A Proxy-Based Architecture for Secure Networked Wearable Devices

Todd Mills, Matthew Burnside, John Ankcorn, Srinivas Devadas
MIT Laboratory for Computer Science
200 Technology Square, Cambridge, MA 02139 USA
{mills, event, jca, devadas}@lcs.mit.edu

Abstract

We describe the software and hardware architecture for a wearable communicator, and a secure protocol for communication between it and its software proxy. The proxy runs on a fast computer so it is capable of implementing sophisticated cryptographic algorithms, while the wearable communicator contains a simple embedded processor capable of only simple algorithms. The architecture of the wearable communicator is generic, so the same design can be used as a device controller for automation purposes. A prototype automation system using the wearable communicator has been constructed and evaluated. A variety of applications have been implemented on the system including private and public messaging throughout a building, where the messages can be text, audio, or still images. We present an evaluation of our system using scalability, energy and other metrics.

1. Introduction

Attaining the goals of ubiquitous and pervasive computing[3][1] is becoming more and more feasible as the number of computing devices in the world increases exponentially. 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.

For flexibility, the devices must have high connectivity. For convenience, they must be mobile and easily wearable. When connectivity is high, the security of the system is a key factor, since devices in the system must be robust and resistant to tampering.

Devices must only allow access to authorized users and also must keep the communication private. This is important because the devices may be transmitting personal or private information. Implementing typical forms of secure, private communication using a typical public-key infrastructure on devices of this type is difficult because the necessary cryptographic algorithms are CPU-intensive.

We describe a device communication architecture that enables secure devices while utilizing only a small portion of the processing power on the device. Devices built using this architecture are more secure than comparable devices, with only a nominal increase in size and complexity. This is ideal for wearable devices where size, security and power consumption are the most important factors.

This architecture also works well when used as a network of wearable devices such as cameras, displays, input devices, or context sensors. For convenience and flexibility, a user would prefer the devices to communicate wirelessly, rather than wrapping wires around his or her body. Wireless communication is susceptible to eavesdropping, however, and therefore securing these communication channels is important. The architecture described in this paper seamlessly handles this security requirement.

We also describe an implementation of a device called a wearable communicator, based on the previously described architecture. This device uses the Cricket[8] system to determine the user's location. The communicator transmits this information securely to a software proxy running on a trusted machine. The proxy runs scripts for the user, such as a messaging application for routing audio-, text-, or image-based messages to the output device nearest the user's location.

The organization of this paper is as follows: we first describe the architecture and then go on to describe the implementation of the wearable communicator. We present an evaluation of our implementation using factors such as processing power required, scalability, and energy consumption. We then describe some example applications.

2. Related Work

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.

X10 is a simple protocol that allows the control of devices over 110-volt house wiring[14]. It is designed for use with simple devices that only need to be turned on or off; however there are also packages for more complex items such as thermostats. The X10 protocol requires manual configuration of each device and it is not secure. Every device must be manually configured with a unique address. Since X10 implements no authentication or authorization, other people on the same 110-volt power feed can gain unauthorized control of devices.

The Resurrecting Duckling is a security model for ad-hoc wireless networks[12][11]. 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.

The architecture described in this paper is an extension of the resurrecting duckling security model. In this system the master for each device is a software component called a proxy. The proxy can run directly on the device, or remotely over the network. The proxy allows for easy integration with resource discovery systems since the proxy can act as an interface to these systems[2]. In addition, the proxies have the ability to authenticate and authorize certain users for control of the device. The set of authorized users can change in a dynamic and arbitrary manner.

3. Device Architecture

The primary design goal of the architecture is security. That is, the authentication, authorization, and privacy of all communication. An architecture that fulfills this requirement needs an end-to-end security layer, from the user controlling the device to the device itself. In addition, the architecture must be appropriate for the devices being controlled. Enhancing the security of, for example, a wearable camera should not require the addition of expensive processing power. The system must be secure with the addition of, at most, a cheap, simple micro-controller.

Public-key cryptography is ideal for authentication and authorization. Unfortunately, public-key cryptography requires significant computational power. 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 [13]. This is using a 32-bit processor; some of the devices in this system may have 8-bit micro-controllers running at 1-4 MHz, so public-key cryptography on the device itself is simply not an option.

However, public-key based communication between devices over a network is still desirable. 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 which we run on a separate, trusted computer. Between the proxy and the device, we implement a symmetric-key-based security protocol. The proxy can implement sophisticated access control and authentication algorithms, while the device remains simple. Additionally, it is possible to run many proxies on the same computer, allowing the amortization of their cost, since they may require a significant amount of processing power and memory to control access to the device.

3.1. Devices

By focusing on impoverished devices, we handle the base case; more complex devices can be built by incorporating more of the proxy software onto the device itself. The devices are most likely controlled by simple 8- or 16-bit micro-controllers running at 1-4 MHz. The devices typically take control commands as input and output simple state values. For example, a radio has simple input variables such as on/off, tuning the station, and adjusting the volume. It outputs state such as the current station and volume level.

Devices also need a method for communicating with their proxies. A device and proxy can communicate using wireless methods such as radio frequency (RF) or infrared, or they could use a wired solution like Ethernet. Regardless of the medium, a reliable communication protocol is required.

3.2. Proxies

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 can implement computationally expensive security algorithms since it runs on a computer that has significantly more processing capabilities than the device. The proxy can also store large access control lists that would not fit in the device's memory. It uses these mechanisms to act as a guardian; the proxy authenticates users and only allows those with valid permissions to control the device.

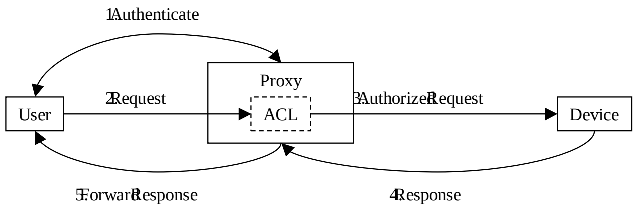


Figure 1. Security model

3.3. Security Model

The proxy and device share a secret key. This secret key allows them to communicate using symmetric-key authentication and encryption. Symmetric-key operations take much less processing power than public-key, so the device can do this computation with a small micro-controller.

All communication passes through the proxy, so it authenticates and then routes communication from the user to the device. The flow of communication is shown in Figure 1 with each step described below.

1. The proxy and user authenticate each other. They also set up a secure communication channel.
2. The user sends his or her request to the proxy.
3. The proxy checks its access control list (ACL) to verify the user is allowed to perform the specified request. If this check succeeds, the proxy forwards the request on to the device. Otherwise, the proxy responds with an error message.
4. The device performs the requested action and sends a response back to the proxy.
5. The proxy forwards the response back to the user.

3.4. Device Initialization

When a device is initialized it must be assigned a proxy and it must obtain a secret key that is shared with the proxy. This is done by physically touching the device to the computer that will run the proxy. When the device is touched to the computer, a proxy is created and the proxy then generates a random secret key that it shares with the device. This initialization is straightforward and easy for the user who is initializing the device. The user does not need to perform any manual configuration.

4. Device Implementation

We have built devices that are wireless, can determine their location, and can communicate securely. The devices can also act as wearable communicators that securely route, among other things, the users' location to their proxies. This section will describe the implementation of these devices, including how they communicate, the security algorithms they use, and how they determine their location. We will also evaluate the performance of the devices in terms of how much memory and processing power is required for their implementation.

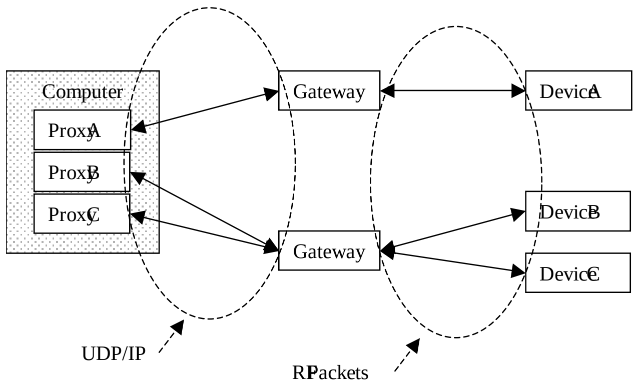


Figure 2. Communication overview

4.1. Communication

Our system uses RF communication because it allows for greater versatility when dealing with device location. RF is also ideal for wearable communicators since the user will obviously be mobile.

RF communication between the 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. Of course, 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 2. 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.

4.1.1. RF Protocol

All communication with the device is initiated by the proxy. This keeps the device's code very simple, since it only has to respond to messages. This protocol also helps keep the RF channel from being over-utilized. If the devices initiated communication, this would lead to a lot of transmission collisions since it is possible for multiple devices to broadcast at the same time. Since the RF channel has a narrow bandwidth, many transmission collisions could lead to severely degraded performance and cause unacceptable delays in communication.

However, since the proxies initiate all communication, the gateway can act as an arbiter over the RF channel. When the gateway receives a message from a proxy, it broadcasts it over the RF channel. Since the device will always respond to a message, the gateway waits 50ms for a response. During this time, the gateway will not transmit any messages over the RF channel, to prevent a transmission collision between the gateway and the device. We assume the gateways have minimal RF overlap so that they can act as the arbiter for their broadcast areas.

To enable reliable communication, the proxy repeatedly transmits the same packet until it receives a response. Every packet has a sequence number and the device responds with the same sequence number to acknowledge that it has received the packet. Once the device acknowledges a packet, it begin looking for packets with the next sequence number. If the device receives a packet with a sequence number one less than the expected number, it re-sends the previous response. This way, the device will only process each unique packet once.

Devices can also be mobile and this presents a problem, since we still want to maintain communication. The problem is that a device can move out of communication range of one gateway and then into the range of another. The device's proxy will not know about this new gateway and so will be unable to contact the device. There must be a mechanism for the device to tell its proxy about the new gateway. So, whenever the device has not received a new packet from its proxy for ten seconds, it begins re-transmitting the last packet once every four seconds. These packets will be routed to the proxy through the new gateway, and from the packet headers, the proxy can determine the address of the new gateway.

4.2. Security

The proxy and device communicate through a secure channel that encrypts and authenticates all the messages. The HMAC-MD5[4][9] algorithm is used for authentication and the RC5[10] algorithm is used for encryption. Both of these algorithms use symmetric keys; the proxy and the device share a 128-bit key.

4.2.1. Authentication

HMAC is a hashed message authentication code (MAC) that produces a MAC that can validate the authenticity and integrity of a message. HMAC uses a cryptographic hash function H, a secret key K, and the message data D. HMAC essentially computes MAC $= H(K, D)$, but the actual computation is a little more complex. Since HMAC uses the secret key only someone who knows that key can create the MAC or verify that the MAC is correct.

HMAC with the MD5 hash function produces a 16 byte MAC. The 8 most significant bytes of the MAC are appended to the end of each packet. This limits the amount of data that must be transmitted with each packet. From a security perspective, this has the advantage of giving less information to an attacker, but the disadvantage of allowing an attacker to have to guess fewer bits. We feel this is a good tradeoff since if all 16 MAC bytes are included in every packet, then even more of each packet would be devoted to authentication instead of useful data.

4.2.2. Encryption

The data is encrypted using the RC5 encryption algorithm. RC5 was chosen because of its simplicity and performance. It does not require tables to speed up processing, as it is primarily XOR and rotate operations.

Our RC5 implementation is based on the OpenSSL[6] code. RC5 is a block cipher, which means 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. Also by using OFB mode, only the encryption routine of RC5 is needed; not the decryption routine.

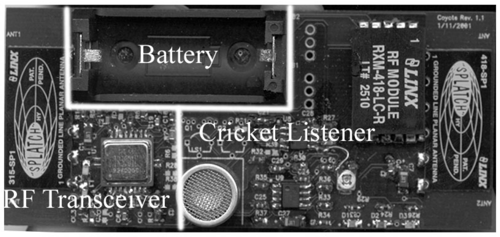
OFB mode works by generating an encryption pad from an initial vector and a key. The encryption pad is then XOR'ed with the data to produce the cipher text. Since $X \oplus Y \oplus Y = X$, the cipher text can be decrypted by producing the same encryption pad and XOR'ing it with the cipher text. Since 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 and the same 128-bit key we used for authentication.

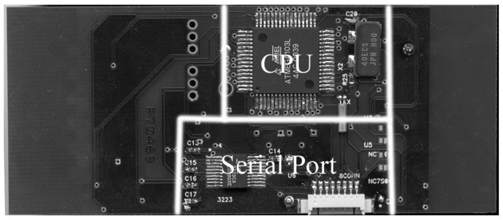
4.3. Location

Device location is determined using the Cricket location system[8][7]. Cricket has several useful features, including user privacy, decentralized control, low cost, and easy deployment. It also works indoors. Cricket's user privacy means that there is no central service where locations are stored. 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 hears the RF message,



Top



Bottom

Figure 3. Picture of the circuit board

it measures the time until it hears the ultrasound pulse. Using the time difference, the listener can determine the distance to the beacon. It determines its location by using the information coming from what it determines to be the nearest beacon.

The Cricket system is easily confused by external sources of ultrasound. Many common sounds such as jangling keys or coins produce ultrasound that can confuse a Cricket listener. It is possible to avoid this problem by sampling for a longer period before making a decision, but this leads to delays from 6-10 seconds before a decision is made.

4.4. Board Design

This section describes a circuit board that acts as the brains of a device, or by itself as a wearable communicator. It contains the necessary components for RF communication, interfacing to the Cricket system, implementing the security algorithms, and interfacing with devices. With a slightly different configuration of software and hardware, the same circuit board can act as a gateway. A photograph of the board is shown in Figure 3; it highlights the major components of the design which are: the battery, RF transceiver, Cricket listener, CPU, and serial port. The current board is 43mm x 102mm; a little large for a wearable communicator but future prototypes will be considerably smaller.

The battery is a 3-volt lithium battery with a nominal capacity of 1,200mAh. This battery has a long life, which means there are fewer battery outages, so debugging the system is simpler. However, it is fairly large, relative to the size of the circuit board. In future boards a coin-type battery will be used to make the board smaller.

The serial port allows the device to communicate with a personal computer, or to control other devices that also have a serial port. The serial port is used by gateways to send and receive RF packets from a personal computer.

The Cricket listener is used by the device to determine its location. This component is not needed on all devices; only devices that need to know their location. It consists of an RF receiver to listen for the location information from Cricket beacons, as well as an ultrasound receiver to listen for the ultrasound pulses.

The CPU was chosen to be representative of the processing power the simplest devices might have. It is the Atmel ATMega103L; an 8-bit CPU that uses the Atmel AVR instruction set and operates at 3 volts. It has 128KB of flash memory, 2KB of RAM, and 512 bytes of EEPROM. It runs at 4MHz. The CPU's flash memory is quite large and may not represent what most simple devices have, but it is useful for software development. This CPU is extremely easy to use, since it only requires a clock crystal and power to operate. All of the memory is internal so the chip size is small. It is programmed via a simple cable plugged into the parallel port of a computer.

The RF Monolithics TR-3001 is used for device to gateway communication. It has a reasonable amount of bandwidth (19.2 Kbps), does not take much current, and does not require many external components.

The Cricket listener uses a Linx Technologies RFM-418-LC, since the beacons use the corresponding transmitter. The Cricket listener operates at 418 MHz, while the device to gateway communication operates at 315 MHz. Thus, there is no interference between them.

4.5. Evaluation

In this section we evaluate our devices in terms of their memory and processing requirements as well as the quality of the RF communication.

4.5.1. 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

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

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.

To further optimize the code size, a smaller authentication algorithm could be used. This system uses HMAC-MD5 which takes up 4.6KB of memory. Another possibility would be to use an authentication algorithm with an encryption routine instead of a hash function. This could significantly reduce code size, since the RC5 code might be reusable in the encryption and authentication. This could potentially reduce the code size to less than 8 KB.

4.5.2. Processing Requirements

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

The security algorithms put the most demand on the device. Table 2 breaks down the approximate time it takes for each of these algorithms to run. 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 and so anything less than 56 bytes is padded. Since we limit the RF packet size to 50 bytes, we only care how long the HMAC-MD5 routine takes for packets of size less than or equal to 50 bytes.

To get an idea of the timing of the device communication, we will examine how long it takes the device to receive a packet, process it, and send a response. For this example, 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, so to receive the packet it takes:

$$\frac{\text{packet size} + \text{RF header}}{\text{bandwidth}} = \frac{12 \cdot (27 + 4)}{19200} = 19.4\text{ms}$$

It then takes the device 11.5ms to authenticate the packet and then 2.3ms to decrypt it, for a total of 33.2ms. The device always sends back a response. In this example, we 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 33.2ms. This means the device can handle approximately $\frac{1000}{33.2 \cdot 2} \approx 15$ transactions per second. Fifteen transactions per second is sufficient for most purposes, with a simple device.

Another optimization that could be performed is the reduction of the size of the secret key. This would speed up the RC5 algorithm. Additionally, the number of rounds performed by the RC5 algorithm could be reduced. Of course, both of these optimizations reduce the overall security of the system.

4.5.3. RF Communication

Unfortunately, the RF communication is the weakest component in the current implementation. On average, with the device and gateway spaced three feet apart, the proxy needs to transmit a packet three times before it receives a response. This is because the RF chip, RFM TR3001, is highly sensitive to noise. A better option would be to use an RF chip with a more robust modulation technique, such as frequency shift keying, that is more immune to interference.

In the previous section we showed that we could support approximately fifteen devices per gateway, based on processor time analysis. However, since the RF does not perform well, in reality it is only possible to support four or five devices per gateway, assuming each device is receiving approximately one packet per second from its proxy. This is a limiting factor in the scalability of the system; the solution is to attenuate the RF signals. Thus, the area covered by a single gateway is reduced, increasing the total possible number of gateways and thereby increasing the total number of devices.

4.5.4. Power Consumption

The board was not specifically designed for low power consumption, but power considerations are significant when it comes to mobile devices. When the RF transceiver is in receive mode, the board draws 22mA of current, or 66mW of power. At this rate, in nominal conditions, the battery will last 54 hours. When the board transmits, it draws 29.5mA of current, or 88.5mW of power. Most of the time, the board is in receive mode. For devices that do not need to know their location, the Cricket listener can be removed to save power. The Cricket listener draws 10mA of current or 30mW of power so removing the listener reduces the board's power consumption by almost half.

Swapping the Atmel ATMega103L for a Microchip PIC16F877 processor would reduce power by 15 mW, but would require considerable compression of the already tightly packed code. Other methods for reducing power including modifying the communication protocol to shut down the RF chips for short periods of time or putting the processor to sleep when it has nothing to do. More details of the device implementation can be found in [5].

We believe that a redesign with off-the-shelf components will result in a wearable communicator with a coin-type battery that lasts for several days. This can be improved even further by building customized silicon (which would have the disadvantage of less flexibility).

5. Applications

The system can be used as the framework to build many different types of applications. In this section we will describe an example application that highlights the functionality and privacy that is provided by the wearable communicator.

5.1. Mobile Audio

We developed a mobile audio application using the above described system. That is, as a user with a wearable communicator moves from room to room, a single audio stream will follow him or her, always playing from the nearest speakers. The wearable communicator is constantly being polled by its proxy, asking for its location. This information is reported to an automation script that runs on top of the proxy.

When an audio stream is sent to the proxy, the automation script uses a directory server to obtain a list of speakers that are reporting their location in the same area as the wearable communicator. The automation script chooses the closest one and redirects the audio. If, at any time, the location of the wearable communicator changes, and hence, the nearest speakers change, the audio output is again redirected. Figure 4 is an overview of the application.

Since the audio is redirected by the proxy (and only the proxy knows the user's location), the user's location is kept private. The proxy could also route other types of information to the user's location such as text messages, or video, while keeping the users actual location private. For other applications, the user could set the wearable communicator's proxy to only give out the location to select people but keep it private from others.

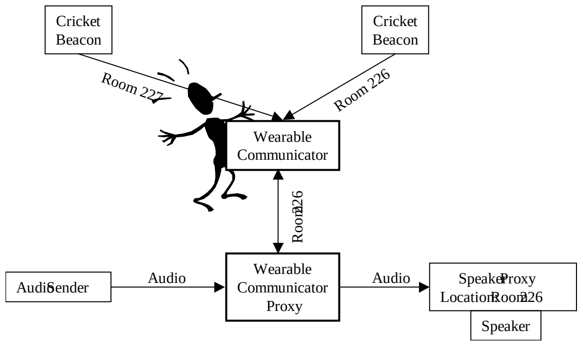


Figure 4. Audio Example Application

5.2. Other Applications

The system can also easily support other applications. For example many customizations can be made upon entering a new room, such as turning on the lights, setting the thermostat, and opening the blinds. Another application would be to customize the desktop of the computer where the user is logged in. The system could also forward phone calls to the phone nearest the user. Or, for privacy, the system could be set up to only print your documents when you are located next to the printer.

6. Conclusion and Future Work

We have described a lightweight infrastructure that allows secure communication between wearable and non-wearable devices. By taking advantage of the computationally simple symmetric-key cryptographic algorithms and introducing a proxy, we keep the additional resources a device needs to be secure very low. Our implementation also uses RF communication, allowing the devices to be mobile.

We have also described an application that uses our implementation. The wearable communicator provides private location information to a proxy that can execute scripts for the user. In our application, the scripts allow sound messages to follow a user as they move between rooms.

Future directions of research include perfecting our current implementation of the wearable communicator. We would like to make it smaller, consume less power, and have more robust RF communication. We would also like to incorporate our architecture into other devices.

7. Acknowledgments

This work was funded by Project Oxygen within the MIT Laboratory for Computer Science.

We thank Ali Tariq for his support of this work. We also thank Dwaine Clarke, Ron Rivest and Anant Agarwal for valuable input during the course of this project. We thank Bodhi Priyantha and Hari Balakrishnan for help with the incorporation of the Cricket system into our implementation.

References

[1] 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.

[2] M. Burnside. Architecture and implementation of a secure server network for Project Oxygen. Master's thesis, Massachusetts Institute of Technology, Work in Progress.

[3] M. Dertouzos. The future of computing. *Scientific American*, August 1999.

[4] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. Internet Request for Comments RFC 2104, February 1997.

[5] T. Mills. An architecture and implementation of secure device communication in Oxygen. Master's thesis, Massachusetts Institute of Technology, May 2001.

[6] OpenSSL. The OpenSSL project. http://www.openssl.org.

[7] N. Priyantha. Providing precise indoor location information to mobile devices. Master's thesis, Massachusetts Institute of Technology, January 2001.

[8] N. Priyantha, A. Chakraborty, and H. Balakrishnan. The Cricket location-support system. In *Proc. ACM MOBICOM*, August 2000.

[9] R. Rivest. The MD5 message-digest algorithm. Internet Request for Comments RFC 1321, April 1992.

[10] R. Rivest. The RC5 encryption algorithm. *Dr. Dobbs Journal*, January 1995.

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

[12] F. Stajano and R. Anderson. The Resurrecting Duckling: Security issues for ad-hoc wireless networks. In *Security Protocols, 7th International Workshop Proceedings*, 1999.

[13] M. Weiner. Performance comparison of public-key cryptosystems. *RSA Laboratories' CryptoBytes*, 4(1), 1998.

[14] X-10.ORG. X-10 technology and resource forum. http://www.x10.org.