# Securing the Deluge Network Programming System

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

## ABSTRACT

A number of multi-hop, wireless, network programming systems have emerged for sensor network retasking but none of these systems support a cryptographically-strong, public-key-based system for source authentication and integrity verification. The traditional technique for authenticating a program binary, namely a digital signature of the program hash, is poorly suited to resource-contrained sensor nodes. Our solution to the secure programming problem leverages authenticated streams, is consistent with the limited resources of a typical sensor node, and can be used to secure existing network programming systems. Under our scheme, a program binary consists of several code and data segments that are mapped to a series of *messages* for transmission over the network. An *advertisement*, consisting of the program name, version number, and a hash of the very first message, is digitally signed and transmitted first. The advertisement 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, binding each message to the one logically preceding it in the series through the hash chain. We augmented the Deluge network programming system with our protocol and evaluated the resulting system performance.

## Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols; D.4.6 [**Operating Systems**]: Security and Protection

## General Terms

Algorithms, Design, Experimentation, Performance, Reliability, Security

## Keywords

Wireless Sensor Networks, Dissemination Protocols, Network Programming, Authenticated Broadcast, Security


*IPSN'06,* April 19–21, 2006, Nashville, Tennessee, USA.


## 1. INTRODUCTION

Wireless sensor networks (hereafter sensornets) represent a new computing class consisting of *large numbers* of highly *resource-constrained* nodes [8, 16, 32] which are often *embedded* in their operating environments [36, 43, 27], *distributed* over wide geographic areas [15, 34, 4], or located in *remote* regions [42, 18, 3]. 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. Many such changes are possible by varying management parameters [38], executing database queries [10], or downloading scripts [25]. More substantial changes require installing new program binaries using single- or multi-hop wireless network programming schemes [35, 17, 22, 23]. Network reprogramming provides great flexibility and convenience for retasking large-scale, embedded, distributed, and remote systems. Unfortunately, *none of the proposed solutions incorporate cryptographically-strong, public-key-based techniques for verifying authenticity, checking integrity, or ensuring freshness of the program binary.* The goal of this study is to explore whether and how these existing systems can be secured *efficiently.*

Conventional approaches to verifying program authenticity and integrity, and ensuring freshness, are poorly suited to the extreme resource constraints typical of sensornets. First, since wireless is a broadcast medium, an attacker can easily inject or corrupt a packet which passes the CRCs used by link layers. Such an attack can be detected *but not before the entire program has been downloaded.* Since the offending packet(s) cannot be identified, the entire program must be discarded. Second, since most sensor nodes have only a few kilobytes of RAM, yet programs are typically tens of kilobytes, the nodes must buffer the program in larger flash (or EEPROM) memory *before verifying program authenticity, integrity, or freshness.* However, writing to flash memory is expensive [9] and requires a higher operating voltage. Some flash technologies only allow whole sector erases, requiring the entire image to be dropped or major portions of it copied to a different sector, so this buffering exacerbates the cost of a failed verification. Third, in the event of an attack, the energy cost of receiving, storing, and verifying the program is largely expended by the victim at a small expense to the attacker. Section 2 reviews these issues further.

There are three essential elements to *securing* network programming. First, given a multi-hop network of wireless sensor nodes with network programming capability, we wish to augment this system so that it can efficiently *authenticate* a program binary. By authenticate, we mean that a recipient

may be reasonably certain that a binary was actually created by its purported author and not forged by another party. Second, the secure network programming system must be able to verify *integrity* incrementally as each packet arrives. By integrity, we mean that it is computationally infeasible for the binary to be changed in transit without the changes being detected by the recipient. Third, the system must ensure *freshness* in the sense that older versions of a program cannot be installed over a newer installed version. In addition to these goals, privacy may play a role in the design space. Detailed security and efficiency requirements, as well as simplifying assumptions, are presented in Section 3

In contrast with earlier approaches to code signing, our approach transforms the secure network programming 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 [11]. Our key observation is that network programming fits these requirements perfectly: a program binary is a finite length octet string whose contents are known ahead of time. The transformation involves mapping the program binary into a series of messages, with each message containing a cryptographic hash of the next message in the series, and digitally signing the head of the hash chain. This approach supports an incremental *receive-verify-store* model which is far more resistant to attack than the *receive-store-verify* model used by network programming systems such as Deluge [17]. Our design is presented in Section 4.

The two cryptographic primitives used in our security protocol are digital signatures and cryptographic hashes. Our particular implementation uses RSA signatures [33] and SHA-1 hashes, but any other convenient signature and hash scheme could be used. We ported an existing RSA implementation [41] for the Mica2 to the Telos platform [32] and improved its performance by incorporating Montgomery reduction. An empirical evaluation of our cryptographic primitives and security protocol is presented in Section 5.

# 2. RELATED WORK

In this section, we review the sensornet literature on network programming, cryptographic primitives, and security services, as well as the security literature on code signing and authenticated broadcast. Finally, we review concurrent research on secure network programming, highlighting the tradeoffs between three different approaches.

## 2.1 Sensornet Network Programming

A number of bulk data dissemination protocols suitable for network programming have been proposed. These protocols include Multi-hop Over-the-Air Programming [35], Deluge [17], Infuse [22], and Multi-hop Network Programming [23]. The protocols differ in their design choices but they all have two things in common. First, they are used to disseminate a program over a one- or multi-hop sensornet. Second, *none of them were intended to provide security and all are vulnerable to simple, but debilitating attacks.*

Since Deluge is the *de facto* TinyOS network programming system, in the interest of space, we focus exclusively on it in this section. However, the other network programming systems are vulnerable to nearly identical attacks. Deluge requires each node to periodically broadcast the program version available for sending. This advertisement is not authenticated, and any node in the network may advertise any version number. Since the radios in use today support a relatively small number of channels, and TinyOS [39] supports 16-bit group identifiers in the packet header to multiplex access to a channel, an attacker can simply scan the radio channels and listen promiscuously for all group identifiers on each channel to discover a running instance of Deluge. Deluge also exports a simple, one-hop network querying interface. Any unauthenticated node can query the network to obtain the program version number currently installed and the program binary pages received and pending. Without authenticated broadcasts, and given the ability to actively query or passively probe the network state, an attacker can advertise a new version number, disseminate any number of arbitrary packets, program any number of nodes, and hijack the entire network.

Deluge uses 16-bit cyclic redundancy checks (CRC) across both packets and pages to verify program integrity. The CRCs do not protect against an attacker injecting a bogus program or program fragment. Thus an attacker could compromise system integrity in a manner that is undetected by using a CRC. The programs themselves, like the advertisements, are not authenticated. Hence, during the transmission of a legitimate program, an attacker may inject any number of malicious packets. Recipients will be unable to distinguish between the legitimate and spoofed packets.

There are various low-bandwidth, denial-of-service (DoS) attacks available for exploitation as well. For example, a method for DoS is to simply continue increasing the version number. In doing so, a legitimate user would be unable to inject a new program since its version will soon become obsolete. Another example for DoS occurs when an attacker inserts a packet that purposely causes the page-level CRC to fail, which causes the node to drop the entire page and request it again. The main issue is that the node detects one or more corrupt packets, but cannot pinpoint which one, and therefore must request the entire page again.

## 2.2 Security Primitives and Services

TinySec implements link layer security in sensor networks via a network-wide shared secret key [19]. TinySec provides authenticity and secrecy as long as the shared secret key is not compromised. The main difficulty is that every receiver can both *verify* the authenticity of the message as well as *generate* authentic messages. Since current mote-class devices are not tamper-resistent, it is simple for an adversary to physically compromise a node or mount a side-channel attack [21, 20] when a large number of nodes is available, so we were motivated to explore public-key approaches that can tolerate node compromise.

Recent research results have demonstrated the feasibility of public-key cryptography schemes on 8- and 16-bit microcontrollers. Malan et al. have implemented elliptic curve operations over the field $F_{2^p}$, and demonstrated an ECC Diffie-Hellman key exchange [26] on mote-class devices. Watro et al. have implemented modular exponentiation, the central primitive for RSA cryptography [41]. Gura et al. have implemented ECC and RSA primitives on 16-bit processors [13], and have extrapolated these primitives to provide estimated running times of signature and verification operations. Digital signatures [6, 33], the cryptographic solution to the problem of data authentication, are usually based on public-key primitives and are an important mechanism for authenticated broadcast since every receiver can

*verify* the authenticity of the message *without* being able to *generate* authentic messages. The drawback with digital signatures are their large size and long verification time, which make using them for packet-level protection too costly.

Sizzle [12] is a new end-to-end security service that implements the Secure Sockets Layer on motes in a standards-compliant way. Secure sockets use public-key operations to authenticate peers and private key operations to establish private, end-to-end unicast connections. However, the use of end-to-end unicast communications is not compatible with the broadcast-based dissemination algorithms used in Deluge and other network programming systems.

## 2.3 Code Signing

Code signing is a common technique for authenticating the source and verifying the integrity of data and executable files. Most popular approaches to code signing compute a digest of each file, sign the digest, and then distribute the file, digest, and signature together. For example, the Java Archive (JAR) format allows one or more files to be packaged together and digitally signed. A signed JAR file contains a manifest file (`MANIFEST.MF`) which includes the full pathname and digest of each file in the archive, a signature file (`*.SF`) that contains a digest of each attribute in the manifest, and a digital signature file (`*.DSA`) that contains a signature of the `*.SF` file as well as the certificate of the entity that signed the archive.

Conventional code signing techniques are based on digitally signing a hash of the entire program. Since the hash is computed over the entire program, the signature cannot be verified until the program is received in its entirety, which presents several problems. First, if the verification fails, there is no way to identify which particular packet(s) caused the failure because packets are not verified individually and can be spoofed. Second, since most program binaries are larger than the available RAM, unverified packets must be written to power-hungry flash, wasting energy prematurely. Third, in the event of an attack, the energy cost of receiving, storing, and verifying the program is expended by the victim at a relatively small expense to the attacker.

## 2.4 Authenticated Broadcast

The literature in the area of authenticated broadcast provides fruitful ideas but also underscores the difficulty of the problem. Perrig and Tygar [31] present an overview of the authenticated broadcast problem including a scheme proposed by Gennaro and Rohatgi [11] that we adapted for securing network programming. This scheme is often ignored because of its dependence on reliable, in-order delivery. Perrig and Tygar also present TESLA [29] and BiBa [28], their solutions to authenticated broadcast. TESLA, and its sensor network variant $\mu$TESLA, provide broadcast authentication via key chain hashes [30] while BiBa achieves authenticated broadcast via precomputed hash collisions and chains.

TESLA, $\mu$TESLA, and BiBa all rely on *loose time synchronization* and require *dynamic server state* to be maintained in the form of precomputed key chains and the current index within those chains. The protocols follow models in which the data are revealed and then, a short time later, are authenticated. Dependence on loose time synchronization is necessary because these protocols all assume the data are unknown *a priori* and hence cannot be authenticated before they are generated. In the case of network programming, this assumption is false: the data (i.e. the program) are known *a priori*. These schemes usually assume a one-to-all broadcast channel or one with predictable transmission latencies. These assumptions do not hold for network programming. First, many dissemination protocols do not make lock-step, page-by-page consistency guarantees, and instead choose to pipeline pages in a multi-hop network for efficiency reasons. In such cases, a routing wormhole can circumvent the security of these schemes. Second, sensornets regularly experience long periods of disconnected operation [37]. Because of pipelining and disconnected operation, a broadcaster cannot easily determine when it is safe to release the next key in the key chain. If the broadcaster were to distribute the next key before some nodes receive the data for the current round, an attacker could use this key disclosure to forge a signature for an arbitrary data payload. The problem with dynamic server state is that it makes multiple, independently-operating broadcasters impossible.

Perrig and Tygar [31] review several other schemes for authenticated streams that have various underlying assumptions. HTSS, for example, operates under the assumption that the entire data object is known ahead of time and builds a Merkle hash tree over all the packets. The Merkle tree authentication information consists of hashes of all siblings to the path from the packet to the root making it robust to message loss, *but robustness to packet loss is not necessary because Deluge provides reliable dissemination.* The drawback of HTSS is that it requires twice as many hashes as packets, which makes it only half as efficient as [11].

## 2.5 Secure Network Programming

Lanigan et al. have recently and independently proposed Sluice, an approach quite similar to ours [24]. The key difference between their protocol and ours is the granularity of the authentication. Sluice verifies hashes at the page-level rather than at the packet-level. The key benefit of Sluice is that it requires less overhead than our approach since the space, time, and message overhead of a hash can be amortized over the multiple packets that constitute a page. However, the cost of network reprogramming is relatively small compared to the cost of day-to-day operations [7], so the overall advantage of this optimization is limited. The worst-case costs, however, are more severe. Hashing at the page-level exposes a node to the simple, low-bandwidth DoS attacks described earlier. In the event of an attack, there is no mechanism to pinpoint a spoofed packet. Since a single bad packet causes the entire page to be discarded, Sluice is vulnerable to asymmetric attacks in which the attacker expends a small fraction of the effort that the attacked node expends, yet the attacker is able to impede or even halt programming progress. Finally, the cryptographic primitives that Sluice uses are more than an order of magnitude slower than the ones used in our implementation, but a more judicous use of cryptographic primitives, and careful tuning of those primitives, will make Sluice's speed competitive.

Deng et al. have also proposed similar protocols for secure network programming [5]. Their approach allows packets to be verified out-of-order by first sending a hash *tree* over the packets followed by the packets themselves. While the hash tree does not need to be sent with a total order, a partial order *is* required, so their approach relaxes, but does not fully eliminate, the ordering requirements. In addition, their scheme requires greater memory to store the hash tree.

# 3. PROBLEM DEFINITION

In the context of the background and related work, we now formulate a crisp definition of the problem and the requirements for a complete solution.

## 3.1 Security Requirements

- Authenticity. The source of a program must be verified by a node prior to installation, ensuring that only a trusted source can install a program.
- Integrity. It must be possible to ensure that a program has not been altered during transit from the trusted source to recipient.
- Freshness. An earlier version of a program binary cannot be installed over a program with the same or greater version number, ensuring a node always installs the most recent version of a program binary.
- Compromise Tolerant. It must not be possible to use a compromised node to cause an uncompromised node to violate the above security requirements.
- Delay Tolerant. Nodes in the network may experience long periods of disconnected operation so there must be no dependence on time synchronization.

## 3.2 Efficiency Requirements

- Incremental Verification. Authenticity, integrity, and freshness must all be verified before a packet is written to flash.
- No Dynamic Server State. A server should maintain only static state (e.g. a public-key pair) but *not* any dynamic state (e.g. a precomputed key chain and the current position in that chain).

## 3.3 Simplifying Assumptions

- The authenticated broadcast protocol *does not need to be robust to message loss* since the network programming service itself provides reliable dissemination.
- The authenticated broadcast protocol *does not need to tolerate out-of-order delivery* since the network programming service itself buffers and pipelines on page boundaries and uses selective negative acknowledgements to request retransmissions of missing packets.
- Communications privacy is not supported but can be provided by a link-layer encryption service like TinySec [19].
- Computationally efficient digital *signing* of program binaries is unnecessary since a PC-class computer generates such signatures but efficient *verification* is required since mote-class nodes perform the verification.

## 3.4 Threat model

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 secret key used for digital signatures.

# 4. DESIGN AND IMPLEMENTATION

This section presents our design of a *secure network programming* system. First, we present the cryptographic notation used in this section. Then, we present an overview of our design. Finally, we present the details of our design.

## 4.1 Notation

We use the notation presented in Table 1 in the remainder of this section. This notation was adapted from Abadi and Needham [1].

**Table 1: Cryptographic Notation.**

| Symbol | Meaning |
|---|---|
| $A$, $B$ | Principals |
| $S$ | A trusted server |
| $D$, $X$, $Y$ | User data |
| $M_{t,i}$ | A message of type $t$ and (optionally) index $i$ † |
| $CA$ | Principal $A$'s certificate |
| $H(X)$ | A cryptographic hash of $X$ |
| $PK$ | A public-key in a public-key system |
| $SK$ | A secret-key in a public-key system |
| $[X]_{SK}$ | $X$ is signed with $SK$ |
| $N$ | A nonce |

† The notation for a message of type $t$ and (optionally) index $i$ from $A$ to $B$ with payload $X$ is:
$M_{t,i} \triangleq A \rightarrow B : X$

## 4.2 Design Overview

The process used to transform a program to a stream of packets for secure network programming is illustrated in Figure 1. A program is divided into $P$ fixed-sized pages, except possibly the last one. Each page is divided into $G$ fixed-size packets, except possibly the last one. A hash of each packet $i$, is computed and placed in the previous packet, except for the last packet whose hash field contains zeros. The hash of the first packet is *digitally signed* and the hash and signature, taken together, form the *advertisement* packet for the program.

When an *advertisement* packet is received, the receiving node checks the program identification and version number in the packet (not shown in Figure 1 for clarity). If the program information matches and the version number is greater than the one currently running on the node, the node checks the digital signature. If the digital signature is valid, then the node caches the hash and uses Deluge's normal procedure to request the packets in the first page. If packets arrive in order, they are buffered in RAM like Deluge, with one exception: as each packet arrives, the hash over the contents of that packet is checked against the hash in the immediately prior packet. If packets do not arrive in order, then Deluge's selective negative acknowledgements are used to request the missing packets after the initial transmission of the page is complete. Any out-of-order packets are buffered optimistically in RAM. If any inconsistent duplicate packets are received, either the packet with the correct hash is cached (in the case of in-order delivery up to the packet under question) or a coin toss determines the packet to optimistically cache (in the case of out-of-order delivery that precludes immediate verification). If an optimistically cached packet is later discovered to fail verification, it is discarded and a selective negative acknowledgement for that packet is signaled. This

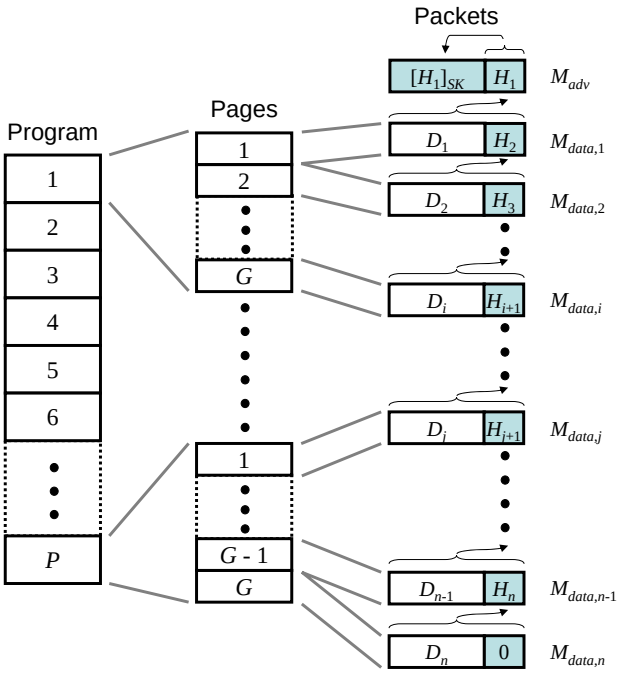


**Figure 1: Illustration of the program to packet transformation used to secure network programming. The shaded boxes represent information added for security. Note: Several details have been omitted from the figure for clarity but will be discussed in the remainder of this section.**

process is repeated for the remaining pages until the entire program binary has been received.

## 4.3 Node Information

A node is pre-programmed with several pieces of information. The name of the trusted server $S$, and the server's public key $PK$, are pre-installed on the node. A node program identifier $X^{pid}$ is used to identify the program a node runs. If $X^{pid}$ is set to a unique value for each application deployed under the aegis of a particular server $S$, then multiple, co-located applications can co-exist.

## 4.4 Preparing the Program for Dissemination

The first step in preparing a program for secure dissemination is to query the version number of the currently running program in the network through a version number request message, $M_{ver}$, broadcast from a network basestation:

$M_{ver} \triangleq S \rightarrow A : S, X^{pid}$

$S$ is the trusted server, $A$ is an arbitrary node, and $X^{pid}$ is the program identifier. The version query triggers an advertisement message:

$M_{adv} \triangleq A \rightarrow S : [S, X^{pid}, X^{ver}, N, H(N, M_{D,1})]_{SK_S}$

$X^{ver}$ is the program version number, $N$ is a random nonce originally selected by the server, and $H(N, M_{D,1})$ is the nonce-seeded hash of the first data message. These fields are signed with the private key of the trusted server, which establishes their authenticity. The largest authenticated value of $X^{ver}$ that is returned from the network becomes the version number of the new program. The nonce serves to randomize the hash function which makes precomputing a pre-image collision computationally infeasible [14].

An application called the packager converts an Intel Hex, Motorola S-record, or any binary representation of the program into an advertisement, $M_{adv}$, and a list of data messages, $M_{data}$, collectively called a *package*:

$M_{adv} \triangleq A \rightarrow B : [S, X^{pid}, X^{ver}, N, H(N, M_{D,1})]_{SK_S}$

$M_{data,1} \triangleq A \rightarrow B : X_1, D_1, H(N, M_{D,2})$

$M_{data,i} \triangleq A \rightarrow B : X_i, D_i, H(N, M_{D,i+1})$

$M_{data,n} \triangleq A \rightarrow B : X_n, D_n, 0$

Each data message consists of several header fields, a data payload field, and a hash field. The packager generates this list of messages as follows. First, the program is split into pages. Then, each page is further divided into packets. Once the data has been appropriately segmented into $n$ such packet payloads, the packets are built up in reverse from the $M_{data,n}$ to $M_{data,1}$. Message $M_{adv}$, the advertisement, identifies the trusted server, program identifier, version number, nonce to seed the hash, and the hash of $M_{data,1}$. The headers in messages $M_{data,1}$ to $M_{data,n}$ contain a composite field, $X$, which uniquely identify the data payload in the message:

$X \triangleq \langle pid, ver, page, pkt \rangle$

The contents of the data field, $D_i$, of the $i$-th data message, $M_{data,i}$, is the $i$-th data block of the program binary. The hash field in the $i$-th data message contains the nonce-seeded hash of the $i+1$ data message. Note that each hash serves as a commitment for the next message to be received. Therefore, if the very first hash, $H(N, M_{data,1})$ can be authenticated as being from a trusted source, then the authenticity and integrity of all remaining messages follow by induction.

## 4.5 Integration with Deluge

The mechanics of dissemination are handled by Deluge [17]. We leveraged the existing implementation and added hooks to the Deluge code to: (i) check the authenticity and integrity of advertisements; (ii) intercept and verify packets as they arrive or as enough information becomes available, and (iii) remove optimistically buffered packets from Deluge's RAM buffer if these packets fail verification.

Our implementation truncates the SHA-1 hash to 64 bits. We believe this hash size represents a reasonable balance between protection and efficiency for a sensor network. Support for 1024-bit RSA signatures is forthcoming but our current implementation supports only 512-bit signatures because only these smaller signatures are compatible with the Deluge single-packet advertisement architecture and the radio's MTU. It is possible to modify Deluge to use several large advertisement packets, but the changes are cumbersome and far-reaching, so we avoided them at this point. We report performance results for a 1024-bit RSA modulus since the 640-bit modulus has been factored by Bahr et al. [2].

# 5. EVALUATION

To evaluate the performance of our proposed approach, we implemented our scheme using TinyOS. We adapted several cryptographic primitives to the Telos Rev. B mote [32], extended Deluge to include authentication and verification hooks to support our security scheme, wrote a desktop application to transform Intel Hex files to the format used in our protocol, and tested the performance on a real network of motes.

## 5.1 Methodology

We performed several experiments to evaluate the performance of our proposal. First, we characterized the static memory and flash footprint using the standard TinyOS build process and module memory usage script. Then, we characterized the runtime performance of the cryptographic primitives by instrumenting them, running test vectors through them, and measuring the running times. After characterizing the cryptographic primitives, we obtained baseline performance statistics for the standard Deluge 2.0 version included in TinyOS. The standard Deluge, labeled *Deluge-23* because of its 23-byte payload, was instrumented to log advertisement, packet, and page reception times, as well as programming completion times. We then modified Deluge-23 to use a 64-byte payload and called the resulting variant *Deluge-64*. The payload size was chosen to be the same as the payload *plus* the hash used in the secure version of Deluge, called *Secure Deluge-56+8*, and instrumented to log packet verification times. The Deluge variants are summarized in Table 2 and all use 5 bytes for the 802.15.4 preamble and SFD, 10 bytes for the TinyOS radio header, and 5 bytes for the Deluge header.

**Table 2: Details of the Deluge Variants.**

| Variant | Header | Payload | Page Size |
|---|---|---|---|
| Deluge-23 | 20 bytes | 23 bytes | 48 packets |
| Deluge-64 | 20 bytes | 64 bytes | 24 packets |
| Secure Deluge-56+8 | 20 bytes | 56 bytes | 24 packets |

We tested our network programming system on the 28-node Telos Rev. B Omega Testbed shown in Figure 2. The Telos mote contains a 16-bit MSP430 microcontroller operating at 8MHz, offers 48KB of flash memory and 10KB of RAM, and provides an 802.15.4 radio that operates at a data rate of 250kbps in the 2.4GHz ISM band. We used the default TinyOS MAC backoff algorithm which selects an initial backoff $\sim \mathcal{U}[305\mu s, 4880\mu s]$ and a congestion backoff $\sim \mathcal{U}[305\mu s, 19520\mu s]$. All experiments used CC2420 transmit power level = 3. This power level results in a three hop network and represents a balance between a fully-disconnected network and a fully-connected one.

## 5.2 Cryptographic Microbenchmarks

We used publicly available code whenever possible for each of our cryptographic primitives and ported it to the TinyOS environment as needed. The performance of our TinyOS ports of RSA and SHA-1 are presented in Table 3. In the table, Time refers to the elapsed time to complete one primitive operation. The primitive operation for RSA-1024 is a modular exponentiation with $e = 3$ and for SHA-1 is computing a 160-bit hash with 64-byte blocks over 64 bytes.

**Table 3: Performance of Cryptographic Primitives.**

| Primitive | Time | RAM | ROM |
|---|---|---|---|
| RSA-1024 | 0.7 s | 529 bytes | 1.2 KB |
| SHA-1 | 0.014 s | 95 bytes | 2.3 KB |

For RSA, we implemented the algorithm in C code as described in [13], 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 [13], 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 seconds to 0.7 seconds, RAM usage decreased from 755 bytes to 529 bytes, and ROM usage decreased from 2.7 KB to to 1.2 KB.

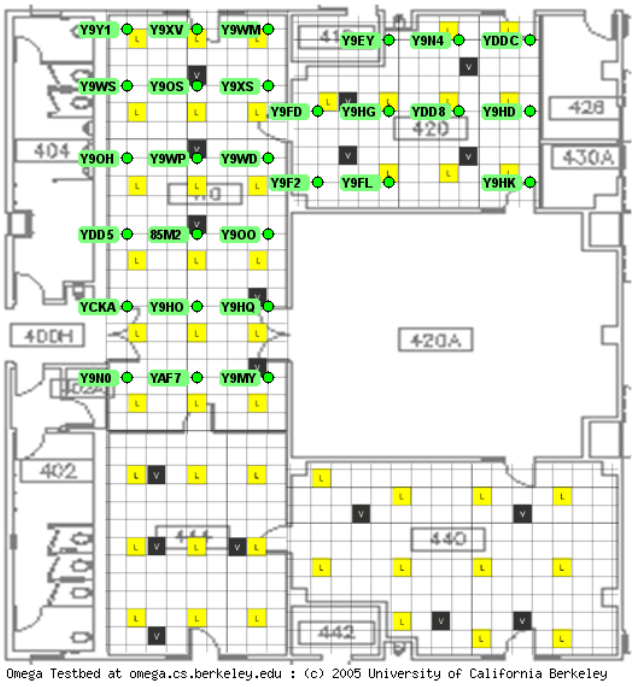


**Figure 2: The Omega Testbed at U.C. Berkeley.**

For SHA-1, we ported publicly available code written 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 approximately 14 ms (median = 13.4 ms, mean = 13.9 ms, standard deviation = 0.86 ms) to hash a stream of 64 bytes using 64-byte blocks. Once again, no optimizations were made for the MSP430, so we believe performance could be improved if optimizations targeted to the MSP430 were implemented. However, the current performance is generally acceptable even though the implementation is not optimized for TinyOS's run-to-completion task model: calling the hash function results in 14 ms of unyielding computation, constraining the realizable radio throughput to 76 packets per second.

## 5.3 Reprogramming Macrobenchmarks

A 21 KB program binary was introduced at the node labeled YDDC (upper right corner) and disseminated using three different Deluge versions: (i) Deluge-23 (standard Deluge 2.0), (ii) Deluge-64 (standard Deluge with 64-byte payload, and (iii) Secure Deluge-56+8 (the secure version of

Deluge with a 56 byte payload and 8 byte hash).

We recorded the times at which each node received the advertisement, finished receiving each page, and completed programming. Each experiment was repeated three times. The results are summarized in Table 4 and shown in Figure 3. Recall that Deluge-23 uses 48 packets per page whereas Deluge-64 and Secure Deluge-56+8 both use 24 packets per page.

**Table 4: Page count and completion times (minimum, median, and maximum) for Deluge variants.**

| Variant | Pages | Min | Median | Max |
|---|---|---|---|---|
| Deluge-23 | 21 | 133 s | 140 s | 154 s |
| Deluge-64 | 14 | 83 s | 86 s | 98 s |
| Secure Deluge-56+8 | 16 | 98 s | 102 s | 114 s |

The median completion time for Deluge-23 is 140 seconds, Deluge-64 is 86 seconds and Secure Deluge-56+8 is 102 seconds. Under ideal conditions, we would expect Secure Deluge-56+8 to require about 15% more messages and time to complete programming than Deluge-64 due to a 15% smaller payload size. Indeed a 15% message overhead (i.e. number of pages) is apparent.

In contrast, however, the data indicate the time to completion for Secure Deluge-56+8 is close to 19%. Figure 3(a) offers one possible explanation for this delay. This figure shows packet reception and verification times for a single page. What we observe is that packets one through five arrive sequentially and are verified in order. Then, packet six is lost. The following packet numbers and reception times (in seconds) were recorded in arrival order:

```
1, 0.000000000000
2, 0.018035888671
3, 0.035888671875
4, 0.053771972656
5, 0.072204589843
...
7, 0.091674804687
```

Note that inter-packet arrival times for packets one through five are approximately 18 ms but the time between packets five and seven is approximately 19.4 ms – much less than the expected value of approximately 36 ms. Since a hash verification takes about 14 ms, it appears as if packet 6 arrived before packet 5 was verified, so it was dropped. However, since we did not actually receive packet 6, we can only speculate. In contrast, packets 17 and 22 appear to have been legitimately lost. As a result of these packet drops, the page reception time increases from an ideal of about 0.4 s to about 0.9 s – an overhead exceeding 200%. However, the median completion times exhibit an overhead of only 19%. These results are a bit surprising but they can be explained by channel contention at scale, which impacts completion time far more than verification delays.

## 5.4 Completeness of the Design

We show our working system satisfies the security and efficiency requirements outlined in Section 3. Program source *authenticity* is satisfied by the RSA digital signature. The *integrity* requirement is met by the resistance of SHA-1 to pre-image attacks. Collision attacks against SHA-1 have been reported [40], but such attacks are ruled out by our

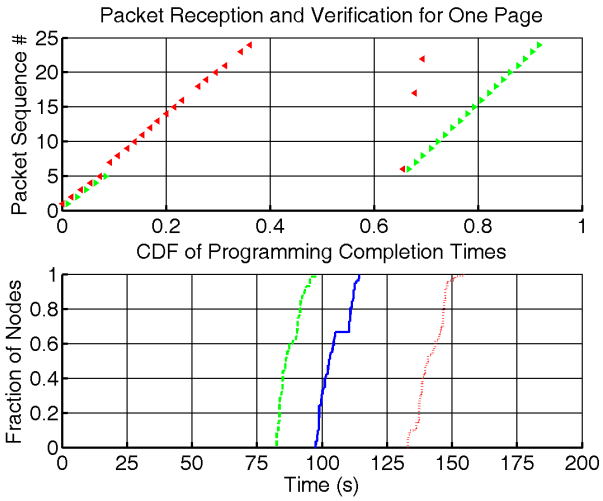


**Figure 3: Empirical Results: (a) Packet reception (◄) and verification (►) times for one page of data (Note: verification times have been delayed by 10 ms for clarity); (b) CDF of programming completion time for 28 nodes using Deluge-23 (dotted red line), Deluge-64 (dashed green line) and Secure Deluge-56+8 (solid blue line).**

threat model. *Freshness* is ensured by using monotonically increasing and digitally signed version numbers. *Compromise tolerance* is achieved by not storing any secret keys on a node. The system is *delay tolerant* because there is no dependence on time synchronization.

Support for *incremental verification* is achieved because under our hash chain construction, each hash is a commitment for the next message, permitting incremental authentication of packets. In the pessimistic case of extreme packet loss, the system can still make progress using Deluge's support for selective negative acknowledgments. Finally, *no dynamic server state is required* since a server only stores its public key pair, which is static. Other information, such as the program version number, is retrieved from the network.

# 6. CONCLUSION

We present and evaluate a system for *secure* and *efficient* network programming in resource-constrained wireless sensor networks. This work demonstrates the feasibility and low overhead of adding public-key-based program source authentication, strong integrity verification, and freshness checks to existing network programming services. We tested our implementation on a real testbed consisting of 28 Telos motes. Our secure network programming protocol can be readily generalized to solve the problem of *secure, reliable, bulk data dissemination* in sensor networks.

# 7. ACKNOWLEDGMENTS

We thank Chris Karlof, David Molnar, Naveen Sastry, Jay Taneja, David Wagner, and the anonymous reviewers for their invaluable feedback. This work was supported by the NSF under grant# 0435454 ("NeTS-NOSS"), the NSF Graduate Research Fellowship Program, and DARPA grant F33615-01-C1895 ("NEST").

# 8. REFERENCES


[1] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, 1996.

[2] F. Bahr, M. Boehm, J. Franke, and T. Kleinjung. 640-bit RSA modulus factored. NMBRTHRY@LISTSERV.NODAK.EDU, Nov 2005.

[3] 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.

[4] S. Coleri, S. Y. Cheung, and P. Varaiya. Sensor networks for monitoring traffic. In *Forty-Second Annual Allerton Conference on Communication, Control, and Computing*, Univ. of Illinois, Sept. 2004.

[5] J. Deng, R. Han, and S. Mishra. Secure code distribution in dynamically programmable wireless sensor networks. In *Proceedings of the Fifth International Conference on Information Processing in Sensor Networks (IPSN'06)*, Apr 2006.

[6] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):74–84, 1976.

[7] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN'04)*, pages 455–462, Nov. 2004.

[8] P. Dutta, M. Grimmer, A. Arora, S. Bibyk, and D. 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.

[9] P. K. Dutta and D. E. Culler. System software techniques for low-power operation in wireless sensor networks. *ICCAD*, 2005.

[10] J. Gehrke and S. Madden. Query processing in sensor networks. *Pervasive Computing*, Jan. 2004.

[11] R. Gennaro and P. Rohatgi. How to sign digital streams. *Lecture Notes in Computer Science*, 1294:180+, 1997.

[12] V. Gupta, M. Millard, S. Fung, Y. Zhu, N. Gura, H. Eberle, and S. C. Shantz. Sizzle: A standards-based end-to-end security architecture for the embedded internet. In *Third IEEE Conference on Pervasive Computing and Communications*, pages 247–256, 2005.

[13] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shants. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In *Workshop on Cryptographic Hardware and Embedded Systems*, 2004.

[14] S. Halevi and H. Krawczyk. Strengthening digital signatures via randomized hashing, May 2005.

[15] T. He, S. Krishnamurthy, J. A. Stankovic, T. Abdelzaher, L. Luo, R. Stoleru, T. Yan, L. Gu, G. Zhou, J. Hui, and B. Krogh. Vigilnet:an integrated sensor network system for energy-efficient surveillance. *In submission to ACM Transaction on Sensor Networks*, 2004.

[16] J. Hill, M. Horton, R. Kling, and L. Krishnamurthy. The platforms enabling wireless sensor networks. *Communications of the ACM*, 47(6):41–46, jun 2004.

[17] J. W. Hui and D. 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*, pages 81–94, 2004.

[18] P. Juang, H. Oki, Y. Wang, M. Martonosi, L.-S. Peh, and D. 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.

[19] C. Karlof, N. Sastry, and D. Wagner. Tinysec: A link layer security architecture for wireless sensor networks. In *Second ACM Conference on Embedded Networked Sensor Systems (SensSys 2004)*, November 2004.

[20] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. *Lecture Notes in Computer Science*, 1666:388–397, 1999.

[21] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. *Lecture Notes in Computer Science*, 1109:104–113, 1996.

[22] S. S. Kulkarni and M. Arumugam. INFUSE: A TDMA based data dissemination protocol for sensor networks. Technical report, Michigan State Univ., East Lansing, MI, USA, 2004.

[23] S. S. Kulkarni and L. Wang. MNP:multihop network reprogramming service for sensor networks. In *International Conference on Distributed Computing Systems (ICDCS'05)*, Jun 2005.

[24] P. E. Lanigan, R. Gandhi, and P. Narasimhan. Sluice: Secure dissemination of code updates in sensor networks. In *The 26th International Conference on Distributed Computing Systems (ICDCS '06)*, July 2006.

[25] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, 2002.

[26] D. Malan, M. Welsh, and M. 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.

[27] J. Paradiso, J. Lifton, and M. Broxton. Sensate media - multimodal electronic skins as dense sensor networks. *BT Technology Journal*, 22(4):32–44, Oct. 2004.

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

[29] A. Perrig, R. Canetti, J. Tygar, and D. X. Song. Efficient authentication and signing of multicast streams over lossy channels. In *IEEE Symposium on Security and Privacy*, pages 56–73, May 2000.

[30] A. Perrig, R. Szewczyk, V. Wen, D. 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.

[31] A. Perrig and D. Tygar. *Secure Broadcast Communication: In Wired and Wireless Networks*. Kluwer Academic, 2002.

[32] J. Polastre, R. Szewczyk, and D. 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.

[33] R. Rivest, A. Shamir, and L. Adelman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[34] C. Sharp, S. Schaffert, A. Woo, N. Sastry, C. Karlof, S. Sastry, and D. 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.

[35] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical report, UCLA, Los Angeles, CA, USA, 2003.

[36] R. Szewczyk, A. Mainwaring, J. Polastre, and D. 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.

[37] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. Lessons from a sensor network expedition. In *Proceedings of the First European Workshop on Sensor Networks (EWSN)*, Jan. 2004.

[38] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *2nd European Workshop on Wireless Sensor Networks*, Jan. 2005.

[39] University of California, Berkeley. Tinyos. `http://www.tinyos.net/`, 2004.

[40] X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full SHA-1. In *CRYPTO*, pages 17–36, 2005.

[41] R. Watro, D. Kong, S. fen Cuti, C. Gardiner, C. Lynn, and P. 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*, pages 59–64, 2004.

[42] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, and M. Welsh. Monitoring volcanic eruptions with a wireless sensor network. In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN'05)*, Jan. 2005.

[43] N. Xu, S. Rangwala, K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A wireless sensor network for structural monitoring. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, Nov. 2004.