of active certificates. More importantly, it allows transparency to be preserved; as remarked above, chained certificates can operate without knowledge that they are part of a chain. This transparency can be sacrificed in order to achieve better interplay between certificates in a chain, but the base case of transparent operation is preserved.

Additionally, the general model requires the requester to perform a form of meta-programming, using individual active certificates as building blocks and combining them into a configuration. Although it is possible (and potentially useful) to do this manually in some cases, performing such meta-programming in an automated or even a semi-automated fashion is a highly complex problem. Of course, even in the chained scenario it is necessary to come up with a mechanism to select the relevant certificates and chain them together properly; but there is more hope that, at least

	Alice		Resource
1	$R(x)?$		
2		$\xrightarrow{R(x)?}$	
3			$A\ says\ R(x)?$
4			Auth check
5			execute $R(x)$
6			$R(x) = y$
7		$\xleftarrow{R(x)=y}$	
8	$R\ says\ R(x) = y$		

Figure 2.6: Simple deduction.

within restricted contexts, it will be possible to do so automatically.

2.4 Security Analysis

In this section, we formally model the operation of active certificates using a belief logic defined by Abadi, Burrows, Lampson, and Plotkin [1]. Formal methods have been used to examine and formally verify a large number of security systems; they have helped to identify problems and hidden assumptions in many. Even outside the context of proofs of security, a formal specification of a system can often lead to a better understanding of its properties. We will therefore proceed to describe the operation of active certificates using the logic.

First, let's look at how we model ordinary access to the Resource. As shown in Figure 2.6, Alice will ask the Resource to perform a request x, which we will represent as $R(x)?$ (1). She then sends it to the Resource over an authenticated channel (2). The Resource now believes that $A\ says\ R(x)?$ (3). It then performs a check to see that Alice is authorized to request x (4) and executes the request (5), producing the response y (6). Finally, the response is sent to Alice (7).

The deduction in the case of delegated access is more complicated. First, we need to model Alice's certificate. It is signed by Alice's key, i.e $cert = \{...\}_{K_A}$. The contents of the certificate represents Alice's policy delegating access to Bob, so it may be tempting to say $cert = \{B \Rightarrow A\}_{K_A}$, where $\Rightarrow$ is the "speaks for" operator, defined as:

$$(B \Rightarrow A) \supset ((B\ says\ s) \supset (A\ says\ s))^1 \tag{2.1}$$

However, this would be incorrect, since that statement gives B unrestricted ability to do anything

[1] Similar to the notation used by Abadi et al, we use $\supset$ to represent the logical containment relation; $t \supset s$ means that if t then s.

A is allowed to do, as opposed to only the things allowed by the certificate program. We must therefore examine the operation of active certificates more closely.

The active certificate contains a program, which we will call F. The program processes authenticated requests such as "Bob: x" and produces a new request x'. To model this, we define an abstract function $\hat{F}$, which operates on statements in the logic, and represents the operation of the program F:

$$F(\text{“Bob: } x\text{”}) = x' \supset \hat{F}(B \; says \; R(x)?) = R(x')? \tag{2.2}$$

The abstract function $\hat{F}$ will, in turn, define a new principal $P_{\hat{F}}$, with the rule:

$$t \supset (P_{\hat{F}} \; says \; \hat{F}(t)) \tag{2.3}$$

In other words, $P_{\hat{F}}$ says whatever the program outputs. Note that the t must be held true in order for the rule to apply; in other words, the input to the certificate needs to be valid in order for the output to reflect Alice's true position. Since the entire deduction is performed from the point of view of the Resource, it can be trusted to provide only input it believes to be true to the certificate. Now we can interpret the certificate as a statement that the program within is acting on Alice's behalf:

$$\{F, P_{\hat{F}} \Rightarrow A\}_{K_A} \tag{2.4}$$

Let us use these rules in an example, shown in Figure 2.7. Line 1 shows the representation of Alice's certificate from rule (2.4). Bob sends this certificate to the Resource (2), where it is processed by the active certificate infrastructure (3). Unwrapping the signature on the certificate, we deduce (4) from (3). We will assume that the Resource knows Alice's public key, i.e. that $K_A \Rightarrow A$, and use it to deduce (5). We need another assumption, that $A \; controls \; (X \Rightarrow A)$, which means that Alice has the authority to delegate her own right — this is not implicit in the logic, but it is necessary in the active certificate framework. Applying this, we arrive at (6), where Alice's authority has been delegated to the certificate program.

Now Bob wants to make a request x (7) so he sends it to the Resource (8). It's received first by the active certificate infrastructure (9), The infrastructure passes the request to the certificate, evaluating F to obtain a new request $x' = F(\text{“Bob: } x\text{”})$. Using the abstract function $\hat{F}$ to model this operation, we have that $\hat{F}(B \; says \; R(x)?) = R(x')?$ (2.2). Applying rule (2.3) and evaluating $\hat{F}$, we obtain (10,11). Since $P_{\hat{F}} \Rightarrow A$ (6), the request is authenticated as coming from Alice (12). The result of this deduction is then passed on to the Resource (13,14). Note that since the infrastructure is part of the Resource's TCB, the statement is taken as truth, as opposed to $X \; says \; A \; says \; R(x')?$,

	Bob		AC infrastructure	Resource
1	$\{F, P_{\hat{F}} \Rightarrow A\}_{K_A}$			
2		$\xrightarrow{\{F,P_{\hat{F}} \Rightarrow A\}_{K_A}}$		
3			$B\ says\ \{F, P_{\hat{F}} \Rightarrow A\}_{K_A}$	
4			$K_A\ says\ F, P_{\hat{F}} \Rightarrow A$	
5			$A\ says\ P_{\hat{F}} \Rightarrow A$	
6			$P_{\hat{F}} \Rightarrow A$	
7	$R(x)?$			
8		$\xrightarrow{R(x)?}$		
9			$B\ says\ R(x)?$	
10			$P_{\hat{F}}\ says\ \hat{F}(B\ says\ R(x)?)$	
11			$P_{\hat{F}}\ says\ R(x')?$	
12			$A\ says\ R(x')?$	
13			$\xrightarrow{A\ says\ R(x')?}$	
14				$A\ says\ R(x')?$
15				Auth check
16				execute $R(x')$
17				$R(x') = y'$
18			$\xleftarrow{R(x')=y'}$	
19			$R(x') = y'$	
20			$P_{\hat{F}}\ says\ \hat{F}(R(x') = y')$	
21			$P_{\hat{F}}\ says\ R(x) = y$	
22			$A\ says\ R(x) = y$	
23		$\xleftarrow{A\ says\ R(x)=y}$		
24	$R\ says\ A\ says\ R(x) = y$			

Figure 2.7: Deduction with an active certificate.

for some principal X. The Resource performs an authentication check (15), executes the request (16), and produces the result y' (17).

Unlike the previous case, the response is not sent directly to Bob. Instead, it is returned back to the infrastructure (18,19), which passes it back to the program F, to obtain a new response y. We will once again use $\hat{F}$ to model this operation, applying rule (2.3) to pass $R(x') = y'$ as an argument to $\hat{F}$. We create a rule similar to (2.2):

$$F(\text{“result: } y'\text{”}) = y \supset \hat{F}(R(x') = y') = \{R(x) = y\}^2 \tag{2.5}$$

Evaluating $\hat{F}$ in this way, we obtain (21). Invoking $P_F \Rightarrow A$ once again, the response is authenticated by Alice (22). Finally, this response is sent to Bob (23,24).

Notice that the final statement deduced by Bob is weaker than either $R\ says\ R(x) = y$ or $A\ says\ R(x) = y$. However, it is the proper interpretation of the result. The response is computed by Alice's agent, and not by the Resource; however, neither Alice nor Bob can monitor the execution of the agent, and so the Resource can potentially modify the response. Hence both Alice and the Resource must be trusted in order to trust the result. This situation is analogous to the use of proxies, where it's easy to see that a result must be interpreted as $A\ says\ R\ says\ R(x) = y$.

This weak statement is sufficient in many common instances of delegation. Consider, for example, the case where Alice gives Bob the right to check her email while she is away, or where Alice shares access to some of her files with Bob because they are working on a project together. In both cases, it does not make sense for Alice to try to deceive Bob by returning malicious results in her certificate; delegation here is used as a tool for cooperation, which requires a certain degree of mutual trust to begin with. Problems arise when Bob's ability to use the Resource properly is not directly beneficial for Alice; for example, if she is selling her access to the Resource to Bob. In such cases, Bob may want to examine the operation of the active certificate in order to derive a stronger statement on the result. However, in general, properties of the certificate may be undecidable given the program code; providing better support for auditing is the subject of future work.

We also could have modeled active certificates using the restricted delegation primitive defined by defined by Howell and Kotz [22]. $B \overset{T}{\Rightarrow} A$, or "B speaks for A regarding T" means that B has the authority to act on the behalf of A for any action contained in the set T. An active certificate would then be modeled as:

$$K_A\ says\ \forall B.B \overset{\{F(B\ says\ x)|\forall x\}}{\Longrightarrow} A$$

[2]For simplicity, we ommited the necessary assumption about the nature x and x', namely, that $x' = \hat{F}(B\ says\ x)$.

However, this expression is awkward and difficult to understand, since an active certificate defines both the potential recipients of delegated rights and the set of allowed actions implicitly (and in general, these sets are not computable).

Chapter 3

Implementation

We have built a prototype implementation of active certificates as part of the service call mechanism in Ninja [17]. The Ninja project aims to serve as a platform for building a distributed services infrastructure, with a focus on service composition. This section discusses the details of our implementation.

3.1 Service Calls

Service calls in Ninja are represented as typed messages, or *tasks*. A task is implemented as a Java object. Java [16] is used in Ninja because it provides a rich type hierarchy, platform independence, and automated memory management. When a client wishes to send a task to a service, it calls the `handleTask` method on a stub object for the service. The task is serialized and sent to the service for processing. Responses, or *completions*, are returned in the form of typed messages as well.

3.2 Certificate Implementation

Because of its support for code mobility and restricted program execution, we use Java as the language for active certificates in our implementation. This choice also simplified the integration of active certificates with the rest of the Ninja framework.

An active certificate implements the `ActiveCertIF` interface, which has two methods: `init`, which accepts a reference to a stub object for the downstream service, and `handleTask`, which performs a policy decision on incoming tasks and sends tasks to the service, using the `han-`

`dleTask` method on the stub object. It also processes the responses received from the service, potentially modifying them before returning them to the user. The interposition of the active certificate is transparent to both the service and the client.

The `ActiveCertIF` interface is well suited for chaining. An active certificate that is part of a chain gets a reference to another certificate, and not to a service stub, as the argument to its `init` method. Therefore, calls to `handleTask` pass the tasks to the next certificate in the chain. In this way, chaining is also transparent to all the certificates.

3.3 Authentication

We will not discuss the authentication protocol used by Ninja, other than to say that it is similar in spirit to TLS [9], and can be modeled as a secure channel. The result of authentication is expressed as message metadata: each typed message includes an `authKey` field that is set by the infrastructure to be the public key of the authenticated originator of the message. When a service receives a message M with `authKey` $= K$ it can derive the statement K *says* M.

Active certificates are implemented by changing the `authKey` field of messages. When an active certificate receives a message from a client, the `authKey` is set to the client's public key. When it calls `handleTask`, the resulting message that is sent to the service has its `authKey` set to the signer of the certificate. This makes the service behave as if the principal who signed the certificate was interacting with it directly.

3.4 Certificate Format

An active certificate consists of four fields: the certificate program, represented by the bytecode for a class that implements the `ActiveCertIF` interface, a parameter object (see below), an expiration date, and the public key of the signer of the certificate. The final certificate consists of a byte array containing the serialized version of these fields and a signature over the byte array using the specified public key.

When the infrastructure receives an active certificate from a client, it first verifies the signature. If the verification succeeds, a special class loader is used to load the implementation of the certificate with restricted permissions. Then the infrastructure creates an instance of the certificate class, passing the parameter object to the constructor. Finally, it installs the certificate in the message path between the client and the service by calling the certificate's `init` method.

The parameter object allows the reuse of a single class implementing an active certificate program in multiple certificates. For example, a blanket delegation certificate that delegates all possible rights to key X (for a limited time) might store the value of X in a parameter object. This allows the same implementation to be reused to perform a similar delegation to key Y. In the absence of a parameter field, X or Y would have to be specified as a static field in the class, requiring two separate classes for the two certificates.

3.5 Principal Names

The Ninja infrastructure does not have an inherent understanding of principal names; it uses public keys to identify participants. To support named principals, we implemented a hierarchical PKI as described in Section 2.3.4. We created a special wrapper message type called `MessageFrom`, which contains a name attribute and a message. The semantics of a message of the form `MessageFrom`(N, M) can be modeled as $N\ says\ M$. However, unlike the `authKey` field, the `name` field in a `MessageFrom` object is not verified by the infrastructure, so a service must be careful to accept such messages only from trusted sources. In our prototype hierarchical PKI each service knows the key of the root authority and only accepts `MessageFrom` objects authorized by that key.

The root authority issues delegation certificates that accept `MessageFrom` messages authenticated by its subauthorities, checking that the name field is within the jurisdiction of each authority. The subauthorities, in turn, issue certificates that accept messages sent by a particular public key and create a `MessageFrom` message that includes the corresponding name. An example of such a certificate is shown in Figure 3.1. When a client accesses a service, it sets up a chain of active certificates leading up to the root, and then proceeds to send requests. The first certificate in the chain will create a `MessageFrom` message, which will be accepted by the certificates that follow it in the chain. Finally, the message will arrive at the service authenticated by the root authority. The service can then perform a decision based on the now-authenticated name field.

3.6 Applications

We built a certificate directory service, which is used to look up active certificates by name. Clients use the certificate service to look up their name certificates, which they use to authenticate themselves to services. The directory service only accepts updates from the root authority.

```
public class NameCertificate
  implements ActiveCertIF {
  private PublicKey key;
  private Name name;
  private ServiceIF service;
  // ...

  void handleTask(Task task) {
    if (task.authKey.equals(key)) {
      service.handleTask(
        new MessageFrom(name, task), ...)
    } else {
      // error
    }
  }
}
```

Figure 3.1: A Name Certificate.

However, we use a delegation certificate issued by the root that implements the following policy: any client that can authenticate itself under name N is allowed to update the certificate stored for that name. This allows clients to update their own entries in the directory, but not those of others. Note that this policy was implemented by the root authority without modifying the directory service, as would be necessary in a conventional system.

We also experimented with using active certificates to delegate access to the Ninja Jukebox [14] and NinjaMail [36] services. We successfully implemented certificates with policies to provide read-only access to individual song preferences to a "collaborative DJ" service. In NinjaMail, we use active certificates to grant a procmail-like [35] service the ability to examine message headers and automatically file messages into folders. In this way, a compromise of the procmail service will have limited impact on the mail system; in particular, mail cannot be deleted.

3.7 Discussion

Java Platform. Our experience using Java has been generally positive. The Java 1.2 Security Architecture [15] is a big improvement over the previous version; restricting the execution of active certificates was quite natural. It is impossible, however, to enforce resource limits such as CPU time or memory usage on the certificate in our prototype. We are hoping to benefit from research

on resource limits in Java [13, 33], and provide better resource monitoring for active certificates as well as other components of the Ninja framework.

The method of creating Java active certificates presented a barrier to automated certificate generation. To create a certificate, it is necessary to locate the bytecode for its implementation; in an interactive setting this is done by reading the corresponding `.class` file off the file system. However, Ninja services are shipped as mobile code to their execution environments and frequently do not have access to the file system. To let a service create new certificates, it is necessary to include a static parameter to the service that contains the bytecode of the certificate implementation. This approach is functional, but it requires administrative overhead to set and update the service parameter. If the bytecode implementation of a class visible at runtime could be obtained through reflection, automatic generation of certificates would be more natural.

Message Interfaces. The use of typed message interfaces helped make active certificates simpler and cleaner. The previous version of the Ninja platform [18] used RMI-style interfaces, which were a collection of method signatures (i.e. a Java interface). To interpose on a service that uses a method interface it is necessary to provide an implementation of each method. Message interfaces, on the other hand, allow a certificate to operate as a message filter, with only partial or no knowledge of the interface. This makes expressing "vertical policies", which are the same for every request type, very natural. (An example of such vertical policy is a name certificate described in Section 3.5.) Request-dependent "horizontal" policies can also be easily represented using message interfaces by branching on the message type. Even in this case, message interfaces have the advantage of being able to adapt to an evolving service interface by denying any unrecognized request types. A policy that has both horizontal and vertical components (this will be true of many policies in practice) is also natural to represent in message interfaces; method interfaces on the other hand would require code duplication to implement the vertical components of policies.

This experience suggests that in other systems that use messages to encode remote calls (e.g. RPC over SOAP [7]), active certificates should be implemented as message filters instead of RPC wrappers.

Chapter 4

Other Work

4.1 Related Work

A number of certificate systems have attempted to incorporate the concept of delegation. For example, proxy certificates [34] are a proposed way to add delegation to X.509 [8]; SPKI [10] uses delegation as a central concept in its operation. Both systems include a mechanism to restrict delegation: proxy certificates allow one to specify a restriction in a (yet-to-be-specified) policy language, and SPKI supports application-specific restriction tags. In both cases, further standardization on application semantics is required, and this process must be repeated for each new application domain.

Several systems have used a general-purpose programming language to specify policy. PolicyMaker [4] is a system that manages collections of assertions, which can include arbitrary programs in a safe version of AWK, and computes policy decisions on their basis. Proof-Carrying Certificates [3] use proofs written in Twelf [29], which is a powerful, if not general-purpose, language. Both systems, in typical usage, lack the transparency of active certificates. PolicyMaker applications must define security attributes that are relevant and specify local policy in terms of them. Proof-Carrying Certificates must prove an application-dependent theorem, with local policy represented by axioms. However, a variant of PolicyMaker could be used to produce a system similar to active certificates, wherein an entire request is passed as a query to the policy management system, and a language appropriate for parsing such requests is used to define assertions. Such a system would lack the full proxying aspects of active certificates, and have a less general area of application than PolicyMaker, but it would combine a number of their strengths.

Proxy-based solutions can be used to implement general delegation policies with complete

transparency. Several projects have used proxy technology to perform security adaptation [32, 12]. However, maintaining an online proxy imposes significant computational, connectivity, and management overhead on its owner. In addition, prevalence of such proxies might put excessive bandwidth requirements on the infrastructure because of the resulting inefficient routes. Most importantly, the proxy has to maintain its owner's private key, which makes it an attractive attack target.

Active certificates avoid all of these pitfalls by executing at the Resource. They are, however, less expressive than proxies, since they are instantiated only temporarily during access to the Resource, and cannot maintain persistent state. To implement policies that require persistent state, a hybrid solution is possible, wherein an online proxy stores the persistent state necessary and an active certificate is used to specify policy with input from the proxy. The proxy does not need to store its owner's public key, instead it can have its own key recognized by the certificate. Such a solution combines the expressive power of proxies with the security advantages of active certificates, since the proxy is only trusted to maintain correct state, but not to authorize use of Alice's rights.

4.2 Future Work

Although active certificates provide a very powerful delegation mechanism, it is important to be able to manage certificates effectively in order to exploit their full potential. Instead of specifying an active certificate directly or indirectly (through a virtual resource name) during access, it may be desirable to have an automated search mechanism to find a sequence out of a pool of available certificates that will allow Bob to use the Resource. There has been much research into the problem of deciding authentication [20, 23, 5, 6] with varied results; however, it should be clear that the use of programs to specify policies makes this problem undecidable. Nonetheless, we hope to be able to attach attributes to certificates to make searches for a trust path feasible in practice, by trying to express which certificates may be useful to solve a particular authentication problem. For example, if a higher-level policy language is translated into active certificates, such attributes could take the form of the original high-level language source. This would allow active certificates to be managed in the same way as conventional certificates.

Such "translation annotations" can also serve to check certain certificate properties, if it is possible to prove that the certificate code is indeed a semantically equivalent translation of the annotation [30]. Such a proof would ensure that the certificate program is bound by any restrictions that are inherent in the source language. For example, a translation from a policy language that

has a bounded execution time can ease concerns of resource misuse by the certificate. We are also investigating other properties that may be useful to prove about active certificates, and other ways of proving them.

Finally, we are evaluating the performance impact of using active certificates. One promising feature of active certificates is that complex functions such as interpreting high-level policies or finding a trust path are shifted from servers onto clients; this allows us to exploit the vast disparities in the aggregate computing power of services and their large user bases to improve performance.

Chapter 5

Conclusions

In this thesis we presented a novel approach to delegation based on active certificates. It combines the strengths of previous approaches, including expressivity, transparency, offline operation, and convenience; these features make active certificates useful tools for expressing delegation. We also explained how to use active certificates as a platform to build larger systems; this approach has important advantages such as extensibility. We performed a formal security analysis of active certificates and built a prototype implementation validating our techniques. Active certificates are an exciting new direction in delegation and present many directions for further research.

Bibliography

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

[2] American National Standards Institute. Public key cryptography for the financial service industry: Certificate management. ANSI X9.57-1997, 1997.

[3] A.W. Appel and E.W. Felten. Proof-carrying authentication. In *5th ACM Conference on Computer and Communications Security*, pages 52–62, Singapore, November 1999. ACM Press.

[4] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, May 1996. IEEE Computer Society Press.

[5] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance checking in the PolicyMaker trust management system. In Hirschfeld [21], pages 254–274.

[6] D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R.L. Rivest. Certificate chain discovery in SPKI/SDSI. `http://theory.lcs.mit.edu/~rivest/publications.html`.

[7] WWW Consortium. Simple object access protocol (SOAP) 1.1. `http://www.w3.org/TR/SOAP/`.

[8] Consultative Committee on International Telegraphy and Telephony. *Recommendation X.509: The Directory—Authentication Framework*, 1988.

[9] T. Dierks and C. Allen. The TLS prtocol version 1.0. RFC2246, January 1999.

[10] C.M. Ellison, B. Frantz, B. Lampson, R. Rivest, B.M. Thomas, and T. Ylonen. SPKI certificate theory. Internet Draft, March 1998. Expires: 16 September 1998.

[11] A. Fox, S. D. Gribble, Y. Chawathe, and E. A. Brewer. Adapting to network and client variation using active proxies: Lessons and perspectives. *Special Issue of IEEE Personal Communications on Adaptation*, August 1998.

[12] A. Fox and S.D. Gribble. Security on the move: Indirect authentication using Kerberos. In *2nd ACM International Conference on Mobile Computing and Networking*, November 1996.

[13] G.Back, W.C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, October 2000.

[14] I. Goldberg, S. Gribble, D. Wagner, and E. Brewer. The Ninja Jukebox. In *Second USENIX Symposium on Internet Technologies and Systems (USITS)*, Boulder, CO, October 1999.

[15] L. Gong. *Inside Java 2 Platform Security*. Addison-Wesley, June 1999.

[16] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1997.

[17] S. Gribble, M. Welsh, R. von Behren, E. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Josheph, R. Katz, Z. Mao, S. Ross, and B. Zhao. The Ninja architecture for robust Internet-scale systems and services. *Special Issue of Computer Networks on Pervasive Computing*, March 2001.

[18] S.D. Gribble, M.Welsh, E.A. Brewer, and D.Culler. The MultiSpace: An evolutionary platform for infrastructural services. In *Proceedings of the 1999 USENIX Annual Technical Conference (USENIX-99)*, pages 157–170, Berkeley, CA, June 6–11 1999. USENIX Association.

[19] P. Gutmann. X.509 style guide. `http://www.cs.auckland.ac.nz/~pgut001/pubs/x509guide.txt`, October 2000.

[20] M.H. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, 1976.

[21] R. Hirschfeld, editor. *Financial Cryptography*, Anguilla, British West Indies, February 1998.

[22] J. Howell and D. Kotz. A formal semantics for SPKI. In *6th European Symposium on Research in Computer Security*, pages 140–158, 2000.

[23] A.K. Jones, R.J. Lipton, and L. Snyder. A linear time algorithm for deciding security. In *17h IEEE Symposium on the Foundations of Computer Science*, pages 33–41, 1976.

[24] P. Kocher. On certificate revocation and validation. In Hirschfeld [21], pages 172–177.

[25] S. Micali. Efficient certificate revocation. Technical Memo MIT/LCS/TM-542b, Massachusetts Institute of Technology, Laboratory for Computer Science, March 1996.

[26] Microsoft. Microsoft .NET. `http://www.microsoft.com/net/`.

[27] Sun Microsystems. Sun Open Net Environment (Sun ONE). `http://www.sun.com/software/sunone/`.

[28] M. Naor and K. Nissim. Certificate revocation and certificate update. In *Proceedings of the 7th USENIX Security Symposium*, pages 217–228, Berkeley, January 26–29 1998. Usenix Association.

[29] F. Pfenning and C. Schurmann. System description: Twelf — a meta-logical framework for deductive sstems. In *16th International Conference on Automated Deduction (CADE-16)*, Trento, Italy, June 1999.

[30] A. Puneli, M. Siegel, and E. Signerman. Translation validation. In *4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lisbon, Portugal, March 1998.

[31] R. Rivest. Can we eliminate certificate revocation lists? In Hirschfeld [21], pages 178–183.

[32] S. Ross, J. Hill, M. Chen, A. Joseph, D. Culler, and E. Brewer. A composable framework for secure multi-modal access to Internet services from Post-PC devices. In *Third IEEE Workshop on Mobile Computing Systems and Applications (WMCSA), to appear*, Monterey, CA, 2000.

[33] A. Rudys, J. Clements, and D.S. Wallach. Termination in language-based systems. In *Network and Distributed Systems Security Symposium '01*, 2001.

[34] S. Tuecke. Internet X.509 public key infrastructure proxy certificate profile. Internet Draft, 2001.

[35] S.R. van den Berg. Procmail - autonomous mail processor. `http://www.procmail.org/`.

[36] J.R. von Behren, S. Czerwinski, A.D. Joseph, E.A. Brewer, and J. Kubiatowicz. NinjaMail: The design of a high performance clustered, distributed e-mail system. In *First International Workshop on Scalable Web Services*, Toronto, Canada, August 2000.