Towards Secure Network Programming and Recovery in Wireless Sensor Networks

Prabal Dutta
Jonathan W Hui
David Chiyuan Chu
David E. Culler

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2005-7
http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/EECS-2005-7.html

October 14, 2005

Towards Secure Network Programming and Recovery in Wireless Sensor Networks

Prabal Dutta, Jonathan Hui, David Chu, and David Culler
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720
{prabal,jwhui,davidchu,culler}@cs.berkeley.edu

ABSTRACT

A number of multi-hop wireless reprogramming systems have emerged for sensor network retasking but none of these systems support a cryptographically-strong, public-key-based system for program authentication or any form of recovery from authenticated, but Byzantine, programs. The traditional techniques for authenticating a program and recovering from Byzantine user programs, namely a digital signature of the program hash and hardware-based memory protection, respectively, are not suited to resource-contrained sensor nodes. We present techniques that are consistent with the limited resources of sensor networks, can be used to secure existing wireless reprogramming systems, and allow recovery from Byzantine programs. Our solution to the secure reprogramming problem is based on authenticated streams. A program image consists of several code and data segments that are mapped to a series of messages for transmission over the network. A hash of the first message in this series is digitally signed and the hash and signature are prepended to the series. The signed hash authenticates the first message, which in turn contains a hash of the second message. Similarly, the second message contains a hash of the third message, and so on, recursively binding each message to the one logically preceding it in the series through the hash chain. The solution to the recovery problem requires both on- and off-chip hardware support in the form of a write-protected boot block and a grenade timer. Recovery is enforced by periodically resetting the node which executes a trusted bootloader located in the boot block. We implemented the security and recovery primitives using TinyOS and demonstrated that the overhead incurred is small compared with the cost of network programming.

1. INTRODUCTION

Wireless sensor networks (WSNs) represent a new computing class consisting of large numbers of resource-constrained nodes called *motes* [1, 2, 3] which are often embedded in their operating environments [4, 5, 6], distributed over wide geographic areas [7, 8, 9], or located in remote and largely inaccessible regions [10, 11, 12]. These networks must operate unattended for extended periods of time during which evolving analysis and requirements can change application semantics, creating the need to alter system behavior. While many such changes are possible by varying management parameters [13], executing database queries [14], or downloading scripts [15], more substantive changes require installing new program binaries. However, traditional methods of programming embedded systems cannot scale to large numbers of geographically distributed nodes. Fortunately, the wireless capability of sensor networks coupled with the in-system programmability of many modern microcontrollers allows nodes to be programmed *wirelessly*.

One-hop wireless reprogramming of sensor nodes [16] has been available for a few years. More recently, this one-hop wireless reprogramming capability has been extended and many proposals have emerged for *multi-hop* schemes which enable dissemination of programs through an entire network [17, 18, 19, 20]. Multi-hop wireless reprogramming of sensor networks promises great flexibility and convenience, and enables efficient reprogramming of large-scale, embedded, distributed, and remote systems. However, these benefits come with attendant risks. Since the dissemination protocols typically rely on epidemic algorithms in which nodes propagate newer programs to neighboring nodes, a single faulty or malicious node can expose the entire network to potential security vulnerabilities and irrecoverable fault conditions.

To make wireless programming safe for large-scale sensor networks, we are concerned with achieving two goals. First, given a multi-hop network of wireless sensor nodes, we wish to inject and propagate a program binary from an arbitrary point that allows each node to authenticate the source and verify the integrity of the program. Second, we wish to recover and reprogram a node which has been programmed with a Byzantine application. We provide a more detailed problem definition in Section 2 and an overview of our solution in Section 3.

We reduce the secure dissemination problem to signing (and verifying) digital streams. This problem has a surprisingly simple and efficient solution for a finite stream whose contents are known *a priori* and can be delivered reliably [21]. The solution is to transform the program binary into a series of messages, with each message containing a commitment hash of the next message in the series, and signing the head of the message series. This approach supports an incremental *receive-verify-store* model which is more energy-efficient and far more tolerant to message corruption than the monolithic *receive-store-verify* model used by network programming systems such as Deluge [17]. Section 4 describes our system for secure dissemination of programs.

Our recovery system consists of a grenade timer [22, 3], network bootloader, and lockable storage. The grenade timer periodically resets the node which transfers execution control to the bootloader. The bootloader initializes the radio, queries neighbors for their program version numbers, and decides whether to acquire a new program by comparing the local and network version numbers. If a new program is not available, the bootloader checks the integrity and authenticity of the existing program and transfers control to the program if it passes these checks. If the current program fails the integrity or authenticity checks, the bootloader enters a recovery state in which it retrives the most recent program available from neighboring nodes. If no neighbors are present, then the node periodically retries with decreasing frequency. The details of our recovery system are presented in Section 5.

Secure dissemination and node recovery work hand-in-hand. It does not help to boot to a new program unless one is certain of the authenticity and integrity of the image. Similarly, it does not help to disseminate a program securely if one cannot run it.

The two basic cryptographic operations underlying our security protocol are digital signature verification and hash computations. Our implementation uses RSA signatures [23] and SHA-1 hashes for the security primitives. We ported an existing RSA implementation [24] for the Mica2 to the Telos platform [2] and improved it by incorporating Montgomery reduction. Our implementation is not well optimized for Telos but it can still check RSA signatures in 0.7 s and occupies just 529 bytes of RAM and 1.2KB of ROM. We adapted a Java implementation of SHA-1 to the Telos platform as well. Our SHA-1 implementation can compute the hash of our protocol message in 13 ms and occupies 95 bytes of RAM and 2.3KB of ROM. Although the Telos radio transmits and receives packets faster than we can process them, the processor performance and the processor-radio pathway are the bottlenecks, which means we can process packets at realized channel capacity. This level of performance and overhead makes cryptographically-secure program dissemination feasible for mote-class devices. Evaluation details are presented in Section 6.

We considered a number of tradeoffs in the design of our system. Among them, the choice of digital signature algorithms and sizes, broadcast authenication schemes, and frequency of incremental checking (e.g. per message, per block of messages, or per program image) are among the most important. We chose the RSA signature algorithm because of the asymmetry in the signature and verification operations which map well to the PC-class and mote-class devices which perform these operations, respectively. We chose a broadcast authentication scheme which requires (nearly) in-order message delivery because it provides the minimum number of hashes required to authenticate every message incrementally. We chose to verify *every* message upon arrival and increased the default message size from 36 bytes to 104 bytes to reduce the overhead of a hash in every packet. Our work has led us to conclude that microcontrollers destined for a networked world will require greater hardware protection than today's devices offer. Finding the right balance between no protection and fully protected modes of operation remains an open question. An in-depth discussion of these topics are presented in Section 7.

2. PROBLEM DEFINITION

In this section, we refine the secure and recoverable network programming problem through simplifying assumptions, security and recovery goals, and a threat model. The simplifying assumptions, in particular, makes this problem tractable on the resource-constrained mote-class devices typically found in sensor networks.

2.1 Simplifying Assumptions

The authenticated broadcast protocol used to ensure source authentication and program integrity *does not need to be robust to message loss.* This is the key simplification that makes the secure network problem tractable on resource-constrained sensor nodes. Since *reliable, bulk data transfer* is the distinguishing feature of the network program dissemination problem [20, 17], we can safely assume that the underlying dissemination protocol masks the high packet rates common in dense sensor networks. This assumption is supported by the fact that program images are generally needed in their entirety to be useful, particularly on embedded devices with monolithic images and in dense sensor networks, a node can always recover a lost packet from any one of several neighbors.

We assume that all messages are received in nearly sequential order. Exisiting network program dissemination protocols divide large data objects into smaller pages, and pages into packets, which are transferred sequentially. Packets are transmitted using a windowed protocol and pages must be transferred in full before the next page is transferred.

Secrecy of the program image is not essential. Since, in the absence of tamper-resistant or tamper-proof hardware, a compromised node's memory contents can be accessed, we argue that the secrecy of the program image is difficult to ensure since it is stored on every node. However, we might still like to provide protection against passive eavesdropping. If so, we argue that it is sufficient to encrypt the program image with a network-wide symmetric cipher. But with current hardware, the compromise of any node would compromise the program anyway.

Efficient generation of program images and the corresponding distribution packages is unnecessary since we assume that a PC-class computer generates all of the authentication and integrity information and that this process occurs infrequently. This opens the door to possibility of computationally asymmetric signature schemes like RSA.

Protection against denial-of-service (DoS) is not addressed and a variety of denial of service attacks are not considered. In particular, we do not consider defending against the class of DoS attacks aimed at energy starvation including repeated false announcements, repeated version number requests, combinatorial explosion attacks using forged packets with conflicting sequence numbers, and repeated packet resend requests.

2.2 Security Goals

Source authentication: The source of a program must be verified by a node prior to installation. Conversely, a malicious attacker must not be able to trick a node into installing an unauthenticated program. This ensures that only a trusted source can install a program.

Integrity verification: The integrity of a program must be verified prior to installation. This ensures that a program has not been altered during transit from the trusted source to target node. A node must be able to verify the authenticity, integrity, and freshness of a messages incrementally, assuming messages are received in order. This goal is motivated by the limited RAM available on mote-class devices and the energy cost of optimistically writing unverified data to flash memory. Note that we are not assuming strictly in order packet arrival.

Freshness: An earlier version of a program binary cannot be installed over a program with the same or larger version number. This ensures that a node always installs the most recent version of a program binary.

Compromise tolerance: It must not be possible to use a compromised node to cause an uncompromised node to violate our security goals. In particular, a compromised node must not be able to cause an uncompromised node to install a forged, corrupted, or stale program.

Minimal state: A server should maintain no more state than the server's private key. This property ensures that any server which has access to the private key, or to a service that can sign messages with the server's private key, can generate signed program distributions. The motivation is to avoid schemes which require, for example, precomputing a large hash chain and seeding the nodes with the initial value.

2.3 Recovery Goals

Available: At an aggregate level, network nodes should be available. However, individual nodes may encounter latent bugs, the algorithms they run may fail at scale, or emergent pathological behavior may appear and cause local faults. Even in the face of Byzantine application behavior, the nodes should exhibit *eventual availability*. That is, eventually a trusted program must regain execution control.

Retaskable: A node must exhibit *eventual reprogrammability* over a multi-hop wireless network without human intervention even in the face of Byzantine applications *as long as there are intermediate nodes which provide connectivity between any pair of nodes with differing program versions.*

2.4 Threat model

Sensor nodes are readily interfaced with desktop PC's, so we assume a PC-class attacker with significantly greater computational ability than the nodes themselves. Due to the distributed and embedded nature of sensor nodes, we assume an attacker can compromise an arbitrary number of nodes or introduce an arbitrary number of new malicious nodes. Because nodes communicate wirelessly, we do not trust the wireless medium. An attacker may eavesdrop on, inject, change, delete, and delay packets. We assume that the attacker *cannot* compromise the trusted server which safeguards the private key used for digital signatures.

3. DESIGN OVERVIEW

In this section, we provide an overview of our secure and recoverable wireless programming system for sensor networks. A detailed description of these systems are presented in Sections 4 and 5. We present our system as two separate subsystems: (i) secure dissemination to distribute program binaries among nodes and (ii) recovery mechanisms on the node itself. Both subsystems are required to satisfy the design goals of our system.

Secure dissemination is the basic mechanism to distribute new program binaries to nodes within the network. The security is based on public-key cryptography, with all nodes in a given administrative domain operating under the same public-private key pair. The private key is kept secret on a desktop PC and is used to sign a message that states the intention of disseminating a new program binary. A monotonically increasing version number is included within this message, allowing nodes to determine the freshness of a program. A hash of the entire program is also included to authenticate the image itself. During dissemination, nodes use the server's public key to authenticate the intention of transferring a new image. Because the program binary is generally much larger than the maximum size of a packet supported by the hardware, it is fragmented into a set of messages. It is desirable to authenticate messages as they are received, since the wireless medium is untrusted and potentially malicious. To accomplish this, a reverse hash chain is computed on the desktop PC over the program binary, and allows nodes to efficiently verify whether a given message is truly part of the sequence. The final value of the hash chain is signed using the private key and serves as the hash value representing the entire program binary.

In addition to secure dissemination, recovery mechanisms on the node itself are also required to achieve the design goals. The first component is the trusted network bootloader, whose major responsibilities include initializing the hardware, checking the current version of the program binary, initiating a download of a new binary when necessary, checking the integrity of stored programs, and programming the node. The second component is a grenade timer, which periodically returns control to the trusted bootloader regardless of what program is currently running. This functionality is necessary in microcontrollers that do not provide privileged instructions. The grenade timer is armed by the trusted bootloader. The third component is lockable storage which allows the network bootloader to store updates to the public key beyond the reach of applications.

The security system needs the recovery system to ensure that the secure dissemination software is executed periodically. The secutity system also depends on the recovery system to provide lockable storage. Without such a facility, securely changing server private keys would be more difficult. Conversely, the recovery system's network bootloader uses the same cryptographic primitives that are used in the secure dissemination software. These dependencies underscore the deep relationship between these two systems and justify their presentation in a single paper.

4. SECURITY MECHANISMS

This section presents the design of our security subsystem. At manufacture time, nodes are given signed certificates and pre-loaded with a default program and a low version number. The program dissemination process requires several steps. First, the server takes as input a program in Intel hex or Motorola S-record format and several network parameters like message payload size. The server then queries the network for the version number of the current program. Next, the server produces an ordered list of messages to be transmitted by incrementing the version number, chaining the messages which will comprise the program and its meta-data, and signing the message at the head of the chain which includes the new version number. The server, which is connected to a base station, then transfers the new program to neighboring nodes using any appropriate dissemination protocols including MOAP, Deluge, MNP, or INFUSE. However, these protocols must be modified to include our security primitives and protocol. The dissemination protocols must verify the signature in the head message, transfer messages sequentially, incrementally check messages upon arrival, buffer out-of-order messages optimistically, and use a moving window and negative acknowledgements to notify the sender of missing messages.

4.1 Notation

The symbols A and B are used to represent arbitrary principals, CA represents principal A's certificate, D represents user data, $H(X)$ denotes the hash of X, K a key and K^{-1} its inverse. In a symmetric cryptosystem, $K = K^{-1}$, and in a public-key cryptosystem, K is the public key and K^{-1} is the private key, and (K, K^{-1}) is the key pair. The notation $\{X\}_K$ means X is encrypted under K and $[X]_{K^{-1}}$ means that X is signed with K^{-1}. We use L to represent the length or size of some object, M a malicious attacker, N a nonce, S a server, T a timestamp, and X and Y represent user data.

We use the following notation to define message type M_n as a message intended from A to B with payload X. We ask the reader to forgive our overloaded use of the M symbol:

$$M_n \triangleq A \rightarrow B : X$$

Even though the notation does not explicitly show M_n, A, and B as being included in the message contents, these fields *are* included and available to the receiver. If these fields need to be protected by a hash, message authentication code, signature, or under encryption, these operations will be explicitly shown:

$$M_n \triangleq A \rightarrow B : [A, B, M_n, X]_{K_A^{-1}}$$

This notation is adapted from [25].

4.2 Node Initialization

When a node is first manufactured, the boot block is programmed with the following information:

1. **Globally Unique Identifier:** An identifier that unique identifies the node globally. One possible identifier includes the IEEE GUID which is a 64-bit number constructed from a vendor-specific 24-bit IEEE OUI and a 40-bit vendor serial number. Another possible identifier is Dallas Semiconductor's UniqueWare SerialID products. We use A to denote A's unique identifier.
2. **Factory Authority:** The factory name F, and corresponding RSA public key K_F, is pre-installed on the node during manufacture.
3. **Node Certificate:** Much like an X.509 certificate, a node A's certificate CA consists of the following fields:

$$CA \triangleq ([Z, T, F, A]_{K_F^{-1}}) \quad (1)$$

   Z is the certificate type, and a value of Z_i means that certificate uses format i. Currently, only Z_1 is supported. This format includes the certificate type Z, expiration timestamp T, certificate signatory F, and principal name A, all of which are signed by the signatory's RSA private key K_F^{-1}.
4. **Network Bootloader:** See Section 5.

When control over a node is transferred from the factory to a new server (e.g. the node is sold), the following information is programmed into the node's lockable storage:

1. **Server Authority:** The name of the server S, and the corresponding server RSA public key K_S, is pre-installed on the node during manufacture and may be subsequently changed.
2. **Object Identifier:** The node's object identifier X^{oid} is used to match a node to its program. If X^{oid} is set to a unique value for each network deployed under a particular server authority S, then multiple co-located networks can co-exist since version announcements of one network cannot be used to violate the freshness invariants of another network.

4.3 Program Version Numbering

A node decides whether it should acquire a new version of a program based on whether a more recent version number is available. We assign version numbers by querying the network for the current version number, incrementing this value by one, and setting the next version number to the resulting value. We assume the existence of an out-of-band process to ensure that two or more simultaneous, but conflicting, upgrades with identical version numbers do not occur. However, we note that even if such a process is initiated, nodes will *eventually* acquire whichever conflicting version of the program it first learned about and reject messages from the alternate conflicting version(s). The version number discovery protocol is:

$$M_{V,1} \triangleq A \rightarrow B : X^{oid}$$

$$M_{V,2} \triangleq B \rightarrow A : [S, X^{oid}, X^{ver}, N, H]_{K_S^{-1}}$$

where S is the signatory, X^{oid} is the object identifier, X^{ver} is the object version number, N is a random nonce selected by the server, and H is the hash of the first data message

These fields are signed with the private key of the trusted server, which establishes their authenticity and provides a commitment for the remaining messages.

In this protocol exchange, A sends a version number request to B for the program with identifier X^{oid}. B responds with a cached program announcement message as described below. B originally received this information when it first acquired the program referenced by X^{oid}. If B is the broadcast address, all nodes which receive A's message respond. When A is the server and B is the broadcast address, the protocol describes how the server obtains the list of potential current version numbers. The server then selects the highest-numbered verifiable program version number as the new version number.

The semantics and management of version numbering schemes can become complex. In the case of our motivating application – network reprogramming – we take a rather simple view on generating version numbers. Specifically, the *next* version number of a program should be relative to the *current* version number of the program *within the target network*. This perspective captures the user intent of our application: "I want to program the network with the program image I have right here regardless of what abstract version the network is currently running." Hence, the scope of version numbers is entirely local and only serves to provide a point of reference for nodes when deciding whether to acquire a newer object. This scheme minimizes the state that must be maintained outside of the network.

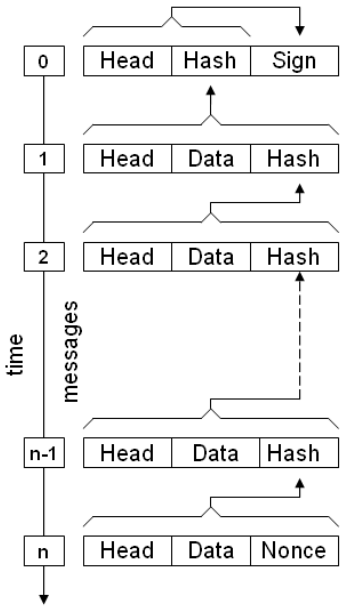

Figure 1: Output of the package creation process. The program binary is split into sequential messages. The hash of message i serves as a commitment for message $i+1$. The first hash, $H_{P,0}$ is authenticated as being from a trusted source, thus the authenticity and integrity of all remaining messages follow by induction.

4.4 Program Packaging for Distribution

The packaging algorithm takes several inputs when preparing a program for distribution. The program to be transmitted is provided as an Intel hex, Motorola S-record, or similar format. A program identifier that uniquely identifies the program (or network) is also included. The packaging algorithm is also given several network and cryptographic parameters including the message payload size, message digest algorithm, public key signature algorithm, and private key with which to sign the package. The output of the packaging algorithm is an ordered list of messages to be transmitted as shown in Figure 1 and listed below.

The contents of the messages shown in Figure 1 are :

$$M_{P,0} \triangleq A \rightarrow B : [S, X^{oid}, X^{ver}, N, H_{P,0}]_{K_S^{-1}}$$

$$M_{P,1} \triangleq A \rightarrow B : X_{P,1}, D_{P,1}, H_{P,1}$$

$$M_{P,i} \triangleq A \rightarrow B : X_{P,i}, D_{P,i}, H_{P,i}$$

$$M_{P,n} \triangleq A \rightarrow B : X_{P,n}, D_{P,n}, N$$

Each message consists of several header fields, a data payload field, and a hash field. The packaging algorithm, which generates this list of messages, works as follows. First, the program is split into pages. Then, each page is further divided into smaller blocks that can fit into the available message payload. Once the data has been appropriately segmented into n such blocks, the messages are built up from the last message, $M_{P,n}$ to the second message $M_{P,1}$, in reverse order. The first message, $M_{P,0}$, identifies the server authority, object identifier, version number, nonce to seed the hash, and the hash of the second message. The hash of the second message is signed and both the hash and signature are included in the first message. The signature authenticates the source of the new program announcement. In practice, the first message may actually span multiple packets depending on the choice and size of the signature algorithm and hash function.

The header field in messages $M_{P,1}$ to $M_{P,n}$ contains X, a data structure which uniquely identifies a data message:

$$X \triangleq (oid, ver, adu, seq) \tag{2}$$

The contents of the data field of the i-th message, $M_{P,i}$, is simply the i-th data block. The hash field H_i, $\forall i \neq n$, contains the hash of the concatenation of a nonce N, headers X, data D, and hash H, of message $M_{P,i+1}$:

$$H_{P,i} \triangleq H(N, X_{P,i+1}, D_{P,i+1}, H_{P,i+1}) \tag{3}$$

The hash field of the n-th message contains a nonce. The same nonce is used throughout a package, but different nonces are chosen at random for different packages. The nonce serves to seed the hash function, which in turn reduces

the required hash size to achieve a given level of protection against pre-image and collision attacks.

Note that each hash serves as a commitment for the next message to be received. Therefore, if the very first hash, $H_{P,0}$ can be authenticated as being from a trusted source, then the authenticity and integrity of all remaining messages follow by induction.

4.5 Program Dissemination

The object dissemination phase can use any reliable, page-oriented scheme like Deluge. However, individual messages must be verified incrementally and *sequentially*, in addition to any integrity checking that the dissemination protocol itself might perform. Messages that are received out of order are buffered optimistically but they cannot be verified until *all* prior messages have been received and verified.

4.5.1 New Package Announcement

A new object announcement consists of transmitting message $M_{P,0}$ created by the packaging algorithm in Section 4.4. A receiving node verifies the signature provided that the object version number is newer than its own version number. If a valid, newer version is received, then the object dissemination phase begins. Otherwise, the node discards the announcement if it contains a version number that is older than the node's current version.

4.5.2 Incremental Checking

A node can check the authenticity, integrity, freshness of a messages incrementally, assuming messages are received in order. To carry out this check, a node compares the hash value $H_{P,i}$, the hash received in the i-th message, with the $H(N, X_{P,i+1}, D_{P,i+1}, H_{P,i+1})$, a hash over the contents of the $i+1$-th message.

Incremental checking is motivated by the limited RAM available on mote-class devices and the energy cost of optimistically writing unverified data to flash memory. Program binaries range in size from a few kilobytes to tens of kilobytes.[1] In many cases, the programs are too large to fit into the available RAM of 4KB, 2KB, and 10KB on the Mica/Mica2, Telos Rev. A, and Telos Rev. B nodes, respectively, requiring the nodes to buffer the program to flash memory prior to verifying the integrity of the binary. In the absence of incremental checking, a *receive-store-verify* operation would be required and these operations would cost dearly in terms of energy usage. Of course, this cost must be borne for an authentic program binary, but a *receive-store-verify* sequence gives an adversary an advantage because the adversary only pays the energy cost of transmitting packets but does not pay the energy cost of writing to flash or verifying.

4.6 Permissions Management

We have heretofore described our system as it pertains to a single immutable trusted server; this principal's public key K_S installed on the nodes at manufacturing time determines all future accesses. This model conveniently generalizes: the principal may extend trust to other principals with distinct public keys through permission management messages:

$$M_1 \triangleq A \rightarrow B : [S', K_{S'}, permissions]_{K_S^{-1}}$$

Note that this does not require infrastructure support like trusted third parties. Our scheme also supports principal transfer functionality. If a permission delegation message adds server S' and removes server S, a transfer of ownership will have occurred.

Permission and key management requires careful control. While trusted software must *access* this data to verify authenticity and integrity of programs, and must *modify* this data to support updates to server public keys, *application code must not be allowed to modify this data.* Supporting this functionality requires some form of hardware protection. Our solution, lockable storage integrated with a grenade timer, is presented in Section 5. Neither write-protected memory nor add-only memory solves this problem if application software can jump to code segments which access these storage facilities.

5. RECOVERY MECHANISMS

Section 4 addressed how to verify program authenticity, incrementally verify program integrity, and ensure version freshness. This section addresses how to recover a node if it is accidentally programmed with a pathological program. Recovery and reprogramming is initiated when a new program version is available so the version number freshness provided by the security mechanism is essential for recovery.

Our recovery system consists of a grenade timer, network bootloader, and lockable storage. The grenade timer periodically resets the node which transfers execution control to the bootloader. The bootloader, which is located in a write-protected memory section of the microcontroller, initializes the radio, queries neighbors for their program version numbers, and decides whether to acquire a new program by comparing version numbers. If a new program is not available, the bootloader checks the integrity and authenticity of the existing program and transfers control to the program if is passes these checks. If the current program fails the integrity or authenticity checks, the bootloader enters a recovery state in which it retrives the most recent program available from neighboring nodes. If no neighbors are present, the the node periodically retries with decreasing frequency.

5.1 Grenade Timer and Lockable Storage

Our grenade timer circuit is shown in Figure 2. This design is based on the ideas outlined in [22] and the implementation used in the XSM [3]. However, our design improves upon the XSM design in two ways. First, our design does not suffer from the "stuttering reset" problem of the XSM design because we incorporate a hardware interlock which prevents the grenade timer from resetting the processor as long as the timer has not been armed. Second, we incorporate lockable storage into the design to allow a trusted server's public key to be safely stored beyond the reach of application programs. We use the same hardware interlock to allow the bootloader

[1] The TinyOS `Blink` application, which simply toggles an LED, requires 1,644 bytes on the the Mica2 platform and 2,672 bytes on the Telos. The `CntToLedsAndRfm`, which increments a counter, displays the results on three LEDs, and transmits the counter value over the radio requires 10,948 bytes on the Mica2 and 11,540 bytes on the Telos.

to read and write keys to the lockable storage but prevent the application from accessing these keys once the grenade timer is armed. Without lockable storage, server keys cannot be trusted since a malicious application can modify a public key that resides in unprotected memory. Additionally, we provide a detailed description of the circuit's theory of operation which should aid others designing similar circuits.

Otherwise, our design provides the same features as the XSM grenade timer. An asynchronous trigger allows the grenade timer to be fired by the application program and force a node reset. An adjustable timeout allows the reset frequency to be changed. A lockout ensures that once the grenade timer is started, it cannot be stopped and the lockable storage cannot be accessed.

The grenade timer circuit works as follows. After a power-on-reset (POR), capacitor C_2 begins charging through R_4 from an initially discharged state. As long as the voltage across C_2 is below V_{IH}, the high-level input voltage of the AND gate (U_6), the output of the AND gate remains low. The FIREN should be tri-stated during a reset and hence tracks the voltage across capacitor C_2. The AND gate's other input is pulled high by R_2 since the INT output of the DS2417 (U_1) is asserted low only during an interrupt interval and not during a POR.

The time constant, τ, of the R_4C_2-circuit is 1ms and the equation for the voltage across capacitor C_2 is:

$$V(t) = V_{CC}(1 - e^{-t/\tau}) \tag{4}$$

Rearranging to solve for t, we have:

$$t = -\tau \ln\left(1 - \frac{V(t)}{V_{CC}}\right) \tag{5}$$

At a supply voltage of 3V, the AND gate's V_{IH} is 2.1V. Substituting 3V and 2.1V for V_{CC} and $V(t)$, respectively, gives $t = 1.2$ms. Therefore, after 1.2ms, both of the AND gate's inputs are high and the AND gate's output goes high as well, allowing the processor to exit the reset state and begin program execution.

The output of the AND gate is also connected to the asynchronous clear input of the D-type flip-flop U_2. By waiting 1.2ms before asserting this line, the power is given enough time to stabilize before the flip-flop's state is cleared (set low). Whenever the flip-flops's output, Q, is low, the multiplexer/analog SPDT switch (U_4), connects the processor's ONEWIRE signal to the DS2417's DIO pin, allowing the processor to communicate with the DS2417, a real-time clock with a built-in timer. To start the grenade timer, the bootloader loads the value of T_{fizz} into the DS2417 using the Dallas 1-wire bus ONEWIRE and enables the device. The legal T_{fizz} values are: 1s, 4s, 32s, 64s, 2048s (34.13min), 4096s (68.27min), 65,536s (18.30hrs), and 131,072s (36.41hrs).

The bootloader or application code can start the grenade timer and ensure that the no subsequent operation can alter or disable the grenade timer, by asserting the LOCK signal high. Doing so creates a low-to-high transition which has the effect of clocking the positive edge-triggered flip-flop, U_3. Once clocked, the flip-flop output, Q, assumes the value of its input, D. Since D is tied to V_{CC}, the value of Q goes after the first clock and remains high until it is asynchronously cleared.

Once Q is high, the multiplexer/analog SPDT switch (U_4), disconnects or "locks out" the processor from communicating with the DS2417 and DS2433. No additional clocking can reverse this latch-out of the processor until the next DS2417 interrupt occurs or the processor asserts FIREN low, both of which asynchronously clears the D flip flop, resets the processor (if LOCK was previously asserted), and returns control to the bootloader. We note that if neither the bootloader nor the application code asserts LOCK high, the DS2417 and DS2433 remain accessible to the processor and can be used as a real-time clock and storage, respectively.

A second multiplexer/analog switch SPDT switch, U_5, was added to our design to compensate for the "stuttering reset problem" described in the XSM design [3]. This switch implements a hardware mutex; the processor can either configure the grenade timer or be reset by the grenade timer but the processor cannot be reset by the grenade timer while the timer is unlocked.

5.2 Network Bootloader

A grenade timer [22] periodically resets the microcontroller. If the boot block of a microcontroller is write-protected, then a grenade timer can be used to transfer control to a *trusted* bootloader. Our recovery process is implemented in the *network bootloader*, or simply *bootloader*, which is invoked immediately after a node is reset. The network bootloader consists of a minimal network stack, a network reprogramming module, the grenade timer drivers, and a small application. The bootloader is responsible for the following operations:

1. **Hardware Reset:** Causes the microcontroller to reset, the program counter to jump to the reset vector, and bootloader execution to commence.

2. **Hardware Initialization:** The bootloader initializes the radio.

3. **Version Checking:** Check the version of the current application image. The current image's version number should be stored in an area of non-volatile memory that is protected from the application code.

4. **Availability Checking:** Check with neighboring nodes for a newer version of the application image by broadcasting a request and listening for a response.

5. **Download:** Initiate the downloading of a new application image, if any, from a neighboring node. Keep track of the application image version in a manner that preserves authenticity.

6. **Integrity Checking:** Verify the integrity of a new application image and its version number through cryptographically secure techniques.

7. **Programming:** Move or copy the newly downloaded application image to make it the default image.

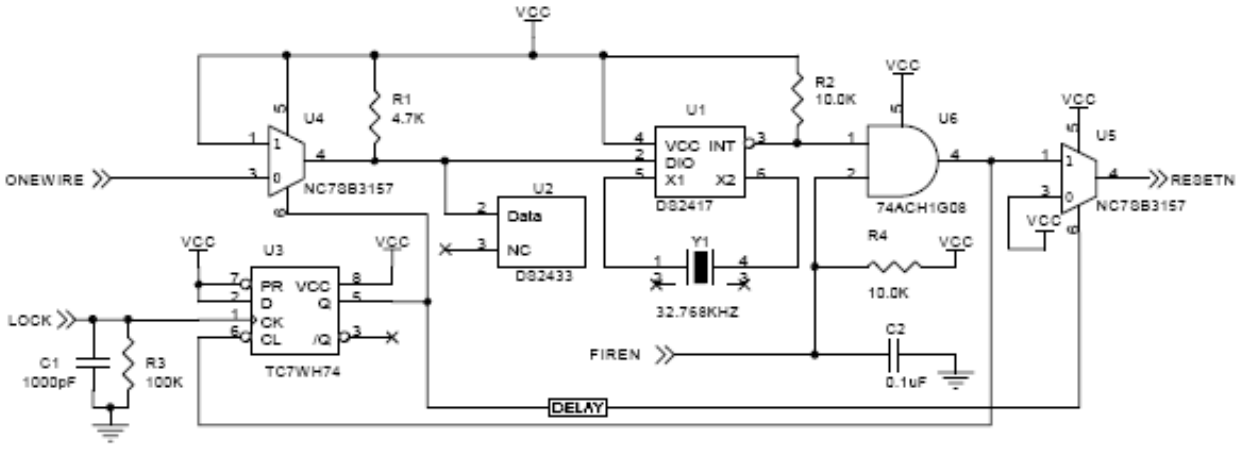

Figure 2: Grenade timer.

8. **Arm (Grenade) Timer:** Enable and arm the (grenade) timer to interrupt (reset) the processor in after some period of time. Disable any further operations on the (grenade) timer.
9. **Load Application:** Jump to the application entry point and begin execution.

6. EVALUATION

To evaluate the feasibility of our system on mote-class devices, we wrote a program packaging application in Java and implemented or ported each of the required security primitives to Telos rev B [2], a TinyOS supported hardware platform. The Telos node contains a 8MHz, 16-bit MSP430 microcontroller, offers 48KB of flash memory and 10KB of RAM, and communicates via an 802.15.4 radio operating in the 2.4GHz ISM band and capable of transmitting at 250Kbit/s [26].

6.1 Primitives

For experimental purposes, we used publicly available code when possible for each of our primitives and ported it to the TinyOS environment as needed. Specifically, we experimented with RSA and SHA-1 and the results are summarized in 1. We do not provide computation times for the base station since it is a PC-class device.

For RSA, we implemenged the algorithm in C code as described in [27], including the use of Montgomery reduction to speed modular exponentiation. Using $e = 3$ and a key size of 1024 bits, a modular exponentiation takes an average of 0.7 seconds and represents the time to regenerate the plaintext. Our implementation has a 40% higher running time than the results reported in [27], which were based on a highly optimized assembly-language implementation. The code we used was optimized for 8-bit CPUs and does not take advantage of the MSP430's 16-bit core. We believe comparable or better results are possible after making these optimizations. We note a significant performance improvement after incorporating Montgomery reduction – running time for a signature verification decreased from 1.5 s to 0.7 s, RAM usage decreased from 755 bytes to 529 bytes, and ROM usage decreased from 2.7 KB to to 1.2 KB.

Primitive	Time	RAM (bytes)	ROM (KB)
RSA-1024	0.7 s	529	1.2
SHA-1	0.013s	95	2.3

Table 1: Summary of Primitives. Time is the time elapsed to complete one primitive. The primitive for RSA-1024 is a modular exponentiation with $e = 3$ and SHA-1 is computing a 160 bit hash with 64-byte blocks over a stream of 64-bytes.

For SHA-1, we used publicly available code implemented by Chuck McManis which implements the operation as described in FIPS PUB 180-1. SHA-1 produces a 160-bit message digest for a given data stream. The code consumes 95 bytes of RAM, 2.3 KB of ROM, and takes an average of 13 ms to hash a stream of 64 bytes using 64-byte blocks. Once again, no optimizations were made for the MSP430, and we believe performance could be improved if optimizations targeted to the MSP430 were implemented. However, the current performance is sufficiently strong. While the radio can theoretically receive packets at a rate faster than we can compute SHA-1 hashes, this effect did not surface in practice. We may be able to use a block cipher like AES in Davies-Meyer [28] mode in place of the SHA-1 hash to improve performance.

6.2 Complete Operation

We implemented the system described in this paper and demonstrated that our ideas work. To test the feasibility of integrating our system with Deluge, we included RSA for announcement authentication and SHA-1 for data message authentication with the Deluge protocol. The current prototype implementation includes support for our version numbering scheme, program package creation, program announcement, and program dissemination as described in Sections 4.3 through 4.5. For this prototype version, we only included single-hop support. Multi-hop support would not only require utilizing the external flash chip on the Telos, but any attacks on the Deluge protocol are essentially single-hop attacks. Securing the single-hop case is sufficient for maintaining security.

Using the prototype integration of our system with the Deluge protocol, the observable overhead includes the cost of verifying an announcement (0.7 seconds), an additional 8 bytes per packet for the SHA-1 hash, and the additional RAM and ROM consumption. As we suspected, the SHA-1 hash computation over each incoming packet did not limit the reception rate of the radio.

6.3 Sufficiency of the Design

We show our working system satisfies the security goals outlined in the problem definition in Section 2. We briefly list the goals and describe how our system satisfies each goal.

- **Source authentication:** Met by virtue of the RSA digital signature.
- **Integrity verification:** Met by virtue of the resistance of SHA-1 to pre-image attacks.[2]. Under our hash chain construction, each hash is a commitment for the next message, permitting incremental message authentication.
- **Freshness:** Ensured by using monotonically increasing and digitally signed version numbers.
- **Compromise tolerance:** No secrets are stored in a node (we assume the source code to the bootloader and application program are widely available) so compromising a node does not give an adversary any additional information or advantage. Similarly, the base station does not transitively delegate its trust to nodes in the network.
- **Minimal state:** A server only stores its private key.
- **Available:** The grenade timer periodically resets the node at which time control returns to the bootloader. If the the boot block can be write-protected, as is the case with the Atmel ATmega128L, then the bootloader can be trusted and availability is enforced. If the boot block cannot be write-protected, as is the case on the TI MSP430, then availability cannot be guaranteed.
- **Retaskable:** The bootloader contains the network programming code so if its integrity is preserved and it eventually gains execution, then retaskability is enforced.

7. DISCUSSION

We considered a number of tradeoffs in the design of our system. In this section, we discuss some of these design tradeoffs and present several observations. We begin with the main obstacle to recoverable operation in most 8- and 16-bit microcontrollers – the lack of protected operation. Mechanisms common on 32-bit processors, like memory protection and privileged instructions which protect the operating system from applications and applications from each other, are not available on most 8- and 16-bit microcontrollers like the Atmel ATmega128L processor used in the Mica2, MicaZ, and XSM platforms and the Texas Instruments MSP430 used in the Telos platforms. Lacking hardware protection, applications can take nearly complete control over the hardware, disable timers, turn off interrupts, and leave the operating system with no mechanism to preempt a misbehaving application.

Hardware-based solutions to the problem have been suggested as well. The Mica mote [1] uses a coprocessor to reprogram the main processor, which ensures that the application can always be replaced. The eXtreme Scale Mote (XSM) [3] uses a grenade timer [22] to ensure that a bootloader eventually regains execution control. The XSM bootloader can detect a golden gesture – three manual resets in quick succession – and revert to a factory image. If the XSM's grenade timer fires, the node invokes a special management program which waits for commands to be issued but no automatic recovery mechanism exists.

A number of software-based techniques have been proposed to protect systems from application software including virtual machines (VMs) [15], proof-carrying code (PCC) [29], and software fault isolation (SFI) [30]. Both VMs and PCC incur a level of overhead that is unacceptable for our embedded application. Results reported in [15] range from 33× to no overhead in execution time for interpreted code, although the lower end of this range only emerges when highly expressive and largely application-specific VM operations are invoked. Results reported in [29] show that a JVM (no JIT) incurred a 41× overhead in execution time and that PCC incurred an overhead ranging between 3× and 8× of raw code size. All of these techniques have drawbacks or caveats, which makes them less then suitable for our problem. None of these approaches completely obviate the need to change program binaries and they have their own problems as well.

Our work has led us to conclude that microcontrollers destined for a networked world will require greater hardware protection than today's devices offer. Finding the right balance between no protection and fully protected modes of operation remains an open question. We offer the following ideas for consideration.

Protected Trusted Computing Base: Ensuring that the trusted computing base, minimally consisting of a network bootloader and public keys for authentication, cannot be altered by the application software appears to be the obvious first step. Many microcontrollers, like the Atmel ATmega128L, provide protected pages in onboard flash memory, specifically the bootloader section. However, many other microcontrollers, like the Texas Instruments MSP430 do not. Additionally, any non-volatile state of the trusted computing base should be protected from application code. We suggest that manufacturers more broadly support protected pages in flash memory and a mechanism similar to base+offset memory protection.

Protected Timer and Interrupt: The purpose of the grenade timer is to provide a periodic and guaranteed interrupt. We observe that the same thing could be accomplished with a protected timer in the microcontroller. A timer that can only be started, stopped, and altered from code executing in the protected block, coupled with a similarly protected interrupt that would vector to operating system code, could

[2]We note that the recent attacks against SHA-1 have been collision attacks

replace the grenade timer and eliminate the unpleasant side-effect of resetting a device in the process.

Non-maskable Interrupt: Atmel could have provided a non-maskable external interrupt (i.e. an interrupt that cannot be disabled at all). A non-maskable interrupt, when coupled with the bootloader protection mechanisms and either a protected internal timer or an external source of interrupts, could provide a suitable quasi-protected mode. We do recognize that in certain embedded applications, all interrupts are legitimately disabled during execution of the interrupt handler for a variety of quite valid reasons. However, we argue that it is reasonable to assume that an upper bound exists on the amount of time that any interrupt handler, or atomic code block, is executing within a critical section. If this upper bound can be expressed in clock cycles, we envision that a write-once register could be used to store the maximum amount of time allowed between a non-maskable interrupt being triggered and either enabling interrupt or execution control being forcibly transferred to the non-maskable interrupt handler. Perhaps such features, which can be found in some processors today, will become more common in future microcontrollers.

Disabling Jumps into the TCB: If application code were to allowed to arbitrarily jump into the trusted code, it is conceivable that such jumps could bypass any safety checks and modify the protected flash pages, timers, or interrupts. To guard against such threats, the hardware might disable jumps or calls into the protected memory pages. System calls might be possible through software traps which safely transfer control to a well-known protected interrupt handlers.

A serious drawback to our approach is that when the grenade timer fires, the node is reset and all application context is lost. While this behavior might be preferred for a Byzantine application, it is highly disruptive to the normal operation of an application. Since it takes time and energy to build neighbor tables, estimate link qualities, and synchronize clocks, this kind of state is lost during an unexpected reset. One way to mitigate this problem is to have the application preemptively fire the grenade timer and reset the timeout period. The application can choose a time that is convenient for it – perhaps when it expects to be asleep. If there is non-volatile storage available to the application, as is the case on Mica, Mica2, MicaZ, and Telos motes, then the application save its state to durable storage before firing the grenade timer. An alternate approach might be to insert a delay line on the grenade timer RESETN signal and route the pre-delay signal to an interrupt. This would serve the same functionality as the UNIX `shutdown` command.

The choice of a digital signature algorithm and sizes was an area we explored. We considered ECC based public key operations but had difficulty getting the performance of the ECC implementation presented in [31] to match the performance claimed in [27]. We suspect this is due to our limited experience and interest in optimizing code for 8- and 16-bit microcontrollers. Furthermore, the ECC implementation presented in [27] was not available to us. We were originally interested in ECC because of its small key size and nearly equal encryption and decryption speeds. In the final analysis, we chose RSA digitial signatures because we has access to a prototype implementation [24] and because of the asymmetry in the signature and verification operations which mapped well to the PC-class and mote-class devices which perform these operations, respectively.

We chose a broadcast authentication scheme which requires (nearly) in-order message delivery because it provides the minimum number of hashes required to authenticate every message incrementally. We considered several other authenticated broadcast protocols [32, 33, 34, 35] but none fit the particulars of the program dissemination problem. We chose to verify *every* message upon arrival and increased the default message size from 36 bytes to 104 bytes to reduce the overhead of a hash in every packet.

In Section 4.5.2, we discussed that a receive-verify-store sequence is preferable due to the high cost of flash storage operations. However, another interesting question is whether it is necessary to verify each packet as the are received. With the current reverse hash chain mechanism, data block $n-1$ must be received in order to verify data block n. Increasing the size of the data block and partitioning it across multiple packets is attractive since the cost of including the hash value is amortized over the entire data block. However, the drawback is that the entire data block must be discarded when verification fails. Thus, the bandwidth an attacker must expend to cause a failed verification is inversely proportional with the size of the data block.

8. CONCLUSION

We present and evaluate a system for securing network programming and recovering from Byzantine applications in resource-constrained wireless sensor networks. Our work demonstrates the feasibility of adding public key-based program source authentication to existing network programming services. In particular, we demonstrate that it is possible to verify a digital signature in 0.7 s and verify the authenticity and integrity of messages used to wirelessly transfer a program at wire speeds. Our secure program dissemination protocol can be readily generalized to solve the problem of *secure, reliable, bulk data dissemination* in sensor networks.

Network programming is essential for large-scale sensor networks which are often embedded in their operating environments, distributed over wide geographic areas, or located in remote and largely inaccessible regions. In addition, network programming provides great convenience and flexibility by making reprogramming simple. However, this simplicity also raises multiple security issues. Among the most damaging is that a single malicious or faulty node can change the code of the entire network, either intentionally or accidentally.

The usefulness of multi-hop wireless reprogramming is undeniable but the risks are real. We have experienced firsthand many times that forgetting to "wire-in" the reprogramming module almost instantly turns a wirelessly reprogrammable network into one that reprogrammable only manually. We are aware of a situation in which one large scale Deluge-enabled application was poisoned by an incompatible program from another large scale application. Apparently, a few nodes from the first network wirelessly and unsafely co-

mingled with some nodes from the second network in a hotel lobby while being transported by the two project teams. The infected nodes, when subsequently repatriated, infected the remaining nodes in their network. Experiences like this will be repeated if we fail to secure our sensor networks.

As sensor network research progresses, we are witnessing the emergence of testbeds consisting of tens, hundreds, or thousands of nodes [36, 37, 3]. Since the purpose of a testbed is to develop, test, and characterize experimental platforms and algorithms, it is not only possible but likely that latent bugs may exist, that algorithms may fail at scale, or that emergent pathological behavior may appear and lock up the network. As testbeds become untethered, the risks becomes more severe. These new realities underscore the need for a recoverable operation in large-scale networks.

We have begun integrating the security aspects of our proposal into Deluge, the standard TinyOS network programming component, to create a "Secure Deluge" variant. Our goal for the integration is to evaluate the performance and overhead of the security primitives in side-by-side multi-hop programming experiments. In particular, we are interested in total overhead, which includes latency, bandwidth, and energy. We are building a testbed on the order of 100 nodes to conduct such experiments. These nodes will include the grenade timer and lockable storage hardware.

9. REFERENCES

[1] Jason Hill and David Culler, "Mica: A wireless platform for deeply embedded networks," *IEEE Micro*, vol. 22, no. 6, pp. 12–24, 2002.

[2] Joseph Polastre, Robert Szewczyk, and David Culler, "Telos: Enabling ultra-low power wireless research," *The Fourth International Conference on Information Processing in Sensor Networks: Special track on Platform Tools and Design Methods for Network Embedded Sensors (IPSN/SPOTS)*, Apr. 2005.

[3] Prabal Dutta, Mike Grimmer, Anish Arora, Steven Bibyk, and David Culler, "Design of a wireless sensor network platform for detecting rare, random, and ephemeral events," *The Fourth International Conference on Information Processing in Sensor Networks: Special track on Platform Tools and Design Methods for Network Embedded Sensors (IPSN/SPOTS)*, Apr. 2005.

[4] Robert Szewczyk, Alan Mainwaring, Joseph Polastre, and David Culler, "An analysis of a large scale habitat monitoring application," in *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, Nov. 2004.

[5] Ning Xu, Sumit Rangwala, Krishna Chintalapudi, Deepak Ganesan, Alan Broad, Ramesh Govindan, and Deborah Estrin, "A wireless sensor network for structural monitoring," in *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, Nov. 2004.

[6] J.A. Paradiso, J. Lifton, and M. Broxton, "Sensate media - multimodal electronic skins as dense sensor networks," *BT Technology Journal*, vol. 22, no. 4, pp. 32–44, Oct. 2004.

[7] Tian He, Sudha Krishnamurthy, John A. Stankovic, Tarek Abdelzaher, Liqian Luo, Radu Stoleru, Ting Yan, Lin Gu, Gang Zhou, Jonathan Hui, and Bruce Krogh, "Vigilnet:an integrated sensor network system for energy-efficient surveillance," *In submission to ACM Transaction on Sensor Networks*, 2004.

[8] Cory Sharp, Shawn Schaffert, Alec Woo, Naveen Sastry, Chris Karlof, Shankar Sastry, and David Culler, "Design and implementation of a sensor network system for vehicle tracking and autonomous interception," in *Second European Workshop on Wireless Sensor Networks*, Jan. 2005.

[9] Sinem Coleri, Sing Yiu Cheung, and Pravin Varaiya, "Sensor networks for monitoring traffic," in *Forty-Second Annual Allerton Conference on Communication, Control, and Computing*, Univ. of Illinois, Sept. 2004.

[10] Geoff Werner-Allen, Jeff Johnson, Mario Ruiz, Jonathan Lees, and Matt Welsh, "Monitoring volcanic eruptions with a wireless sensor network," in *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN'05)*, Jan. 2005.

[11] Philo Juang, Hide Oki, Yong Wang, Margaret Martonosi, Li-Shiuan Peh, and Daniel Rubenstein, "Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with zebranet," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, 2002.

[12] V. Bokser, C. Oberg, G. Sukhatme, and A. Requicha, "A small submarine robot for experiments in underwater sensor networks," in *Symposium on Intelligent Autonomous Vehicles*, July 2004.

[13] Gilman Tolle and David Culler, "Design of an application-cooperative management system for wireless sensor networks," in *2nd European Workshop on Wireless Sensor Networks*, Jan. 2005.

[14] Johannes Gehrke and Samuel Madden, "Query processing in sensor networks," *Pervasive Computing*, Jan. 2004.

[15] Philip Levis and David Culler, "Mate: A tiny virtual machine for sensor networks," 2002.

[16] Jaein Jeong, Sukun Kim, and Alan Broad, *Network Reprogramming*, University of California at Berkeley, Berkeley, CA, USA, August 2003.

[17] Jonathan W. Hui and David Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. 2004, pp. 81–94, ACM Press.

[18] Sandeep S. Kulkarni and Mahesh Arumugam, "INFUSE: A TDMA based data dissemination protocol for sensor networks," Tech. Rep., Michigan State University, East Lansing, MI, USA, 2004.

[19] Sandeep S. Kulkarni and Limin Wang, "MNP:multihop network reprogramming service for sensor networks," Tech. Rep., Michigan State University, East Lansing, MI, USA, 2004.

[20] Thanos Stathopoulos, John Heidemann, and Deborah Estrin, "A remote code update mechanism for wireless sensor networks," Tech. Rep., UCLA, Los Angeles, CA, USA, 2003.

[21] Rosario Gennaro and Pankaj Rohatgi, "How to sign digital streams," *Lecture Notes in Computer Science*, vol. 1294, pp. 180+, 1997.

[22] Frank Stajano and Ross Anderson, "The grenade timer: Fortifying the watchdog timer against malicious mobile code," in *Proceedings of 7th International Workshop on Mobile Multimedia Communications (MoMuC 2000)*, Waseda, Tokyo, Japan, Oct. 2000.

[23] Ronald Rivest, Adi Shamir, and Leonard Adelman, "A method for obtaining digital signatures and public key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120.

[24] Ronald Watro, Derrick Kong, Sue fen Cuti, Charles Gardiner, Charles Lynn, and Peter Kruus, "Tinypk: securing sensor networks with public key technology," in *SASN '04: Proceedings of the 2nd ACM workshop on Security of ad hoc and sensor networks*. 2004, pp. 59–64, ACM Press.

[25] Martín Abadi and Roger Needham, "Prudent engineering practice for cryptographic protocols," *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 6–15, 1996.

[26] Moteiv, "Telos revision b datasheet," `http://www.moteiv.com/products.php`, 2004.

[27] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shants, "Comparing elliptic curve cryptography and rsa on 8-bit cpus," in *Workshop on Cryptographic Hardware and Embedded Systems*, 2004.

[28] D.W. Davies and W.L. Price, "Digital signature – an update," in *Proceedings of International Conference on Computer Communications, Oct 1984*. 1985, pp. 843–847, North Holland: Elsevier.

[29] George C. Necula, "Proof-carrying code," in *POPL 97*, Jan. 1997.

[30] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham, "Efficient software-based fault isolation," *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5, pp. 203–216, December 1993.

[31] David Malan, Matt Welsh, and Michael Smith, "A public-key infrastructure for key distribution in tinyos based on elliptic curve cryptography," in *First IEEE International Conference on Sensor and Ad hoc Communications and Networks*, Santa Clara, CA, USA, Oct 2004.

[32] Adrian Perrig and Doug Tygar, *Secure Broadcast Communication: In Wired and Wireless Networks*, Kluwer Academic Publishers, 2002.

[33] Adrian Perrig, Ran Canetti, J.D. Tygar, and Dawn Xiaodong Song, "Efficient authentication and signing of multicast streams over lossy channels," in *IEEE Symposium on Security and Privacy*, May 2000, pp. 56–73.

[34] Adrian Perrig, Robert Szewczyk, Victor Wen, David Culler, and J. D. Tygar, "SPINS: Security protocols for sensor networks," in *Seventh Annual International Conference on Mobile Computing and Networks (MobiCOM 2001)*, Rome, Italy, July 2001.

[35] Adrian Perrig, "The biba one-time signature and broadcast authentication protocol," in *Proceedings of the Eighth ACM Conference on Computer and Communications Security (CCS-8)*, Philadelphia PA, USA, Nov 2001, pp. 28–37.

[36] Geoff Werner-Allen, Pat Swieskowski, and Matt Welsh, "Motelab: A wireless sensor network testbed," in *The Fourth International Conference on Information Processing in Sensor Networks: Special track on Platform Tools and Design Methods for Network Embedded Sensors (IPSN/SPOTS)*.

[37] Brent N. Chun, Philip Buonadonna, Alvin AuYoung, Chaki Ng, David C. Parkes, Jeffrey Shneidman, Alex C. Snoeren, and Amin Vahdat, "Mirage: A microeconomic resource allocation system for sensornet testbeds," in *Proceedings of the 2nd IEEE Workshop on Embedded Networked Sensors (To Appear)*, May 2005.