Distributed SPKI/SDSI-Based Security for Networks of Devices

Matthew Burnside, Dwaine Clarke, Srinivas Devadas, and Ronald Rivest
MIT Laboratory for Computer Science
{event, declarke, devadas, rivest}@mit.edu

Abstract

We describe a distributed security system based on SPKI/SDSI (Simple Public-Key Infrastructure/Simple Distributed Security Infrastructure) for heterogeneously networked, diverse devices. All components of the system, for example, appliances, wearable gadgets, software agents, and users have associated trusted software proxies that either run on the appliance hardware or on a trusted computer. We describe how security is integrated using two separate protocols: a protocol for secure device-to-proxy communication, and a protocol for secure proxy-to-proxy communication. The proxy architecture and the use of two separate protocols allows us to run a computationally-inexpensive protocol on impoverished devices, and a sophisticated protocol for resource authentication and communication on more powerful devices.

The proxy-to-proxy protocol layers SPKI/SDSI access control over an application protocol, which, in turn, is layered over a key-exchange protocol. The SPKI/SDSI framework provides mechanisms for easy maintenance of access control lists (ACLs). It also features an elegant model for forming groups and delegating authority.

We have constructed a prototype system which allows for secure and efficient access to a variety of networked, mobile devices. We present qualitative and quantitative evaluations of this system.

1 Introduction

Pervasive computing trends [6, 2] are causing a rapid increase in the number of sensory and computing devices in our environment. These devices range from lightweight hardware appliances to supercomputers, and from web applets to databases. Groups of devices may be networked using low-bandwidth wireless radio frequency (RF), Ethernet, infrared, or other means. The communication protocols used within each group may be different. Further, mobility dicates that devices may enter and leave a network at a high rate. Diversity in devices, as well as the heterogeneous and dynamic nature of these networks, present significant security challenges.

Implementing existing forms of secure, private communication using a typical public-key infrastructure on all devices is difficult because the necessary cryptographic algorithms are CPU-intensive. A common public-key algorithm such as RSA using 1024-bit keys takes 43ms to sign and 0.6ms to verify on a 200MHz Intel Pentium Pro (a 32-bit processor) [29]. Some devices may have 8-bit microcontrollers running at 1-4 MHz, so public-key cryptography on a device itself may not be an option. However, public-key based communication between devices over a network is still desirable.

In the rest of this section, we give an overview of our approach and introduce our prototype system. We describe the architecture of the resource discovery and communication system in Section 2. The device-to-proxy security protocol is briefly summarized in Section 3. We introduce SPKI/SDSI and present the proxy-to-proxy protocol that uses SPKI/SDSI in Section 4. Section 5 describes the important features of SPKI/SDSI. Related work is discussed in Section 6. We evaluate the system in Section 7. Section 8 concludes the paper.

1.1 Our Approach

To allow our system to use a public-key security model while still allowing for simple devices, we create a software proxy for each device. These proxies are trusted implicitly and run on either an embedded processor on the device or on a trusted computer. In the case where the proxy is running on an embedded processor, we assume that device-to-proxy communication is inherently secure.[1] If the device has minimal computational power,[2] we force the communication to adhere to a device-to-proxy se-

[1]For example, in a video camera, the software that controls various actuators runs on a powerful processor, and the proxy for the camera can also run on the embedded processor.

[2]This is typically the case for lightweight devices, e.g. remote controls, bio-sensors, etc.

curity protocol (cf. Section 3). Proxies communicate with each other using a separate protocol based on SPKI/SDSI. Having two different protocols allows us to run a computationally inexpensive security protocol on impoverished devices and a sophisticated protocol for resource authentication and communication on more powerful devices.

Our main focus in this paper is the proxy-to-proxy protocol, which uses SPKI/SDSI for access control. The proxy-to-proxy protocol layers SPKI/SDSI access control over an application protocol, which in turn is layered over a key-exchange protocol. This allows us to deal with a variety of application protocols which may be implemented across a wired or wireless link in a heterogeneous network.

Using the SPKI/SDSI framework, access control lists (ACLs) associated with resources can be created once and rarely need to be modified. User access rights are modified by issuing certificates based on group membership. SPKI/SDSI also facilitates short certificate validity periods to assist in the problem of certificate revocation. In addition, SPKI/SDSI features an elegant model for delegation of authority, allowing for the partitioning of responsibilities. The principal maintaining the ACL could, but need not be, the same principal that authorizes users to use a resource. This significantly eases the burden of system administration.

1.2 Prototype System

Using the ideas described above, we have constructed a prototype system which allows for secure and efficient access to networked mobile devices. Devices communicate with their proxies securely using various protocols. In particular, lightweight devices use the protocol described in Section 3 and Appendix A. Proxies communicate using the protocol described in Section 4.

By exploiting SPKI/SDSI, security is not compromised as new users and devices enter the system, nor when users and devices leave the system. As presented in this paper, we believe that the use of SPKI/SDSI has resulted in a system that is secure, scalable, efficient, and easy to maintain.

2 System Architecture

The system has three primary component types: devices, proxies and servers. A *device* is our name for any type of shared network resource, either hardware or software. It could be a printer, a wireless security camera, a lamp, or a software agent. Since communication protocols and bandwidth between devices can vary widely, each device has a unique *proxy* to unify

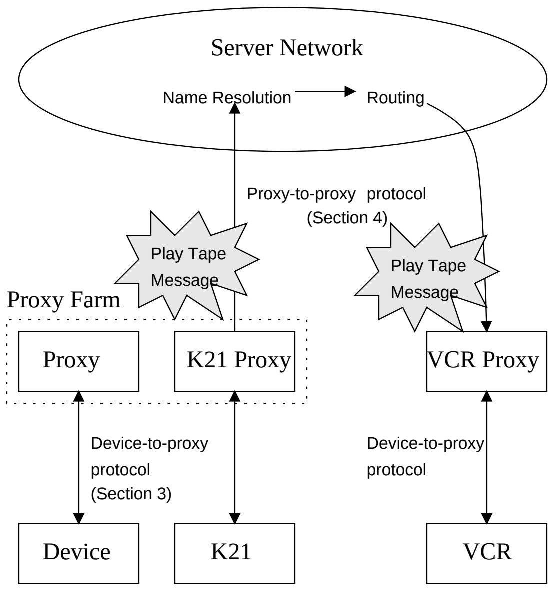

Figure 1: System Overview

its interface with other devices. The *servers* provide naming and discovery facilities to the various devices.

We assume a one-to-one correspondence between devices and proxies. We also assume that all users are equipped with K21s, whose proxies run on trusted computers. Thus our system only needs to deal with devices, proxies and the server network.

The system we describe is illustrated in Figure 1.

2.1 Devices

Each device, hardware or software, has an associated trusted software proxy. In the case of a hardware device, the proxy may run on an embedded processor within the device, or on a trusted computer networked with the device. In the case of a software device, the device can incorporate the proxy software itself.

Each user possesses a badge called the K21 which identifies the user to the system. In addition to information regarding the user's identity, the K21 includes functionality that makes it location-aware. The K21 has a lightweight processor, thus it is a simple device, incapable of executing complex cryptographical operations. It typically has a single button for input.[3]

Each device communicates with its own proxy over the appropriate protocol for that particular device. A printer wired into an Ethernet can communicate with its proxy using TCP/IP. A wireless security camera uses a wireless protocol for the same purpose. The

[3]See [15] for more information on the K21.

K21 communicates to its proxy using the device-to-proxy protocol described in Section 3 and Appendix A. Thus, the device-side portion of the proxy must be customized for each particular device.

2.2 Proxy

The proxy is software that runs on a network-visible computer. The proxy's primary function is to make access-control decisions on behalf of the device it represents. It may also perform secondary functions such as running scripted actions on behalf of the device and interfacing with a directory service.

The proxy provides a very simple API to the device. The *sendToProxy*() method is called by the device to send messages to the proxy. The *sendToDevice*() method is a called by the proxy to send messages to the device. When a proxy receives a message from another proxy, depending on the message, the proxy may translate it into a form that can be understood by the proxy's particular device. It then forwards the message to the device. When a proxy receives a message from its device, it may translate the message into a general form understood by all proxies, and then forward the message to other proxies. Any time a proxy receives a message, before performing a translation and passing the message on to the device, it performs the access control checks described in Sections 4 and 5.

For ease of administration, we group proxies by their administrators. An administrator's set of proxies is called a *proxy farm*. This set specifically includes the proxy for the administrator's K21, which is considered the root proxy of the proxy farm. When the administrator adds a new device to the system, the device's proxy is automatically given a default ACL, a duplicate of the ACL for the administrator's K21 proxy. The administrator can manually change the ACL later, if he desires.

2.3 Servers and the Server Network

This network consists of a distributed collection of independent name servers and routers. In fact, each server acts as both a name server *and* a router. This is similar to the name resolvers in the Intentional Naming System (INS) [1], which resolve device names to IP addresses, but can also route events. If the destination name for an event matches multiple proxies, the server network will route the event to all matching destinations.

When a proxy comes online, it registers itself with one of these servers. When a proxy uses a server to perform a lookup on a name, the server searches its directory for all names that match the given name, and returns their IP addresses.

2.4 Resource discovery

The mechanism for resource discovery is similar to the resource discovery protocol used by Sun Microsystem's Jini [25]. When a device comes online, it instructs its proxy to repeatedly broadcast a request for a server to the local subnetwork. The request contains the device's name and the IP address and port of its proxy. When a server receives one of these requests, it issues a lease to the proxy.[4] That is, it adds the name/IP address pair to its directory. The proxy must periodically renew its lease by sending the same name/IP address pair to the server, otherwise the server removes it from the directory. In this fashion, if a device silently goes offline, or the IP address changes, the proxy's lease will no longer get renewed and the server will quickly notice and either remove it from the directory or change the IP address.

For example, imagine a device with the name [name=foo] which has a proxy running on 10.1.2.3:4011. When the device is turned on, it informs its proxy that it has come online, using the device-to-proxy protocol described in Section 3 and Appendix A. The proxy begins to broadcast lease-request packets of the form $\langle$[name=foo], 10.1.2.3:4011$\rangle$ on the local subnetwork. When (or if) a server receives one of these packets, it checks its directory for [name=foo]. If [name=foo] is not there, the server creates a lease for it by adding the name/IP address pair to the directory. If [name=foo] *is* in the directory, the server renews the lease. Suppose at some later time the device is turned off. When the device goes down, it brings the proxy offline with it, so the lease request packets no longer get broadcast. That device's lease stops getting renewed. After some short, pre-defined period of time, the server expires the unrenewed lease and removes it from the directory.

3 Device-to-Proxy Protocol

The device-to-proxy protocol varies for different types of devices. In particular, we consider lightweight devices with low-bandwidth wireless network connections and slow CPUs, and heavyweight devices with higher-bandwidth connections and faster CPUs. We assume that heavyweight devices are capable of running proxy software locally (i.e., the proxy for a printer could run on the printer's CPU). With a

[4]Handling the scenario where the device is making false claims about its attributes in the lease request packet is the subject of ongoing research.

local proxy, a security protocol for secure device-to-proxy communication is unnecessary, assuming nobody tampers with the hardware itself. For lightweight devices, the proxy must run elsewhere. Appendix A gives a brief overview of a security protocol which is low-bandwidth and not CPU-intensive. Mills' Master's thesis [15] has a more in-depth description.

4 Proxy to Proxy Protocol

SPKI/SDSI (Simple Public Key Infrastructure/Simple Distributed Security Infrastructure) [7, 21] is a security infrastructure that is designed to facilitate the development of scalable, secure, distributed computing systems. SPKI/SDSI provides fine-grained access control using a local name space architecture and a simple, flexible, trust policy model.

SPKI/SDSI is a public key infrastructure with an egalitarian design. The *principals are the public keys* and each public key is a certificate authority. Each principal can issue certificates on the same basis as any other principal. There is no hierarchical global infrastructure. SPKI/SDSI communities are built from the bottom-up, in a distributed manner, and do not require a trusted "root."

4.1 SPKI/SDSI Integration

We have adopted a client-server architecture for the proxies. When a particular principal, acting on behalf of a device or user, makes a request via one proxy to a device represented by another proxy, the first proxy acts like a client, and the second as a server. Resources on the server are either public or protected by SPKI/SDSI ACLs. If the requested resource is protected by an ACL, the principal's request must be accompanied by a "*proof of authenticity*" that shows that it is authentic, and a "*proof of authorization*" that shows the principal is authorized to perform the particular request on the particular resource. The proof of authenticity is typically a signed request, and the proof of authorization is typically a chain of certificates. The principal that signed the request must be the same principal that the chain of certificates authorizes.

This system design, and the protocol between the proxies, is very similar to that used in SPKI/SDSI's Project Geronimo, in which SPKI/SDSI was integrated into Apache and Netscape, and used to provide client access control over the web. Project Geronimo is described in two Master's theses [3, 14].

4.2 Protocol

The protocol implemented by the client and server proxies consists of four messages. This protocol is outlined in Figure 2, and following is its description:

1. The client proxy sends a request, unauthenticated and unauthorized, to the server proxy.

2. If the client requests access to a protected resource, the server responds with the ACL protecting the resource[5] and the *tag* formed from the client's request. A tag is a SPKI/SDSI data structure which represents a set of requests. There are examples of tags in the SPKI/SDSI IETF drafts [7]. If there is no ACL protecting the requested resource, the request is immediately honored.

3. (a) The client proxy generates a chain of certificates using the SPKI/SDSI *certificate chain discovery algorithm* [4, 3]. This certificate chain provides a *proof of authorization* that the user's key is authorized to perform its request.

   The certificate chain discovery algorithm takes as input the ACL and tag from the server, the user's public key (principal), the user's set of certificates, and a timestamp. If it exists, the algorithm returns a chain of user certificates which provides proof that the user's public key is authorized to perform the operation(s) specified in the tag, at the time specified in the timestamp. If the algorithm is unable to generate a chain because the user does not have the necessary certificates,[6] or if the user's key is directly on the ACL, the algorithm returns an empty certificate chain. The client generates the timestamp using its local clock.

   (b) The client creates a SPKI/SDSI sequence [7] consisting of the tag and the timestamp. It signs this sequence with the user's private key, and includes copy of the user's public

[5]The ACL itself could be a protected resource, protected by another ACL. In this case, the server will return the latter ACL. The client will need to demonstrate that the user's key is on this ACL, either directly or via certificates, before gaining access to the ACL protecting the object to which access was originally requested.

[6]If the user does not have the necessary certificates, the client could immediately return an error. In our design, however, we choose not to return an error at this point; instead, we let the client send an empty certificate chain to the server. This way, when the request does not verify, the client can possibly be sent some error information by the server which lets the user know where he should go to get valid certificates.

key in the SPKI/SDSI signature. The client then sends the tag-timestamp sequence, the signature, and the certificate chain generated in step 3a to the server.

4. The server verifies the request by:

   (a) Checking the timestamp in the tag-timestamp sequence against the time on the server's local clock to ensure that the request was made recently.[7]

   (b) Recreating the tag from the client's request and checking that it is the same as the tag in the tag-timestamp sequence.

   (c) Extracting the public key from the signature.

   (d) Verifying the signature on the tag-timestamp sequence using this key.

   (e) Validating the certificates in the certificate chain.

   (f) Verifying that there is a chain of authorization from an entry on the ACL to the key from the signature, via the certificate chain presented. The authorization chain must authorize the client to perform the requested operation.

   If the request verifies, it is honored. If it does not verify, it is denied and the server proxy returns an error to the client proxy. This error is returned whenever the client presents an authenticated request that is denied.

The protocol can be viewed as a typical challenge-response protocol. The server reply in step 2 of the protocol is a challenge the server issues the client, saying, "You are trying to access a protected file. Prove to me that you have the credentials to perform the operation you are requesting on the resource protected by this ACL." The client uses the ACL to help it produce a certificate chain, using the SPKI/SDSI certificate chain discovery algorithm. It then sends the certificate chain and signed request in a second request to the server proxy. The signed request provides proof of authenticity, and the certificate chain provides proof of authorization. The server attempts to verify the second request, and if it succeeds, it honors the request.

The timestamp in the tag-timestamp sequence helps to protect against certain types of replay attacks. For example, suppose the server logs requests

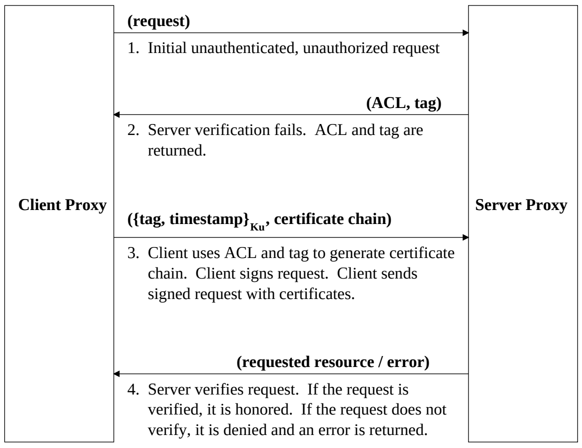
(request)
1. Initial unauthenticated, unauthorized request
(ACL, tag)
2. Server verification fails. ACL and tag are returned.
Client Proxy
({tag, timestamp}$_{Ku}$, certificate chain)
Server Proxy
3. Client uses ACL and tag to generate certificate chain. Client signs request. Client sends signed request with certificates.
(requested resource / error)
4. Server verifies request. If the request is verified, it is honored. If the request does not verify, it is denied and an error is returned.

Figure 2: SPKI/SDSI Proxy to Proxy Access Control Protocol

and suppose that this log is not disposed of properly. If an adversary gains access to the logs, the timestamp prevents him from replaying requests found in the log and gaining access to protected resources.[8]

4.2.1 Additional Security Considerations

The SPKI/SDSI protocol, as described, addresses the issue of providing client access control. The protocol does not ensure confidentiality, authenticate servers, or provide protection against replay attacks from the network.

The Secure Sockets Layer (SSL) protocol is the most widely used security protocol today. The Transport Layer Security (TLS) protocol is the successor to SSL. Principal goals of SSL/TLS [17] include providing confidentiality and data integrity of traffic between the client and server, and providing authentication of the server. There is support for client authentication, but client authentication is optional. The SPKI/SDSI Access Control protocol can be layered over a key-exchange protocol like TLS/SSL to provide additional security. TLS/SSL currently uses the X.509 PKI to authenticate servers, but it could just as well use SPKI/SDSI in a similar manner. In addition to the features already stated, SSL/TLS also provides protection against replay attacks from the network, and protection against person-in-the-middle attacks. With these considerations, the layering of the protocols is shown in Figure 3. In the figure, 'Application

[7]In our prototype implementation, the server checks that the timestamp in the client's tag-timestamp sequence is within five minutes of the server's local time.

[8]In order to use timestamps, the client's clock and server's clock need to be fairly synchronized; SPKI/SDSI already makes an assumption about fairly synchronized clocks when validity time periods are specified in certificates. An alternative approach to using timestamps is to use nonces in the protocol.

SPKI/SDSI Access Control Protocol
Application Protocol
Key-Exchange Protocol with Server Authentication
TCP/IP

Figure 3: Example Layering of Protocols

Protocol' refers to the standard communication protocol between the client and server proxies, without security.

SSL/TLS authenticates the server proxy. However, it does not indicate whether the server proxy is authorized to accept the client's request. For example, it may be the case that the client proxy is requesting to print a 'top secret' document, say, and only certain printers should be used to print 'top secret' documents. With SSL/TLS and the SPKI/SDSI Client Access Control Protocol we have described so far, the client proxy will know that the public key of the proxy with which it is communicating is bound to a particular address, and the server proxy will know that the client proxy is authorized to print to it. However, the client proxy still will not know if the server proxy is authorized to print 'top secret' documents. If it sends the 'top secret' document to be printed, the server proxy will accept the document and print it, even though the document should not have been sent to it in the first place.

To approach this problem, we propose extending the SPKI/SDSI protocol so that the client requests authorization from the server and the server proves to the client that it is authorized to handle the client's request (before the client sends the document off to be printed). To extend the protocol, the SPKI/SDSI protocol described in Section 4.2 is run from the client proxy to the server proxy, and then run in the *reverse* direction, from the server proxy to the client proxy. Thus, the client proxy will present a SPKI/SDSI certificate chain proving that it is authorized to perform its request, and the server proxy will present a SPKI/SDSI certificate chain proving that it is authorized to accept and perform the client's request. Again, if additional security is needed, the extended protocol can be layered over SSL/TLS.

Note that the SPKI/SDSI Access Control Protocol is an example of the *end-to-end argument* [22]. The access control decisions are made in the uppermost layer, involving only the client and the server.

5 Access Control and Authorization in SPKI/SDSI

5.1 Naming Architecture

There are two types of certificates in SPKI/SDSI: name certificates and authorization certificates. Name certificates are described in this section, and authorization certificates are described in Section 5.4. A SPKI/SDSI name certificate defines a local name in the certificate issuer's local name space. A name certificate consists of four fields:

issuer The public key that signs the certificate.

identifier An identifier is a single word over some standard alphabet, such as `Alice`, `Bob`, `Friends`, `A`, `B`. In this document, identifiers will be specified in typewriter font.

The identifier determines the local name that is being defined. The name consists of the issuer's key and this identifier. A name that consists of a single key followed by a single word is known as a "*local name*" [4]. Because a name certificate can only define a local name, each principal can only define names within its own name space.

subject A subject can be a public key or a name consisting of a single public key followed by one or more identifiers. The public key in the subject does not have to be the issuer's key.

In a name certificate, the subject is the new meaning of the local name being defined.

validity specification Normally, the validity specification is a time period during which a certificate is valid, assuming the signature verifies. Beyond this period, the certificate has expired and should be renewed. The validity specification usually takes the form (t_1, t_2), specifying that the certificate is valid from time t_1 to time t_2, inclusive. The validity specification can also take the form of an *online check* that is performed to determine if the certificate is valid. An online check specifies the address of a server to be queried before the certificate can be considered valid. The server would contain a list of revoked certificates, and let the certificate verifier know if the certificate has been revoked.

SPKI/SDSI name certificates bind local names to public keys. Assuming each user has a single public-private key pair, the name-to-key binding is a multi-valued function: each name is bound to one or more keys. A single name certificate can define a name in the issuer's local name space to be a public key, another name in his/her local name space, or a name in another principal's local name space. A name certificate that defines the local name K `A` to be the subject S can be denoted K `A` $\longrightarrow$ S,[9] where K is the issuer's key.

Since each principal can issue name certificates, each principal has its own local name space consisting of the names it defines. SPKI/SDSI, thus, has a local name space architecture which helps to make the infrastructure scalable. A user does not have to ensure that the names he defines are unique in a global name space; he can define names which are meaningful to him, which he can easily remember and recognize.

A SPKI/SDSI *group* is typically a set of principals. Each group has a name and a set of members. The name is local to some principal, the "owner" of the group, and he is the only one who can change the group definition. A group definition may explicitly reference the members of the group, or reference other groups (which may even belong to someone else). To define a group, a group owner issues to each group member a name certificate defining the local name of the group to be that member's key or name. A group owner can also add another principal's group to his group by issuing name certificates binding the name of his group to the name of the other group.

Figure 4 gives an example of a SPKI/SDSI group. In the example, Alice's friends include Bob (K_B) and Carol (K_C); she can add them to her group `friends` by issuing certificate 1 to Bob and certificate 2 to Carol. (She could have achieved the same effect by issuing the certificates K_A `friends` $\longrightarrow$ K_A `Bob` and K_A `Bob` $\longrightarrow$ K_B to Bob, and K_A `friends` $\longrightarrow$ K_A `Carol` and K_A `Carol` $\longrightarrow$ K_C to Carol.) Edward (K_E) has named his key K_E `Edward` by issuing the certificate K_E `Edward` $\longrightarrow$ K_E. Alice adds Edward to her group `friends` by issuing certificate 3 to him. Alice has a sister, Fiona (K_F), and she considers all of Fiona's friends to be her friends. She adds them to her group by issuing certificate 4 to Fiona's friends. Also, Bob's sister's friends are Alice's friends, and she issues certificate 5 to them (note that the name "K_A `B C D`" means K_A's `B`'s `C`'s `D`). In summary, with the certificates in Figure 4, K_A `friends` in Alice's local name space is bound directly to the keys K_B, K_C, and indirectly to the keys referenced by K_E `Edward`, K_F `friends` and K_B `sister friends`.

Note that it is easy for a person to belong to multiple groups. For example, suppose that, besides being

[9]The validity specification will, generally, not be crucial to discussions here since any certificate which fails its validity specification at the time it is being used should be ignored.

$$K_A\ \texttt{friends} \longrightarrow K_B \quad (1)$$
$$K_A\ \texttt{friends} \longrightarrow K_C \quad (2)$$
$$K_A\ \texttt{friends} \longrightarrow K_E\ \texttt{Edward} \quad (3)$$
$$K_A\ \texttt{friends} \longrightarrow K_F\ \texttt{friends} \quad (4)$$
$$K_A\ \texttt{friends} \longrightarrow K_B\ \texttt{sister friends} \quad (5)$$

Figure 4: An example of a SPKI/SDSI group: K_A `friends` (Alice's friends)

Alice's friend, Bob is also a friend of George (K_G) and Harold (K_H). George simply issues Bob the certificate K_G `friends` $\longrightarrow$ K_B, and Harold issues him the certificate K_H `friends` $\longrightarrow$ K_B. With these certificates, and the certificate K_A `friends` $\longrightarrow$ K_B issued by Alice, Bob is a member of the groups K_A `friends`, K_G `friends`, and K_H `friends`.

The ability to define groups is one of the principal notions of SPKI/SDSI. This feature facilitates easy management of ACLs, as will be described in Section 5.2. It also makes it easier and more intuitive to define security policies. Because the names of the groups are at the discretion of the owners, the groups' names can be meaningful and intuitive. Security policies can be defined in terms of these groups, simplifying the auditing of group definitions and ACLs.

5.2 Access Control Lists

SPKI/SDSI is primarily concerned with authorizing principals to perform particular operations on protected resources. An administrator controls access to a resource by setting up an ACL to protect it. A principal (public key) makes a request to perform a particular operation on the resource. Examples of requests are a request to read a file, a request to login to an account, or a request to turn on an appliance. In these examples, the protected resources are the file, account and appliance, respectively.

A SPKI/SDSI ACL consists of a list of entries. Each entry specifies an operation or set of operations that the subject is permitted to perform on the resource the ACL protects. Each entry in the ACL has three fields:

subject The key or group (a key followed by one or more identifiers) that is being permitted to perform the operations in the tag.

tag This is the same as the tag described in Section 4.2.

delegation bit If this bit is true, the entry's subject is allowed to authorize other principals to access the protected resource. If this bit is false, the subject is not allowed to do so. In this case, each principal requesting access to the resource must either be the public key specified in the subject, or be a member of the group specified in the subject, before the request can be honored.

5.3 ACLs and Groups

If an administrator wishes to grant a set of principals access to a number of resources, each protected by a separate ACL, he can simply define a group and place the group name on each ACL. The ACLs need only be updated once. From then on, as new members join the group, they are issued the relevant certificates and are automatically authorized to access the protected resources without the administrator having to update the ACLs again. It is clear that using groups makes it easier and more efficient to maintain and update ACLs, since an explicit list of all the principals does not have to be stored on each ACL.

In the example in Figure 4, if the group name K_A `friends` is specified in an entry on an ACL, then K_B, K_C, and all the keys referenced by K_E `Edward`, K_F `friends` and K_B `sister friends` will automatically be authorized to perform the operation(s) specified in that entry's tag. Furthermore, and perhaps more importantly, there can be a *delayed definition* of the group. An administrator can add a group to one or more ACLs without knowing the group's members beforehand. He can update the ACLs with an entry for the group, and at some later time that is convenient and appropriate, he can issue name certificates adding principals to the group. He does not have to know all the members of a group when he is setting up his ACLs.

Note that an administrator is free to add any principal's group to his ACL. He is not restricted to just adding his own groups (though he could just add his own groups if he so desires). For example, the user controlling K_A could add groups K_F `friends` and K_G `friends` to an ACL he maintains.

5.4 Authorization Certificates

A SPKI/SDSI authorization certificate grants a specific authorization from the certificate's issuer to the certificate's subject. To keep the infrastructure simple, a single certificate cannot both define a name and grant an authorization: i.e., each certificate is either strictly a name certificate or an authorization certificate. A SPKI/SDSI authorization certificate consists of five fields:

issuer The key that signs the certificate. The issuer is the principal granting the specific authorization.

subject The key or group (a key followed by one or more identifiers) that is receiving the grant of authorization.

tag This is the same as the tag described in Section 4.2.

delegation bit This is the same as the delegation bit described in Section 5.2.

validity specification This is the same as the validity specification described in Section 5.2..

An authorization certificate in which the issuer K grants the authorization specified in tag T to subject S with the delegation bit set to true can be denoted as $K\boxed{1} \xrightarrow{\mathrm{T}} \mathrm{S}\boxed{1}$. If the delegation bit is set to false, the certificate is denoted as $K\boxed{1} \xrightarrow{\mathrm{T}} \mathrm{S}\boxed{0}$. SPKI/SDSI ACLs have similar syntax to SPKI/SDSI authorization certificates. In fact, each entry of an ACL can be considered to be an authorization certificate with the issuer being the owner of the ACL, and the subject, tag and delegation bit being as specified in the entry. (It is assumed that the owner of the ACL removes ACL entries when they are no longer valid, and thus validity specifications in ACL entries are optional.) Each entry on an ACL has the representation $\mathit{Self}\boxed{1} \xrightarrow{\mathrm{T}} \mathrm{S}\boxed{1}$ if the delegation bit is true, and $\mathit{Self}\boxed{1} \xrightarrow{\mathrm{T}} \mathrm{S}\boxed{0}$ if the delegation bit is false. The special designator "*Self*" represents the owner of the ACL.

Thus, for example, if K_A is Alice's public key, and K_B is Bob's public key, Alice can issue an authorization certificate, $K_A\boxed{1} \xrightarrow{T_1} K_B\boxed{1}$ granting Bob the authorization specified in T_1 with the permission to delegate this authorization. As another example, $\mathit{Self}\boxed{1} \xrightarrow{T_2} K_B$ `friends`$\boxed{0}$ represents an ACL entry with the group K_B `friends` on it. The members of this group are allowed to perform the operations specified in T_2, but are not allowed to grant this authority to anyone else.

5.5 Delegation

An important feature of SPKI/SDSI is the ability to delegate authorizations. One way of thinking of an authorization certificate is that it *transfers* or *propagates* a specific authorization from the issuer to the subject. If the delegation bit is set in the certificate, the subject is allowed to continue propagating this authorization, or some subset of it, to other principals, by issuing authorization certificates to them. If the subject, in turn, sets the delegation bit on its certificates to true, the principals to whom it is propagating the authorization will also be able to issue authorization certificates granting new principals the authorization, and so on.

When security decisions are made, it is an individual's characteristics that are used to determine whether he should be issued access credentials for a protected resource. However, sometimes the entity responsible for protecting a resource may prefer to delegate the responsibility for making this determination. For example, in a research laboratory, it may be the sys-admin who is responsible for setting up and maintaining ACLs on the laboratory's color printers. Instead of having every new graduate student come to him for access credentials, he may want to delegate this responsibility to one or more floor managers. With SPKI/SDSI, the sys-admin can delegate to the floor managers the authority to determine who is allowed to access the color printers. The sys-admin must trust the floor managers to correctly identify new graduate students before issuing them certificates.

Figure 5 gives an example. In the example, the sys-admin ($K_{Sys-Admin}$) adds the group $K_{Sys-Admin}$ `Floor_Managers` to various ACLs and sets the delegation bit to true. $T_{Color\_Printers}$ represents the authority to access color printers. The ACL entry added by the sys-admin is specified in 'certificate' 6. He then issues a name certificate adding a particular floor manager's key, K_{Flr1_Mngr}, to the group $K_{Sys-Admin}$ `Floor_Managers` (certificate 7). This floor manager (for floor 1) has, thus, been delegated the authority to decide who should access the color printers, since the delegation bit in the ACL entry is true. (Of course, the sys-admin can still authorize people to access the color printers as well, but he may decide to leave this job exclusively to floor managers.) Clearly, the sys-admin can do this for other floor managers as well.

The first floor manager can issue authorization certificates to graduate students on his floor. He may also set the delegation bit in their certificates to true, if he wishes to allow the graduate students to authorize others to access the printers. In this case, these graduate students authorize new students after authenticating and validating them. Suppose a particular senior graduate student's key is K_{Senior_Grad}. The floor manager can authorize him by issuing him certificate 8. The senior graduate student can authorize a junior graduate student (K_{Junior_Grad}) by issuing him certificate 9. The junior graduate student will not be able to authorize other students to print to the color printers since the delegation bit in his certificate is false.

$$\mathit{Self}\ \boxed{1} \xrightarrow{T_{Color\_Printer}} K_{Sys-Admin}\texttt{Floor\_Managers}\boxed{1} \tag{6}$$

$$K_{Sys-Admin}\texttt{Floor\_Managers} \longrightarrow K_{Flr1\_Mngr} \tag{7}$$

$$K_{Flr1\_Mngr}\boxed{1} \xrightarrow{T_{Color\_Printer}} K_{Senior\_Grad}\boxed{1} \tag{8}$$

$$K_{Senior\_Grad}\boxed{1} \xrightarrow{T_{Color\_Printer}} K_{Junior\_Grad}\boxed{0} \tag{9}$$

Figure 5: An example of delegation using SPKI/SDSI certificates

When a user is issued his certificates, he should be given all of the certificates necessary to establish the chain of authorization from the particular ACL entry to his public key. In the example described in Figure 5, the floor manager should be given certificate 7; the senior graduate student certificates 7 and 8; and the junior graduate student certificates 7, 8 and 9. In each case, the user will need to use the certificates to establish the authorization chain from the $K_{Sys-Admin}$ `Floor_Managers` group to his key.

5.6 Certificate Guarantee

The SPKI/SDSI certificate guarantee is: "This certificate is good until the expiration date. Period." [19]. SPKI/SDSI advocates using reasonably short validity periods inside certificates. SPKI/SDSI also advocates using *certificates of health* [19] to deal with the specific issue of the requestor's key being compromised.

As with any PKI, issuers must take care when issuing certificates. If a certificate has been incorrectly issued, there should only be a reasonably short period of time in which it can potentially be used. SPKI/SDSI facilitates the use of short validity periods since, in any particular certificate chain, the function of the chain can be easily partitioned among the certificates. Imagine that Alice wants to attend a particular one of Charlie's online team meetings and the meeting is protected by an ACL with only one entry: K_C `Team`. Charlie issues Alice a name certificate that binds her name in his local name space to her key (K_C `Alice` $\longrightarrow K_A$), and another name certificate that adds K_C `Alice` to Charlie's team (K_C `Team` $\longrightarrow K_C$ `Alice`). Charlie allows Alice to attend only a particular meeting by issuing the latter certificate with a validity period set for exactly the time of that meeting.

Conventional PKIs handle key compromise with the same mechanism that they handle certificate revocation. They use Certificate Revocation Lists

(CRLs). A CRL is issued by a certificate authority, and is a blacklist: any certificate on a CRL is invalid.

SPKI/SDSI treats key compromise as a separate issue from certificate revocation. It argues that "certificates should not be revoked merely because the key is compromised. Rather, the signer should present separate evidence to the acceptor that the key has *not* been compromised. Since, in this framework, the no-compromise evidence is separate, the ordinary certificates can continue to be 'valid' even though the key has been compromised." [19] SPKI/SDSI suggests using a new kind of agent, called a *key compromise agent* (KCA), or a *suicide bureau* (SB). Multiple SBs cooperate to serve SPKI/SDSI communities. When Alice creates her key pair, she also signs a personal *suicide note* which she protects in a private place, and also registers her public key with an SB. In the unlikely event that her key is compromised or lost, she sends her suicide note to the SB. The SB broadcasts this note on the SB network so that other SBs are made aware of the compromised key.

If Alice's key has not been compromised or lost, she can ask an SB for a certificate of health, certifying that she believes that she is the only entity controlling her private key. When Alice makes a request signed with her key and accompanied with a chain of certificates, the server proxy does not need to use a CRL to see if her key has been compromised. It can simply require that Alice present a certificate of health along with her request. This certificate will include the time and date that it was issued, and the server proxy can demand a more recent health certificate before honoring the request.

Note that in comparing certificates of health with CRLs, a certificate of health is essentially a positive statement, whereas a CRL is a negative statement. A certificate of health states that a given key has not been compromised; a CRL states that all keys for which the CRL issuer has issued certificates, except the ones on this list, have not been compromised. Negative statements are much harder to prove correct than positive statements.

6 Related Work

Jini [25] network technology from Sun Microsystems centers around the idea of federation building. Jini avoids the use of proxies by assuming that all devices and services in the system will run the Java Virtual Machine. The SIESTA Project [8] at the Helsinki University of Technology has succeeded in building a framework for integrating Jini and SPKI/SDSI. Their implementation has some latency concerns, however, when new authorizations are granted. UC Berkeley's Ninja project [26] uses the Service Discovery Service [5] to securely perform resource discovery in a wide-area network. Other related projects include Hewlett-Packard's CoolTown [9], IBM's TSpaces [11] and University of Washington's Portolano [28].

$$K_{LCS}\ \texttt{LCS} \longrightarrow K_{Theory}\ \texttt{Theory} \tag{10}$$
$$K_{LCS}\ \texttt{LCS} \longrightarrow K_{AI}\ \texttt{AI} \tag{11}$$
$$K_{Theory}\texttt{Theory} \longrightarrow K_{Allison} \tag{12}$$
$$Self\ \boxed{1} \xrightarrow{T_{print\_to\_Beta}} K_{AI\_Sys-Admin}\boxed{1} \tag{13}$$
$$K_{AI\_Sys-Admin}\boxed{1} \xrightarrow{T_{print\_to\_Beta}} K_{AI}\ \texttt{AI}\boxed{0} \tag{14}$$
$$K_{AI}\ \texttt{AI} \longrightarrow K_{Allison} \tag{15}$$

Figure 6: Example scenario.

6.1 Other projects using SPKI/SDSI

Other projects using SPKI/SDSI include Hewlett-Packard's e-Speak product [10], Intel's CDSA release [12], and Berkeley's OceanStore project [27]. HP's eSpeak uses SPKI/SDSI certificates for specifying and delegating authorizations. Intel's CDSA release, which is open-source, includes a SPKI/SDSI service provider for creating certificates, and a module (AuthCompute) for performing authorization computations. OceanStore uses SPKI/SDSI names in their naming architecture.

7 Evaluation

The scenario described in this section is illustrated in Figure 6. Imagine that the Laboratory for Computer Science (LCS) (K_{LCS}) includes two groups: Theory (K_{Theory} `Theory`) and Artificial Intelligence (K_{AI} `AI`). Allison ($K_{Allison}$) is a student in the Theory group. The AI group has a printer, Beta, to which only members of the AI group are allowed to print.

Certificates indicating that the AI and Theory groups are both groups within LCS are published on a server accessible from any LCS workstation (certificates 10 and 11). Allison possesses a certificate, with a reasonably short validity period, that proves that she is a member of the Theory group (certificate 12). Beta has an ACL entry allowing members of the AI group to print to it ('certificate' 13).

Allison now decides to transfer from the Theory group to the AI group. Certificate 12 stops being renewed and soon expires. When Allison first moves to the AI group and attempts to print a document on Beta (using the protocol from Figure 2), her request is denied. Since Allison does not have permission to print to Beta, her print request fails at step 3 of

Protocol step	Timing analysis	Approx CPU time
Cert chain discovery	The worst case is $O(n^3l)$, where n = number of certs, and l = length of longest subject. However, the expected time is $O(nl)$.	330ms, with $n = 2$ and $l = 2$.
Chain validation	The worst case is $O(n)$, where n = number of certs.	200ms, with $n = 2$.

Table 1: Proxy-to-Proxy Protocol analysis.

Figure 2. Beta responds to Allison with an error indicating that she should contact a AI professor (who controls the key, K_{AI}). The AI professor generates a new certificate, certificate 15, and gives Allison certificates 14 and 15 (certificate 14 was issued by the AI sys-admin to the AI professor). Allison re-issues her print command, and this time, the print job succeeds.

7.1 SPKI/SDSI Evaluation

The system described in this paper, integrating SPKI/SDSI with a resource discovery and communication system, is advantageous for a number of reasons:

- ACLs can be created once, and then rarely require modification.
- Few new certificates need to be generated to grant access privileges to a user. In the example from Figure 6, only one new certificate had to be generated, and no ACLs were changed.
- Short certificate validity periods are used for certificate revocation to eliminate the need for inefficient mechanisms such as certificate revocation lists.
- The elegant delegation system allows for efficient resource administration. In the example from Figure 6, the AI sys-admin maintains the ACL, but the AI professor determines the members of the AI group.

The protocol described in Section 4 is efficient. The first two steps of the protocol are a standard request/response pair; no cryptography is required. The significant steps in the protocol are step 3, in which a certificate chain is formed, and step 4, where the chain is verified. Table 1 shows analyses of these two steps. The paper on Certificate Chain Discovery in SPKI/SDSI [4] should be referred to for a discussion of the timing analyses. The CPU times are approximate times measured on a Sun Microsystems Ultra-1 running SunOS 5.7.

8 Conclusion

We believe that the trends in pervasive computing are increasing the diversity and heterogeneity of networks and their constituent devices. Developing security protocols that can handle diverse, mobile devices networked in various ways represents a major challenge. In this paper, we have taken a first step toward meeting this challenge by observing the need for multiple security protocols, each with different characteristics and computational requirements. While we have described a prototype system with two different protocols, other protocols could be included if deemed necessary.

Our main focus in this paper was the proxy-to-proxy security protocol which is based on SPKI/SDSI. Using this protocol facilitates a high degree of security, while providing flexibility in system administration. Further, device mobility, varying device lifetimes, and other dynamic changes to the network can be handled without compromising system security.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The Design and Implementation of an Intentional Naming System. *Operating Systems Review*, 34(5):186-301, December 1999.

[2] G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. Sussman, and D. Zukowski. Challenges: An Application Model for Pervasive Computing. In *Proc. ACM MOBICOM*, August 2000.

[3] D. Clarke. SPKI/SDSI HTTP Server / Certificate Chain Discovery in SPKI/SDSI. Master's thesis, Massachusetts Institute of Technology, 2001.

[4] D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. Rivest. Certificate Chain Discovery in SPKI/SDSI. *Journal of Computer Security*, 2001. To appear.

[5] S. Czerwinski, B. Zhao, T. Hodes, A. Joseph, and R. Katz. An Architecture for a Secure Service Discovery Service. In *Proc. MOBICOM*, August 1999.

[6] M. Dertouzos. The Future of Computing. *Scientific American*, August 1999.

[7] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. Simple Public Key Certificate. *The Internet Society*, July 1999. See http://world.std.com/∼cme/spki.txt.

[8] P. Eronen and P. Nikander. Decentralized Jini Security. In *Proc. of the Network and Distributed System Security Symposium*, February 2001.

[9] Hewlett-Packard. CoolTown. See http://cooltown.hp.com.

[10] Hewlett-Packard. e-Speak. See http://www.e-speak.hp.com.

[11] IBM. TSpaces: Intelligent Connectionware. See http://www.almaden.ibm.com/cs/TSpaces.

[12] Intel. Intel Common Data Security Architecture. See http://developer.intel.com/ial/security.

[13] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. Internet Request for Comments RFC 2104, February 1997.

[14] A. Maywah. An Implementation of a Secure Web Client Using SPKI/SDSI Certificates. Master's thesis, Massachusetts Institute of Technology, 2000.

[15] T. Mills. An Architecture and Implementation of Secure Device Communication in Oxygen. Master's thesis, Massachusetts Institute of Technology, 2001.

[16] OpenSSL. The OpenSSL Project. http://www.openssl.org.

[17] E. Rescola. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2001.

[18] R. Rivest. The MD5 Message-Digest Algorithm. Internet Request for Comments RFC 1321, April 1992.

[19] R. Rivest. Can We Eliminate Certificate Revocation Lists? In *Proc. of Financial Cryptography '98; Springer Lecture Notes in Computer Science No. 1464 (Rafael Hirscfeld, ed.)*, February 1998. See http://theory.lcs.mit.edu/ rivest/revocation.pdf.

[20] R. Rivest. The RC5 Encryption Algorithm. In *Proc. of the 1994 Leuven Workshop on Fast Software Encryption*, 2001.

[21] R. Rivest and B. Lampson. SDSI - A Simple Distributed Security Infrastructure. See http://theory.lcs.mit.edu/ rivest/sdsi10.ps.

[22] J. H. Saltzer, D. Reed, and D. D. Clark. End-to-End Arguments in System Design. See http://www.mit.edu/∼Saltzer/publications/endtoend/.

[23] F. Stajano. The Resurrecting Duckling – What next? In *Proc. of the 8th International Workshop on Security Protocols*, April 2000.

[24] F. Stajano and R. Anderson. The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks. In *Proc. Security Protocols, 7th International Workshop*, 1999.

[25] Sun Microsystems Inc. Jini Network Techonology. http://www.sun.com/jini.

[26] UC Berkeley. The Ninja Project: Enabling Internet-scale Services from Arbitrarily Small Devices. See http://ninja.cs.berkeley.edu.

[27] UC Berkeley. The OceanStore Project: Providing Global-Scale Persistent Data. See http://oceanstore.cs.berkeley.edu.

[28] University of Washington. Portolano: An Expedition into Invisible Computing. See http://portolano.cs.washington.edu.

[29] M. Weiner. Performance Comparison of Public-key Cryptosystems. *RSA Laboratories' CryptoBytes*, 4(1), 1998.

A Appendix: Device-to-Proxy Protocol for Wireless Devices

A.1 Protocol Design

Wireless communication between a device and its proxy is handled by a gateway that translates packetized RF signals into UDP/IP packets. The gateway routes the UDP/IP packets over the network to the proxy. The gateway also performs the reverse function, converting UDP/IP packets from the proxy into RF packets and transmitting them to the device. Figure 7 is an overview of the design. This figure shows a proxy farm containing three proxies; one for each of three separate devices. The figure also demonstrates the use of multiple gateways.

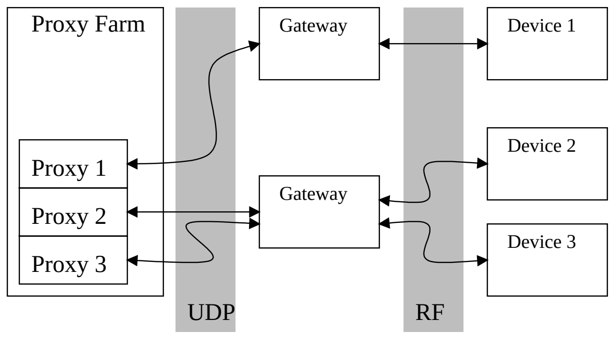

Figure 7: Device-to-Proxy Communication overview

A.2 Security

All communication between the device and the proxy is encrypted and authenticated. RC5 [20] is used for encryption, and the HMAC-MD5 [13, 18] algorithm is used for authentication. Both algorithms use symmetric keys; the proxy and the device share a 128-bit key.

Each packet is encrypted using an OpenSSL [16]-based RC5 implementation. RC5 is a block cipher, which means it is typically used to encrypt 8-byte blocks of data. However, by implementing it using output feedback (OFB) mode, it can be used as a stream cipher. Additionally, by using the OFB mode, only the encryption routine of RC5 is needed, and not the decryption routine. This reduces the software footprint on the device.

HMAC with the MD5 hash function produces a 16-byte message authentication code (MAC). The eight most significant bytes of the MAC are appended to every packet transmitted to and from the device. This reduces the overhead on each packet, but allows an attacker to have to guess fewer bits to forge a MAC. We feel this is a reasonable tradeoff between bandwidth and security, since the range of the RF is less than 100 feet.

A.3 Related work in Device to Proxy Communication

There are many existing technologies that connect devices for automation purposes. Many of these technologies, however, do not focus on the security of the devices at all, or they require the ability to implement complex security algorithms.

The Resurrecting Duckling is a security model for ad-hoc wireless networks [24, 23]. In this model, when devices begin their lives, they must be “imprinted” before they can be used. A master (the mother duck) imprints a device (the duckling) by being the first one to communicate with it. After imprinting, a device only listens to its master. During the process of imprinting, the master is placed in physical contact with the device and they share a secret key that is then used for symmetric-key authentication and encryption. The master can also delegate the control of a device to other devices so that control is not always limited to just the master. A device can be “killed” by its master then resurrected by a new one in order for it to swap masters.

A.4 Evaluation of Device-to-Proxy Protocol

In this section we evaluate the device-to-proxy protocol in terms of its memory and processing requirements.

Component	Code Size (Kb)	Data Size (bytes)
Packet Processing	2.0	191
RF protocol	1.1	153
HMAC-MD5	4.6	386
RC5	3.2	256
Miscellaneous	1.0	0
Total	11.9	986

Table 2: Code and data size. The processor is an Atmel ATMega103L; an 8-bit, 3 volt, CPU running at 4Mhz that uses the Atmel AVR instruction set. It has 128Kb of Flash memory, 2Kb of RAM, and 512 bytes of EEPROM.

Memory Requirements

Table 2 breaks down the memory requirements for various software components. The code size represents memory used in Flash, and data size represents memory used in RAM. The protocol requires approximately 12Kb of code space and 1Kb of data space. The code size we have attained through assembly optimization is small enough that it can be incorporated into virtually any device.

Function	Time (ms)	Clock Cycles
RC5 encrypt/ decrypt (n bytes)	$0.163n + 0.552$	$652n + 2208$
HMAC-MD5 up to 56 bytes	11.48	45,920

Table 3: Encryption and authentication performance.

Processing Requirements

Table 3 breaks down the approximate time it takes for each algorithm to run. The low-end Atmel processor is able to perform encryption and authentication in $\approx 14ms$ because of software optimizations.