Proceedings of the ACM Symposium on Applied Computing, Madrid, Spain, March 2002, to appear.

Proxy-Based Security Protocols in Networked Mobile Devices

Matthew Burnside, Dwaine Clarke, Todd Mills, Andrew Maywah, Srinivas Devadas and Ronald Rivest*

MIT Laboratory for Computer Science
{event, declarke, mills, amaywah, devadas, rivest}@mit.edu

Abstract. We describe a resource discovery and communication system designed for security and privacy. All objects in the system, e.g., 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 and privacy are enforced using two separate protocols: a protocol for secure device-to-proxy communication, and a protocol for secure proxy-to-proxy communication. Using 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.
We detail the device-to-proxy protocol for lightweight wireless devices and the proxy-to-proxy protocol which is based on SPKI/SDSI (Simple Public Key Infrastructure / Simple Distributed Security Infrastructure). A prototype system has been constructed, which allows for secure, yet efficient, access to networked, mobile devices. We present a quantitative evaluation of this system using various metrics.

1 Introduction

Attaining the goals of ubiquitous and pervasive computing [6, 2] is becoming more and more feasible as the number of computing devices

* This work was funded by Acer Inc., Delta Electronics Inc., HP Corp., NTT Inc., Nokia Research Center, and Philips Research under the MIT Project Oxygen partnership, and by DARPA through the Office of Naval Research under contract number N66001-99-2-891702.

in the world increases rapidly. However, there are still significant hurdles to overcome when integrating wearable and embedded devices into a ubiquitous computing environment. These hurdles include designing devices smart enough to collaborate with each other, increasing ease-of-use, and enabling enhanced connectivity between the different devices.

When connectivity is high, the security of the system is a key factor. Devices must only allow access to authorized users and must also keep the communication secure when transmitting or receiving personal or private information.

Implementing typical forms of secure, private communication using a public-key infrastructure on all devices is difficult because the necessary cryptographic algorithms are CPU-intensive. A common public-key cryptographic 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) [30]. Some devices may have 8-bit micro-controllers running at 1-4 MHz, so public-key cryptography on the device itself may not be an option. Nevertheless, public-key based communication between devices over a network is still desirable.

This paper presents our approach to addressing these issues. We describe the architecture of our resource discovery and communication system in Section 2. The device-to-proxy security protocol is described in Section 3. We review SPKI/SDSI and present the proxy-to-proxy protocol that uses SPKI/SDSI in Section 4. Related work is discussed in Section 5. The system is evaluated in Section 6.

1.1 Our approach

To allow the architecture to use a public-key security model on the network while keeping the devices themselves simple, we create a software proxy for each device. All objects in the system, e.g., appliances, wearable gadgets, software agents, and users have associated trusted software proxies that either run on an embedded processor on the appliance, or on a trusted computer. In the case of the proxy running on an embedded processor on the appliance, we assume that device to proxy communication is inherently secure.[1] If the device has minimal computational power,[2] and communicates to its proxy through a wired or

[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, active badges, etc.

wireless network, we force the communication to adhere to a device-to-proxy protocol (cf. Section 3). Proxies communicate with each other using a secure proxy-to-proxy protocol based on SPKI/SDSI (Simple Public Key Infrastructure / Simple Distributed Security Infrastructure). 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. We describe both protocols in this paper.

1.2 Prototype automation system

Using the ideas described above, we have constructed a prototype automation system which allows for secure, yet efficient, access to networked, mobile devices. In this system, each user wears a badge called a K21 which identifies the user and is location-aware: it "knows" the wearer's location within a building. User identity and location information is securely transmitted to the user's software proxy using the device-to-proxy protocol.

Devices themselves may be mobile and may change locations. Attribute search over all controllable devices can be performed to find the nearest device, or the most appropriate device under some metric.[3]

By exploiting SPKI/SDSI, security is not compromised as new users and devices enter the system, or when users and devices leave the system. We believe that the use of two different protocols, and the use of the SPKI/SDSI framework in the proxy-to-proxy protocol has resulted in a secure, scalable, efficient, and easy-to-maintain automation system.

2 System architecture

The system has three primary component types: devices, proxies and servers. A *device* refers to 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 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.

[3] For example, a user may wish to print to the nearest printer that he/she has access to.

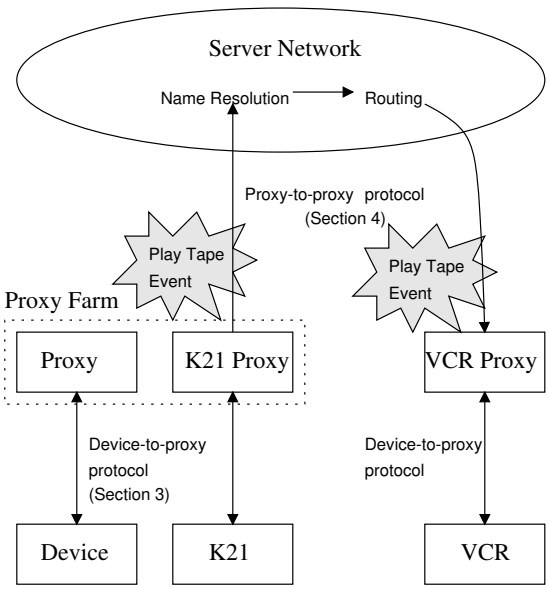

Fig. 1. System Overview

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 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 camera uses a wireless protocol for the same purpose. The K21 (a simple device with a lightweight processor) communicates with its proxy using the particular device-to-proxy protocol described in Section 3. 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 Section 4.

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.

A noteworthy advantage of our proxy-based architecture is that it addresses the problem of viruses in pervasive computing environments. Sophisticated virus scanning software can be installed in the proxy, so it can scan any code before it is downloaded onto the device.

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 the name of the device it represents 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 Communication via events

We use an event-based communication mechanism in our system. That is, all messages passed between proxies are signals indicating that some event has occurred. For example, a light bulb might generate *light-on* and *light-off* events. To receive these messages, proxy x can add itself as an event-listener to proxy y. Thus, when y generates an event, x will receive a copy.

In addition, the system has several pre-defined event categories which receive special treatment at either the proxy or server layer. They are summarized in Figure 2. A developer can define his own events as well. The server network simply passes developer-defined events through to their destination.

CommandEvent Used to instruct a device to turn on or off, for example.
ErrorEvent Generated and broadcast to all listeners when an error condition occurs.
StatusChangeEvent Generated when, for example, a device changes its location.
QueryEvent When a server receives a QueryEvent, it performs a DNS (Domain Name Service) or INS lookup on the query, and returns the results of the lookup in a ResponseEvent.
ResponseEvent Generated in response to a QueryEvent.

Fig. 2. Predefined Event Types

The primary advantage of the event-based mechanism is that it eliminates the need to repeatedly poll a device to determine changes in its status. Instead, when a change occurs, the device broadcasts an event to all listeners. Systems like Sun Microsystems' Jini [26] issue "device drivers" (RMI stubs) to all who wish to control a given device. It is then possible to make local calls on the device driver, which are translated into RMI calls on the device itself.

2.5 Resource discovery

The mechanism for resource discovery is similar to the resource discovery protocol used by Jini. 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 create a new lease with the new 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 a protocol like the device-to-proxy protocol described in Section 3. The proxy begins to broadcast lease-request packets of the form ⟨[name=foo], 10.1.2.3:4011⟩ 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 for wireless devices

3.1 Overview

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 local proxy, a sophisticated protocol for secure device-to-proxy communication is unnecessary, assuming critical parts of the device are tamper resistant. For lightweight devices, the proxy must run elsewhere. This section

[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.

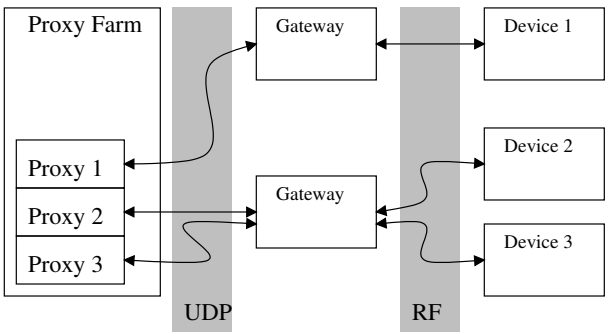

Fig. 3. Device-to-Proxy Communication overview

gives an overview of a protocol which is low-bandwidth and not CPU-intensive that we use for lightweight device-to-proxy communication.

3.2 Communication

Our prototype system layers the security protocol described below over a simple radio frequency (RF) protocol. The RF communication between a device and its proxy is handled by a gateway that translates packetized RF communication into UDP/IP packets, which are then routed over the network to the proxy. The gateway also works in the opposite direction by converting UDP/IP packets from the proxy into RF packets and transmitting them to the device.

An overview of the communication is shown in Figure 3. This figure shows a computer running three proxies; one for each of three separate devices. The figure also shows how multiple gateways can be used; device A is using a different gateway from devices B and C.

3.3 Security

The proxy and device communicate through a secure channel that encrypts and authenticates all the messages. The HMAC-MD5 [13][20] algorithm is used for authentication and the RC5 [21] algorithm is used for encryption. Both of these algorithms use symmetric keys; the proxy and the device share 128-bit keys.

Authentication HMAC (Hashed Message Authentication Code) produces a MAC (Message Authentication Code) that can validate the

authenticity and integrity of a message. HMAC uses secret keys, and thus only someone who knows a particular key can create a particular MAC or verify that a particular MAC is correct.

Encryption The data is encrypted using the RC5 encryption algorithm. We chose RC5 because of its simplicity and performance. Our RC5 implementation is based on the OpenSSL [16] code. RC5 is a block cipher; it usually works on eight-byte blocks of data. However, by implementing it using output feedback (OFB) mode, it can be used as a stream cipher. This allows for encryption of an arbitrary number of bytes without having to worry about blocks of data.

OFB mode works by generating an encryption pad from an initial vector and a key. The encryption pad is then XOR'ed with the data to produce the ciphertext. Since $X \oplus Y \oplus Y = X$, the ciphertext can be decrypted by producing the same encryption pad and XOR'ing it with the ciphertext. Since this only requires the RC5 encryption routines to generate the encryption pad, separate encrypt and decrypt routines are not required.

For our implementation, we use 16 rounds for RC5. We use different 128-bit keys for encryption and authentication.

3.4 Location

Device location is determined using the Cricket location system[18, 17]. Cricket has several useful features, including user privacy, decentralized control, low cost, and easy deployment. Each device determines its own location. It is up to the device to decide if it wants to let others know where it is.

In the Cricket system, beacons are placed on the ceilings of rooms. These beacons periodically broadcast location information (such as "Room 4011") that can be heard by Cricket listeners. At the same time that this information is broadcast in the RF spectrum, the beacon also broadcasts an ultrasound pulse. When a listener receives the RF message, it measures the time until it receives the ultrasound pulse. The listener determines its distance to the beacon using the time difference.

4 Proxy to proxy protocol

SPKI/SDSI (Simple Public Key Infrastructure/Simple Distributed Security Infrastructure) [7, 22] 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. A SPKI/SDSI ACL consists of a list of entries. Each entry has a subject (a key or group) and a tag which specifies the set of operations that that key or group is allowed to perform. To gain access to a resource protected by an ACL, a requester must include, in his request, a chain of certificates demonstrating that he is a member of a group in an entry on the ACL.[5]

If a 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 4, and following is its description:

[5] For examples of SPKI/SDSI ACLs and certificates, see [7] or [3].

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[6] 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,[7] 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 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:

[6] 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.

[7] 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.

(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.[8]
(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 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.[9]

[8] 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.

[9] 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.

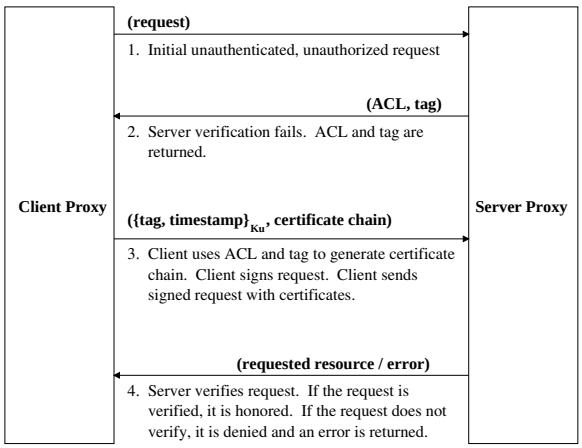

Fig. 4. SPKI/SDSI Proxy to Proxy Access Control Protocol

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 [19] 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 5. In the figure, 'Application 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 re-

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

Fig. 5. Example Layering of Protocols

questing 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* [23]. The access control decisions are made in the uppermost layer, involving only the client and the server.

5 Related Work

5.1 Device to proxy communication

The Resurrecting Duckling is a security model for ad-hoc wireless networks [25, 24]. 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.

5.2 Proxy to proxy communication

Jini [26] 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 [27] 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 [29].

5.3 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 [28]. 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 building certificates, and a module (AuthCompute) for performing authorization computations. OceanStore uses SPKI/SDSI names in their naming architecture.

6 Evaluation

6.1 Hardware design

Details on the the design of a board that can act as the core of a lightweight device, or as a wearable communicator, are given in [15].

6.2 Device-to-proxy protocol

In this section we evaluate the device-to-proxy protocol described in Section 3 in terms of its memory and processing requirements.

Component	Code Size (KB)	Data Size (bytes)
Device Functionality	2.0	191
RF Code	1.1	153
HMAC-MD5	4.6	386
RC5	3.2	256
Miscellaneous	1.0	0
Total	11.9	986

Table 1. Code and data size on the Atmel processor

Memory requirements Table 1 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 device functionality component includes the packet and location processing routines. The RF code component includes the RF transmit and receive routines as well as the Cricket listener routines. The miscellaneous component is code that is common to all of the other components.

The device code requires approximately 12KB of code space and 1KB of data space. The security algorithms, HMAC-MD5 and RC5, take up most of the code space. Both of these algorithms were optimized in assembly, which reduced their code size by more than half. The code could be better optimized, but this gives a general idea of how much memory is required. The code size we have attained 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 2. Performance of encryption and authentication code

Processing requirements The security algorithms put the most demand on the device. Table 2 breaks down the approximate time for each algorithm. The RC5 processing time varies linearly with the number of bytes being encrypted or decrypted. The HMAC-MD5 routine, on the other hand, takes a constant amount of time up to 56 bytes. This is because HMAC-MD5 is designed to work on blocks of data, so anything less than 56 bytes is padded. Since we limit the RF packet size to 50 bytes, we only analyze the HMAC-MD5 running time for packets of size less than or equal to 50 bytes.

We now examine how long it takes the device to receive a packet, process it, and send a response. In this analysis, we assume the device is receiving a packet that has 10 data bytes, making the total packet size 27 bytes, since each packet contains 17 header bytes made up of a 9-byte address field and an 8-byte message authentication field. The device broadcasts at 19.2 Kbps and we encode 8 bits into 12 bits for DC balance. To receive the packet it takes:

$$\frac{\text{packet size} + \text{RF header}}{\text{bandwidth}} = \frac{12 \cdot (27 + 4)}{19200} = 19.38\text{ms}$$

The device then takes 11.48ms to authenticate the packet and $0.163 \cdot 10 + 0.552 = 2.18$ms to decrypt it. Thus, the time for the device to receive a packet and process it is $19.38 + 11.48 + 2.18 = 33.04$ms. The device always sends back a response. In this analysis, we will assume the device responds with a packet of the same size, so the device must encrypt, authenticate, and then transmit the response which will take another 33.04ms. Thus, the device can handle approximately $\frac{1000}{33.04 \cdot 2} \approx 15$ transactions per second. We think that fifteen transactions per second is sufficient for most purposes, with a simple device.

6.3 SPKI/SDSI evaluation

The protocol described in Section 4 is efficient. The first two steps of the protocol are a standard request/response pair; no cryptography

Protocol step	Timing analysis	Approx CPU time
Cert chain discovery	The worst case is $O(n^3 l)$, 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 3. Proxy-to-Proxy Protocol analysis.

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 3 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.

7 Conclusions

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 types of protocols could be included if deemed necessary.

The two protocols we have described have vastly different characteristics, because they apply to different scenarios. The device-to-proxy protocol was designed to enable secure communication of data from a lightweight device. The SPKI/SDSI-based proxy-to-proxy protocol was designed to enable communication between sophisticated devices, whose access control policies can change frequently. The proxy architecture and the use of two different protocols has resulted, we believe, in a secure, yet efficient, resource discovery and communication system.

References

1. William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremey Lilley. The Design and Implementation of an Intentional Naming System.

Operating Systems Review, 34(5):186-301, December 1999.
2. Guruduth Banavar, James Beck, Eugene Gluzberg, Jonathan Munson, Jeremey Sussman, and Deborra Zukowski. Challenges: An Application Model for Pervasive Computing. In *Proc. ACM MOBICOM*, August 2000.
3. Dwaine Clarke. SPKI/SDSI HTTP Server / Certificate Chain Discovery in SPKI/SDSI. Master's thesis, Massachusetts Institute of Technology, 2001.
4. Dwaine Clarke, Jean-Emile Elien, Carl Ellison, Matt Fredette, Alexander Morcos, and Ronald L. Rivest. Certificate Chain Discovery in SPKI/SDSI. *Journal of Computer Security*, 2001. To appear.
5. Steven Czerwinski, Ben Zhao, Todd Hodes, Anthony Joseph, and Randy Katz. An Architecture for a Secure Service Discovery Service. In *Proc. MOBICOM*, August 1999.
6. Michael Dertouzos. The Future of Computing. *Scientific American*, August 1999.
7. Carl 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. Pasi Eronen and Pekka 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. Andrew Maywah. An Implementation of a Secure Web Client Using SPKI/SDSI Certificates. Master's thesis, Massachusetts Institute of Technology, 2000.
15. Todd 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. N. Priyantha. Providing Precise Indoor Location Information to Mobile Devices. Master's thesis, Massachusetts Institute of Technology, January 2001.
18. N. Priyantha, A. Chakraborty, and H. Balakrishnan. The Cricket Location-Support System. In *Proc. ACM MOBICOM*, August 2000.
19. Eric Rescola. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2001.
20. Ronald Rivest. The MD5 Message-Digest Algorithm. Internet Request for Comments RFC 1321, April 1992.

21. Ronald Rivest. The RC5 Encryption Algorithm. In *Proc. of the 1994 Leuven Workshop on Fast Software Encryption*, 2001.
22. Ronald L. Rivest and Butler Lampson. SDSI - A Simple Distributed Security Infrastructure. See http://theory.lcs.mit.edu/ rivest/sdsi10.ps.
23. J. H. Saltzer, D.P. Reed, and D. D. Clark. End-to-End Arguments in System Design. See http://www.mit.edu/∼Saltzer/publications/endtoend/.
24. Frank Stajano. The Resurrecting Duckling – What next? In *Proc. of the 8th International Workshop on Security Protocols*, April 2000.
25. Frank Stajano and Ross Anderson. The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks. In *Proc. Security Protocols, 7th International Workshop*, 1999.
26. Sun Microsystems Inc. Jini Network Techonology. http://www.sun.com/jini.
27. UC Berkeley. The Ninja Project: Enabling Internet-scale Services from Arbitrarily Small Devices. See http://ninja.cs.berkeley.edu.
28. UC Berkeley. The OceanStore Project: Providing Global-Scale Persistent Data. See http://oceanstore.cs.berkeley.edu.
29. University of Washington. Portolano: An Expedition into Invisible Computing. See http://portolano.cs.washington.edu.
30. M. Weiner. Performance Comparison of Public-key Cryptosystems. *RSA Laboratories' CryptoBytes*, 4(1), 1998.