Active Certificates: A Framework for Delegation

Nikita Borisov Eric Brewer
University of California, Berkeley
E-mail: {nikitab,brewer}@cs.berkeley.edu

Abstract

In this paper, 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.

1 Introduction

Delegation is an essential tool of cooperation. In computer systems, components frequently need to delegate rights to other components in order for cooperation to succeed. Delegation of rights always carries with it a risk of misuse; therefore, it is important to minimize exposure by delegating the precise set of rights necessary for the task at hand. This security concern is especially relevant in view of the current trends, as cooperating components are distributed widely over the Internet among many mutually untrusting systems [26, 27].

The issue of delegation has been addressed in several public key infrastructures by introducing the concept of a delegation certificate. 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 access is required, the certificate is interpreted by the access monitor, which combines the delegation policy contained in the certificate with internal policies, as well as any other certificates available, 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.

Another mechanism to perform delegation is 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 and can therefore enforce a wide range of policies. 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 introduces security concerns.

Active certificates are a new approach to delegation that arrives at a combination of these two solutions. An active certificate is a special type of delegation certificate that contains the program code implementing a mobile agent. This mobile agent acts as a delegation proxy, mediating requests and responses. However, the agent is instantiated by the access monitor whenever use of rights is requested. Because the certificate is signed, requests coming from the agent are implicitly authenticated as coming from the signer; 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.

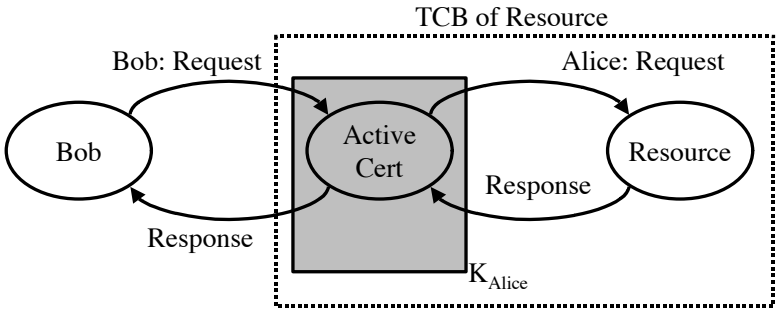

Figure 1. Active certificate in operation.

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 complex enough 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.

The rest of the paper is organized as follows: the next section describes active certificates and their properties. Section 3 explains how to use active certificates to build complex systems. Section 4 formally examines the security of active certificates. Section 5 describes our implementation of active certificates. Sections 6 and 7 discuss related and future work. Finally, Section 8 concludes.

2 Active Certificates

2.1 Operation

In this paper, we will discuss delegation in terms of *Alice*, who has some rights to access a *Resource*, and wishes to delegate a portion of those rights to *Bob*. We will first describe the case when Alice accesses the Resource without delegation. It will be convenient to model her interactions with the Resource as a flow of requests and responses; this abstraction is sufficiently general to represent most kinds of access. In our model, requests from Alice arrive at the Resource over some sort of authenticated channel, and are labeled with the authenticated sender. We will write "Alice: Request" to represent this. The Resource applies a local policy decision to decide whether Alice is authorized to perform the specified request and sends back an appropriate response.

When Alice wants to delegate some of her rights to Bob, she 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 signs the code for the mobile agent with her private key, producing an *active certificate*. This certificate is then distributed to Bob.

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 proxy requests from Bob. 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 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 the file system to Bob. She first creates an agent program, which looks something like the code in Figure 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's requests to access the file are sent to the certificate, which verifies all the necessary conditions and forwards the them onto the file system. Since the forwarded requests are identified as coming from Alice, the file system policy allows exactly those actions that Alice is authorized to perform. However, the checks performed by Alice's program restrict Bob's actions further, enforcing Alice's delegation policy.

2.2 Properties

Active certificates share 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

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

Figure 2. Example Active Certificate.

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 (potentially deliberate) 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.

For example, Java and similar languages are clearly sufficiently general to understand the application semantics of requests and responses. 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 eventually become insufficient for users' evolving needs, and upgrading such interfaces can be a slow and difficult process. Notice that in the case of delegation, the policy is chosen by Alice and not the owner of the Resource, as is the case with authorization. Therefore, Alice may have a hard time convincing the owner to invest the 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 3.3.

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.

3 Composition and Abstraction

Because active certificates are interposed on the request/response stream, there is a natural way to compose them. Such composition enables further re-delegation of rights with additional restrictions. More importantly, it allows decomposition of a complex policy into smaller policy modules. The use of a general purpose language makes it possible for active certificates to define abstractions. Using these two techniques, it is possible to build complex policy systems based on active certificates.

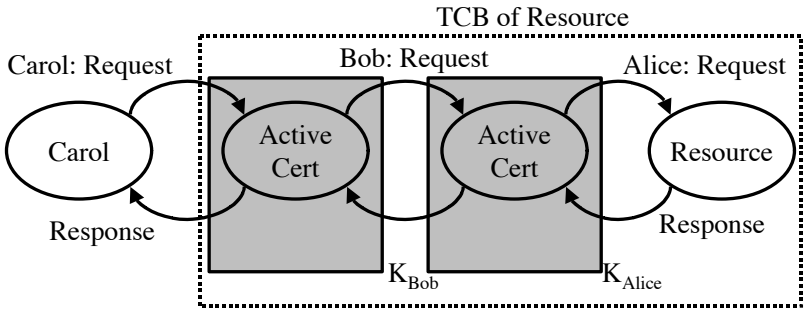

Figure 3. Chained Active Certificates.

3.1 Chained Operation

To begin, consider a simple example, where Bob wishes to delegate rights he acquired via an active certificate further. He creates a new certificate program that enforces his own restrictions, signs it, and hands it to Carol. When Carol wishes to access the Resource, she presents it with both Bob's and Alice's certificates. The Resource instantiates the certificates in a chain, as shown in Figure 3. Carol's request is first passed to Bob's certificate, which then forwards it to Alice's certificate. At this point the request is authenticated as coming from Bob, and will therefore be accepted by Alice's certificate. Responses are passed back up the chain of certificates. Carol can re-delegate *her* rights using the same process; chains of certificates of arbitrary depth are possible.

Notice that although Alice may not wish Bob to be able to redelegate 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.

3.2 Policy Attributes

As witnessed above, chaining can be used to combine several active certificates to implement a composite policy. This is a powerful technique that can be used to break a complex policy into several subcomponents, bringing with it the promise of policy modularity. However, modularity requires another important principle: abstraction.

Abstraction can be implemented by way of *policy attributes*. We create a new type of request, which is a wrapper around another request (of arbitrary type) with an additional attribute field. This field is used to specify a policy abstraction generated by one active certificate and intended to be consumed by another. In this way, active certificates can communicate policy decisions to each other.

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.

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 evolve 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 her friends are. 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.

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 could be rewritten to allow any requests but add a *readOnly* attribute.

Another way to provide support is to create an adapter agent operating outside the application that consumes policy attributes such as *readOnly* and enforces restrictions based on request type. This agent can either be used as a library by Alice's other certificates, or instantiated as a standalone certificate. In the latter case, the certificate would not be performing delegation but instead enforcing a higher-level policy; therefore, it should be signed by Alice and only accept requests from her. Placing the agent outside the application has the advantage that it can evolve independently.

3.4 Hierarchical PKI

A more complex example that uses composition and abstraction is a hierarchical public key infrastructure. A public key infrastructure uses certificates to create associations between names and public keys. We can represent these name associations as policy attributes. For example, in a flat hierarchy, a certificate authority may assign Bob the key K. In this case, the CA would create an active certificate that accepts any request authenticated with key K and forwards an attributed request with the field *name* set to "Bob". Other certificates can rely on such mappings to delegate rights to named principals, instead of public keys; those certificates should accept requests authenticated as coming from the CA and with the *name* field set appropriately. Of course, the CA is then able to issue arbitrary requests claiming to be from an authorized principal, but this is inherent in all hierarchical public key infrastructures: a CA is always free to associate a name with its own key and thus impersonate any principal. Our formulation merely makes this ability more explicit.

It is easy to introduce subauthorities: the root CA creates a certificate which accepts any named request that is authenticated by the key of the subauthority, as long as the name is within the authority's domain of power; this is a straightforward example of rights delegation. Notice that in this case there is a one-to-one correspondence between conventional certificates 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.

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 backwards compatibility. In contrast, conventional certificate systems are difficult to upgrade, since all the libraries that interpret certificates must be replaced, and backwards 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 infeasible to automatically tell what an active certificate does based on its content, thus it is difficult to tell which subset of a collection of certificates will be useful to authenticate Bob to the Resource. 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.

4 Security Analysis

In this section, we formally model the operation of active certificates using a belief logic defined by Abadi et al [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.

When Alice ordinarily accesses the Resource without delegation, she sends a request x over a secure channel. This is represented in the logic as A *says* x. The Resource receives the message and performs an authorization decision to see if Alice is allowed to do action x, and if the authorization is successful, proceeds with the request.

When Bob wishes to access the resource, he first sends Alice's active certificate to the Resource. The certificate is signed by Alice's public key, which we can model by K_A *says* c. The contents of the certificates represent Alice's policy delegating access to Bob, so it may be tempting to say $c \equiv B \Rightarrow A$, where $\Rightarrow$ is the "speaks for" operator, defined as

$$(A \Rightarrow B) \supset ((A \textit{ says } s) \supset (B \textit{ says } s)^1 \qquad (1)$$

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

After sending the certificate, Bob sends his request x to the Resource. The request is given as input to the certificate program, which we will call F. To model the program within the logic, we consider it as a function on statements in the logic. Since the program knows that it was Bob who made the request, we give it B *says* x as input. The program then produces another request $x' = F(B \textit{ says } x)$, which is then passed to the Resource. At this point, x' must be interpreted as if it were coming from Alice for delegation to succeed. Since the program is acting *on Alice's behalf*, it is appropriate to use the $\Rightarrow$ operator. We need to define a new principal P_F representing the program, and introduce the following rule:

$$t \supset (P_F \textit{ says } F(t)) \qquad (2)$$

In other words, P_F says whatever the program outputs. Now Alice's certificate can be defined simply as:

$$K_A \textit{ says } (P_F \Rightarrow A) \qquad (3)$$

[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.

So, upon getting the request x from Bob, the Resource passes it to the active certificate to obtain $F(B \textit{ says } x) = x'$. It then applies (2) to obtain P_F *says* x'. Now it needs to interpret the policy in the active certificate. We will assume that it knows that K_A is Alice's public key, and therefore $K_A \Rightarrow A$. Then it can derive A *says* $(P_F \Rightarrow A)$. We also need another assumption: A *controls* $(X \Rightarrow A)$, which says that Alice has the authority to delegate her rights — this is not implicit in the logic, but necessary for our system. Combining the two statements, the Resource obtains $P_F \Rightarrow A$, which allows it to apply (1) and finally derive A *says* x'. At this point it can apply the authorization decision as if Alice made the request herself, and the delegation succeeds.

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:

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

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

4.1 Authentication of Responses

We can also use the logic to reason about the validity of responses received from the Resource when active certificates are used. When Alice accesses the Resource directly, a response y to a request x is interpreted as R *says* $result_R(x, y)$. The *result* predicate is used to associate a response with the appropriate request. However, what can Bob tell about the response when active certificates are in place? Unfortunately, as a consequence of interposition, Bob cannot assume that R *says* $result_R(x, y)$, because Alice's active certificate is allowed to modify both requests and responses arbitrarily. In a sense, Alice is defining her own predicate $result_A$ by her certificate, which may be different from the Resource's predicate. However, it would be incorrect for Bob to assume that A *says* $result_A(x, y)$, since the execution of Alice's certificate at the Resource is not monitored by either Alice or Bob; the Resource is free to return arbitrary results. We must therefore take a closer look at the operation of active certificates.

Recall that the active certificate passes a request x' to the Resource, where $x' = F(B \textit{ says } x)$. Therefore, when Alice's certificate receives a response y' from the Resource, it can infer that: $result_R(x', y')$. The certificate then uses y' to derive the final response y. It is therefore appropriate

to model the certificate's modifications to the response as:

$$F(result_R(x', y')) = result_A(x, y)$$

In other words, given the response from the Resource to the request x', Alice's certificate computes its own response to the request x. The resource can apply (2), substituting $result_R(x', y')$ for t, and (3), to obtain A *says* $result_A(x, y)$. This can be sent back to Bob, who can obtain the final result:

$$R \text{ says } A \text{ says } result_A(x, y)$$

This must be the interpretation that Bob gives to the result, as it reflects the nature in which the result is derived. Unfortunately, this statement is weaker than one Bob would expect in the non-delegated case; however, it 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.

5 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.

5.1 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.

5.2 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 `handleTask` 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.

5.3 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.

5.4 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.

5.5 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 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 4. 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.

5.6 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. 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.

```
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 4. A Name Certificate.

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.

5.7 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 5.5.) 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.

6 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.

7 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.

In a complicated system with a large pool of available certificates, it is important to have automated search mechanisms to find a sequence of 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.

8 Conclusions

In this paper 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.

9 Acknowledgments

We would like to thank Adrian Perrig, Mark Miller, David Wagner, Dawn Song, Oleg Kolesnikov, and the anonymous referees for insightful comments on earlier versions of this paper.

References

[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.