

# Sluice: Secure Dissemination of Code Updates in Sensor Networks

Patrick E. Lanigan  
Information Networking Institute  
Carnegie Mellon University  
Pittsburgh, PA 15213  
lanigan@cmu.edu

Rajeev Gandhi  
ECE Department  
Carnegie Mellon University  
Pittsburgh, PA 15213  
rgandhi@andrew.cmu.edu

Priya Narasimhan  
ECE Department  
Carnegie Mellon University  
Pittsburgh, PA 15213  
priya@cs.cmu.edu

## Abstract

*Existing network reprogramming protocols target the efficient, reliable, multi-hop dissemination of application updates in sensor networks, but assume correct or fail-stop behavior from participating sensors. Compromised nodes can subvert such protocols to result in the propagation and remote installation of malicious code. Sluice aims for the progressive, resource-sensitive verification of updates in sensor networks to ensure that malicious updates are not disseminated or installed, while trusted updates continue to be efficiently disseminated. Our verification mechanism provides authenticity and integrity through a hash-chain construction that amortizes the cost of a single digital signature over an entire update. We integrate Sluice with an existing network reprogramming protocol and empirically evaluate its effectiveness both in a real sensor testbed and through simulation.*

## 1 Motivation

Wireless sensor networks are typically long-lived, large scale and often deeply embedded into existing infrastructures. In many cases, it might be necessary to update the sensor application to fix bugs or to add new functionality. Loading a new application or upgrading an existing application on a sensor-node requires a physical connection to the node, typically via a serial port or some other hardwired back-channel. Thus, one way to update a network of sensors would be to reprogram them *in situ*, one at a time.

However, physical access to nodes might be limited once the sensor network is deployed. In low-impact environmental monitoring applications, dispatching maintenance personnel to collect nodes or re-flash them physically might adversely affect the sensor application or the environment being monitored. In some cases, physical access might even be impossible, e.g., monitoring sensors implanted into concrete bridges or asphalt roadways. Even if physical access

were possible, manually updating hundreds or thousands of nodes would be a tedious task indeed.

*Network reprogramming* protocols have recently emerged as a way to distribute program updates remotely without requiring manual intervention or physical access to sensor-nodes. Update dissemination typically occurs over the sensor's primary communication channel (i.e., the radio), hence the term, "over-the-air" updates. An entire network of sensors can be updated *en masse* because these multi-hop protocols are mostly epidemic in nature, i.e., sensor-nodes propagate any received updates to their in-range neighbors, which then propagate them further downstream, and so on.

A number of network reprogramming protocols (Deluge [8], MNP [10], MOAP [20], etc.) have been developed to facilitate over-the-air software updates in the context of resource-constrained sensors nodes. For the most part, these protocols focus on providing reliable data-dissemination with low resource consumption and low end-to-end update latency. These protocols generally assume well-behaved (i.e., non-malicious) sensors, and are primarily tolerant to fail-stop nodes and packet loss through mechanisms such as negative acknowledgments and retransmissions.

Because current network reprogramming protocols do not restrict the origin of an update, an adversary who is merely able to inject packets into the network can inject a malicious update into the system. Given the epidemic nature of network reprogramming protocols, an adversary can gain complete control over the entire sensor network by compromising just a *single* node, and then subsequently leveraging the network reprogramming mechanism to distribute/install the malicious code on *every* reachable node in the system. In addition, most of these protocols aim to speed the update propagation through a number of efficiency mechanisms. While a correct node can use these mechanisms to propagate updates quickly through the network, so too can a malicious node. Without a mechanism to halt the propagation of malicious code, the benefits of network reprogramming protocols can be voided because these

protocols can effectively, and often efficiently, allow a lone malicious node to compromise the entire system.

Sluice is an extension to network reprogramming protocols that facilitates the secure dissemination of trusted program updates. The contributions of this paper are:

- **Propagation of trusted updates.** Sluice ensures that only trusted updates are disseminated in sensor networks, while malicious updates are halted from further propagation. This ensures that the compromise of any sensor-node will not lead to the execution of malicious code on any other node.
- **Resource-sensitive, progressive verification.** Sluice respects the resource constraints of sensor-nodes and upholds the use of existing efficiency mechanisms. Thus its verification is both resource-sensitive and progressive. This is enabled through the strategic and synergistic use of a single digital signature over an entire update, along with a hash-chain over sequential pieces (pages) of the update.
- **Experimental evaluation.** Sluice can secure existing network reprogramming protocols without significantly altering the underlying protocol’s operational semantics, scalability, or performance. We demonstrate this by evaluating Sluice’s performance (i) on a test-bed of Telos sensor-nodes running TinyOS [13], and (ii) through sample topologies simulated in TOSSIM [12], the TinyOS simulation environment.

The paper is organized as follows. Section 2 provides background information on various network reprogramming protocols. Section 3 describes the problem that we aim to solve, along with our simplifying assumptions, our threat model, and the required and desired properties of Sluice. Section 4 discusses some of the approaches that we considered, what we learnt from them, and why we decided *not* to adopt them. Section 5 provides the details of our approach, while Section 6 describes our current implementation and evaluation. We discuss related work in Section 7, future work in Section 8, and conclude in Section 9.

## 2 Background

A number of protocols have recently emerged to support network reprogramming for sensor-nodes running the TinyOS operating environment [13]. We use the term “update” to refer to the new application image that needs to be installed on the node. All of these protocols assume well-defined behavior on the part of every node as the update is disseminated throughout the system.

Deluge [8], currently distributed as a part of the TinyOS [13] environment, is the most widely available multi-hop

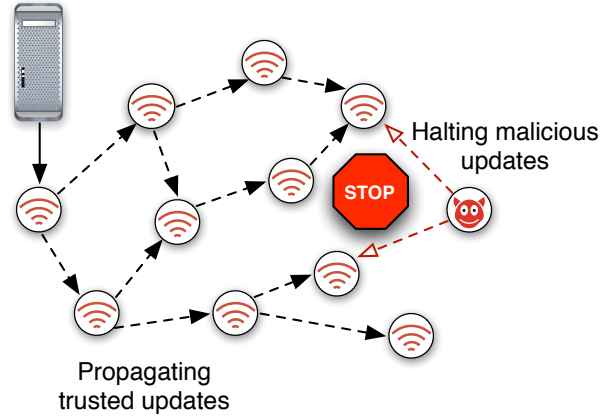


Figure 1. Sluice in operation

network reprogramming protocol. With Deluge, an update is broken up into fixed-size fragments called *pages*. Each page is further broken up into fixed-size *packets* that serve as transmission units. Deluge employs a three-phase (advertise-request-data) process to propagate pages between nodes. Dissemination takes place in a pipelined fashion, whereby nodes are allowed to forward pages that they have already completed without waiting to receive all of the pages that form the entire update. Pages must be received *sequentially*, i.e., a node may not begin forwarding page  $n$  until it has first received all pages numbered  $0, 1, \dots, n - 1$ , inclusive.

Two other network reprogramming protocols are MOAP and MNP. MNP [10] employs a sender-selection mechanism to limit the number of nodes that can transmit data in a particular broadcast neighborhood. Eligible source nodes compete with each other to become senders; nodes that “lose” are placed in a sleep state for a predetermined period of time. MNP also exploits the use of pages and pipelining for efficiency. MOAP [20] uses a publish-subscribe mechanism to disseminate updates on a multi-hop “neighborhood-by-neighborhood” basis. MOAP aims for simplicity and does not use pipelining, instead requiring nodes to receive the entire update before subsequent propagation.

We have leveraged Deluge for our reference implementation because of its wide availability. However, our approach, as described in Section 5, is independent of Deluge, and can be used to extend *any* network reprogramming protocol, subject to the assumptions listed in Section 3.

## 3 Problem Statement

The primary focus of Sluice is the *progressive, resource-sensitive verification of updates in sensor networks so that malicious updates are not propagated or installed, while trusted updates can continue to be efficiently disseminated.*

We illustrate the operation of Sluice in Figure 1.

Many of the existing network reprogramming protocols use efficiency mechanisms such as pipelining to reduce the amount of time taken by the update to reach all of the nodes in the system. In keeping with existing network reprogramming protocols, we make no assumptions regarding time synchronization, i.e., clocks on the different sensor-nodes are not synchronized and message propagation can take an unbounded amount of time. We do assume a monolithic application environment, where an update needs to be available in its entirety for installation. Any underlying network reprogramming protocol must provide reliable, in-order delivery of pages. We assume that each sensor-node, while resource-constrained, does have sufficient memory to hold the security mechanisms that Sluice adds.

### 3.1 Threat Model

We assume that individual sensor-nodes are untrusted, and might exhibit arbitrary behavior. Untrusted, but well-behaved, nodes can become compromised, yielding complete control of their functionality and/or cryptographic material. We also assume an insecure wireless medium.

Our design covers different kinds of adversarial behavior, e.g., an adversary can inject an arbitrary number of corrupt pages or updates into the system, can withhold/delay an arbitrary number of pages or updates from transmission, can modify an arbitrary number of pages at will, and can eavesdrop on the communication of other sensors in the system. We make no assumptions about the number of malicious nodes, their locations, the degree of connectivity or collusion between them. We do not address the confidentiality of updates – our focus is instead on *authenticating the trusted source of the update* and *establishing the integrity of the update*. We do not yet address denial-of-service or battery-drain attacks in our current version of Sluice. Replay attacks, however, could be handled through a relatively minor extension to Sluice (but is outside of our current scope).

We assume that there exists a trusted source (e.g., a base-station), that holds a public/private key pair, and that is the source of authorized updates. The private key is known only to the base, and the base is hardened against compromise. The public key is globally known to all untrusted sensor-nodes. Typically, the base station will be a physically isolated desktop-class computer.

### 3.2 Properties

**Required Properties.** Sluice *must* satisfy the following properties in order to be considered correct:

- **Authenticity.** Updates must be verified as *originating* from a trusted source.

- **Integrity.** Updates must be verified as arriving *unmodified* from the trusted source.
- **Correctness.** No node should ever install, or disseminate, pages of an unverified update, unless the node has been physically compromised.

These properties should hold *regardless of the number of compromised, untrusted nodes*. For conciseness, the term *verification* will be used to encompass the notions of *authenticity* and *integrity* together.

**Desired Properties.** Sluice *should* satisfy the following properties in order to be considered feasible:

- **Completeness.** Every well-behaved node should eventually install a verified/trusted program update. This is a departure from previous work [20, 8, 10] where eventual propagation is a required property, but malicious nodes that may impede progress are not considered.
- **Progressive verification.** Verification should be done in an *ongoing* fashion, so that good data is propagated while malicious data is halted. It should not be necessary to wait until an entire update has been received to verify it.
- **Compatibility.** The verification mechanism should make minimal assumptions about the underlying dissemination mechanism, so as to be compatible with as wide a range of existing protocols as possible.
- **Efficiency.** Nodes should be able to take advantage of existing efficiency mechanisms (i.e. pipelining) to propagate trusted updates as quickly as possible.
- **Resource-sensitivity.** Finally, the security mechanisms should be sensitive to the resource constraints in sensor networks, minimizing processing and transmission overhead relative to existing protocols.

## 4 Alternative Approaches

We explored various relevant and potentially applicable security mechanisms. Ultimately, we discarded some of these mechanisms in favor of our current approach. For the sake of completeness and to provide motivation for Sluice’s mechanism, we discuss these possible approaches here, and our reasons for not adopting some of them.

**Remote verification.** Software attestation techniques, such as SWATT [19], aim for the external verification of the memory contents of embedded devices to establish the absence of malice. The attestation of the embedded device’s memory contents happens through a trusted verification entity that is connected to, but physically different from, the device under verification. Because tampering can only be detected after the fact, SWATT does not prevent attacks. By the time an attack is detected, precious energy might have been wasted propagating, installing and executing a malicious update.

**Symmetric-key cryptography.** Symmetric-key protocols like TinySec [9] provide confidentiality, message integrity, and access control. These schemes protect against eavesdropping, message tampering, and message injection. However, the use of these protocols alone does not provide sufficient assurance against compromised nodes.

Symmetric link-layer security protocols against packet injection by arbitrary nodes by using a message authentication code (MAC) on each packet to authenticate traffic as coming from a legitimate source. If a node is physically compromised, its cryptographic material is potentially compromised as well; thus, an adversary can use the compromised node to inject packets into the network that pass the MAC, but that are nevertheless malicious. In a network reprogramming setting, these seemingly legitimate nodes can propagate malicious updates. Given the epidemic nature of network reprogramming, a small number of compromised nodes can infect the entire sensor network.

Clearly, any secure network reprogramming protocol must be robust against node compromise. Pairwise key schemes [15, 21] attempt to mitigate the threat of compromised nodes by establishing shared keys that are held by only a small subset of network nodes. Unfortunately, a compromised node might propagate updates to nodes that it is paired with, which, in turn, might forward the updates to nodes they are paired with, and so on. As long as the network is connected, the malicious update can eventually reach all nodes.

**Authenticated broadcast protocols.** Authenticated broadcast protocols, such as  $\mu$ TESLA [18], typically use symmetric cryptographic primitives with delayed key-disclosure to authenticate the claimed sender of a broadcast message. Disclosed keys are verified against an initial keychain commitment, through the iterative application of a pseudo-random function. The keychain commitment must be securely distributed to all receivers during a bootstrapping phase.  $\mu$ TESLA unicasts commitments to each receiver using pre-distributed symmetric keys. This limits  $\mu$ TESLA’s scalability in networks with a large number of receivers. Moreover, the security of

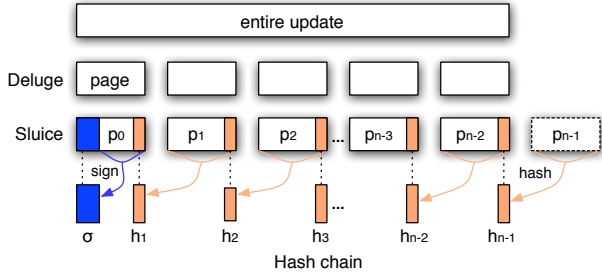
authenticated protocols typically depends on loose time synchronization (i.e., receiver has an upper bound on the sender’s local time) across the network. These inherent timing assumptions further limit applicability to Sluice because network programming protocols do not place any temporal bounds on the nodes or the update itself, and instead aim for the *eventual* completion of the end-to-end update.

**One-way hash chains.** Hash chains were first proposed by Lamport [11] as a mechanism for one-time passwords. Hash chains are based upon a public function  $H$  that is easy to compute, but computationally difficult to invert – thus, the term, one-way hash chains. A hash chain of length  $m$  is generated by repeatedly applying the hash function  $H$  to the last element, say,  $h_m$ , in the chain, to generate a sequence of hashes such that  $h_j = H(h_{j+1})$ . The head of the chain,  $h_0$ , serves as a commitment to the entire hash chain. The hash chain is generated in the order  $h_m, h_{m-1}, \dots, h_1, h_0$  and revealed in the reverse order. It is assumed that the verifying entity has access to the trusted value  $h_0$ . Armed with  $h_0$ , the verifier can verify any other element in the hash chain, say,  $h_i$ , simply by comparing  $h_0$  with  $H^i(h_i)$ .

We leverage the concept of one-way hash chains in Sluice. Note there needs to be a way for Sluice to ensure that the head of the chain is a trusted value that can then be used legitimately as a commitment to the remainder of the chain.

**Public-key cryptography.** Public-key schemes, e.g., digital signatures, depend on asymmetric cryptography, and have been long thought to be impractical for the limited computational, memory and energy resources in sensor networks. However, recent work [6, 7, 14, 16, 22] suggests that, with careful implementation and judicious use, public-key cryptography can be quite feasible on sensor-nodes.

Watro et al. have implemented on TinyOS primitives needed for the RSA cryptosystem [22]. Gura et al. showed that elliptic curve cryptography (ECC) can provide substantial performance gains over RSA on constrained platforms [7]. EccM [16] provides a Diffie-Helman (DH) key-distribution mechanism based on elliptic curves, but does not address digital signatures. Sizzle [6] implements SSL using ECC on the MICA2 and Telos platforms. The highly optimized code can complete a full SSL handshake (which includes ECDSA and ECDH operations) in under 4 seconds. While this work demonstrates the viability of ECC in highly resource-constrained systems, a full SSL handshake is neither necessary nor appropriate to secure network reprogramming protocols. SSL provides one-to-one semantics, while network reprogramming is by nature one-to-many. TinyECC [14] provides an ECC implementation for TinyOS that includes an ECDSA module. This module



**Figure 2. Sluice’s hash chain construction.**

is optimized for the MICAz platform, and is able to verify a digital signature in under 15 seconds. While the Sizzle code is reportedly significantly faster, it is not yet publicly available. Since TinyECC is freely available, we use it to implement the security mechanisms in Sluice.

## 5 Sluice’s Approach

In motivating the progressive-verification approach that we employ in Sluice, we first discuss the trade-offs in the various possible ways that we might leverage digital signatures.

One alternative to provide for the authenticity and integrity of an update would be to compute a single digital signature over the entire update without incorporating any page-level or packet-level verification. This approach would require the update to be received in its entirety before it can be verified at all. Because updates are typically too large to fit into the available RAM on a sensor, pages would need to be stored temporarily in flash memory (stable storage), as they arrive, and then retrieved from flash at verification time. Reading from, and writing to, flash are energy-intensive operations, and should only be executed after data has been verified. This approach is also incompatible with pipelining. If nodes are allowed to forward unverified data, a single malicious node could potentially flood a multi-hop network with malicious updates. Therefore, pipelining would need to be deactivated.

Yet another alternative would be to compute a digital signature over each page of an update. This would allow pages to be buffered temporarily in RAM before they are verified, and subsequently written to flash memory only after verification. This would additionally allow verified pages to be transferred in a pipelined fashion. However, signing every single page of an update, while effective in its security goals, might still be too expensive in resource-constrained sensor networks.

### 5.1 Sluice’s Progressive Verification

We follow an approach that exploits a synergistic combination of digital signatures and off-line hash chains to allow

for the progressive verification of individual pages, while requiring only a single digital signature across the entire update. The basic idea is to create a chain of hashes, by computing a hash of each page and incorporating that hash as a part of the previous page’s payload. We then digitally sign the first page in the chain; thus, there is hash for each page, but only one digital signature for every set of pages that form an update. This concept has previously been exploited to sign digital streams [5]; we exploit it here for Sluice’s progressive verification of streamed updates in the context of network reprogramming.

An existing network reprogramming protocol, such as Deluge, decomposes an entire update into a set of pages. Figure 2 shows the sequence of  $n$  pages that results with Sluice, where each page’s hash is computed and appended to the previous page’s payload, e.g., the hash  $h_{n-1}$  of page  $p_{n-1}$  is computed and appended to the payload of the previous page, thereby forming page  $p_{n-2}$ . The last page in the sequence does not contain a hash, and the first page,  $p_0$  contains the hash,  $h_1$ , of page  $p_1$ , as well as a digital signature,  $\sigma$ , of the combined sequence of bytes represented by the payload of  $p_0$  concatenated with  $h_1$ . The one-way hash chain is formed by the sequence of hashes  $h_1, h_2, h_3, \dots, h_{n-1}$ .

The head,  $h_1$ , of the hash-chain is a part of the page that has been digitally signed by  $\sigma$ , thereby providing Sluice with a way to verify whether it is a trusted value. Thus, the digital signature serves to authenticate the trusted source, and verify the payload in the first page and the hash contained in that page. Once the head of the hash chain,  $h_1$ , has been established to be a trusted value, the hashes embedded in the remaining pages can be verified through the one-way hash-chain properties. Hash  $h_{i+1}$  contained in page  $p_i$  serves to verify the payload and the hash contained in page  $p_{i+1}$ . This provides Sluice with a way to verify the integrity of all of the pages that form the update.

Because one of our goals in Section 3.2 was compatibility, we do not modify the fixed page sizes that the protocol uses. There is likely to be some spatial overhead with Sluice due to the hashes and the digital signature. This spatial overhead does not change the size of each page, or the size of the update that needs to be installed, but might change the number of pages that need to be transmitted on behalf of an update.

Our approach can be divided into functionality that takes place at the trusted source, and functionality that takes place at each sensor-node in the network. The trusted source is responsible for processing the update prior to dissemination by partitioning the update into equal-sized pages, and then computing the hash-chain and digital signature (see Figure 3). Sensor-nodes that receive pages of the update are responsible for verifying each page as it is received. Once a page has been verified as authentic using the digital sig-

```

split update into  $n$  pages  $p_0, p_1, \dots, p_{n-1}$ 
for all  $i = n - 1$  to  $i = 1$  do
     $h_i \leftarrow \text{HASH}(p_i)$ 
     $p_{i-1} \leftarrow (p_{i-1} | h_i)$ 
end for
 $\sigma \leftarrow \text{SIGN}(p_0)$ 
 $p_0 \leftarrow (\sigma | p_0)$ 

```

**Figure 3. Processing at the trusted source.**

```

static  $h, i = 0$ 
while  $i < n$  do
    complete receiving page  $p_i$ 
    if ( $i = 0$ ) then
         $\sigma, m \leftarrow p_0$ 
        if  $\text{VERIFY}(\sigma, m)$  then
             $\text{ACCEPT}(p_0)$ 
        end if
    else
         $h' \leftarrow \text{HASH}(p_i)$ 
        if ( $h = h'$ ) then
             $\text{ACCEPT}(p_i)$ 
        end if
    end if
end while
verified update available for installation

procedure  $\text{ACCEPT}(p)$ 
    make  $p$  available for forwarding
    save  $p$  in stable storage
    if ( $i < n - 1$ ) then
         $p, h \leftarrow p$ 
         $i \leftarrow i + 1$ 
    end if
end procedure

```

**Figure 4. Processing at each receiving node.**

nature or hash chain (as appropriate), the page is then written to flash memory and marked as available for subsequent propagation to other nodes (see Figure 4).

## 6 Implementation and Evaluation

An attacker could be any entity with over-the-air capabilities, and so could have computational capabilities exceeding those of the sensor-nodes. Therefore, we use industry standard cryptographic primitives and security parameters [1, 3, 17]. TinyECC [14] provides ECDSA verification functionality on the sensor-nodes, while the BouncyCastle JCE provider [2] is used for key and signature generation at the trusted source. We construct the hash chain using full 160-bit SHA1 digests [17]. We emphasize that these cryptographic libraries are off-the-shelf and *not* optimized for our platform.

We implemented Sluice as an extension to Deluge [8], which defines 1104-byte pages by default. To avoid changing Deluge’s default page size, we reserve 20 bytes of each page to encapsulate the hash and an additional 44 bytes in the first page to encapsulate the signature (see Figure 2). By embedding the cryptographic data with the update payload, we also avoid the need to change the packet size of the underlying transport layer.

To provide the trusted source’s functionality, we modified the existing Deluge Java toolchain. Our changes were confined to the `net.tinyos.deluge.DelugeImage` class. The original class pads the update to a whole number of pages, calculates a CRC on each page, and stores the CRCs in the first page. We modified this class to calculate the digital signature and one-way hash-chain as well. First, we determine how many pages will be required to hold the update after adding signature and hash bytes. We then split the update into pages, reserving additional space in the first page for a signature and a hash, and space in each subsequent page for a hash alone. If necessary, we pad the final page with zero bytes. The hash-chain is constructed as described in Section 5, starting at the last page and moving forward to the first page. We then compute the signature over the combined application and hash data contained in the first page, and place it immediately following the CRC block.

We modified the Deluge nesC library to provide sensor-node functionality. These modifications were limited to the `DelugePageTransfer` module, which is responsible for managing the transfer of pages between nodes. The original module buffers a few packets at a time before writing them to flash memory. We buffer an entire page in RAM and write the data to flash only after it has been verified. Once the `DelugePageTransfer` module has determined that a complete page has been buffered, it parses the buffered data and passes it to the TinyECC or SHA1 module for verification. If the verification is successful, the hash value contained in the page is stored in RAM for future use, and the entire verified page buffer is written to flash memory. If verification fails, the page is reset and cleared, and a new page-transfer request is initiated.

### 6.1 Discussion of Sluice’s Properties

*Authenticity* and *integrity* are predicated on the unforgeability of the ECDSA digital signature algorithm, and the pre-image resistance of SHA-1. In order to inject arbitrary data, an adversary would have to forge the digital signature of page  $p_0$  or find a pre-image of some hash  $h_i$ . The strength of our cryptographic primitives is supported through our use of standard cryptographic libraries [2, 14], security parameters [3] and modes of operation [5]. Because pages are buffered temporarily in RAM and are only written to flash

memory once they are verified, only trusted updates are installed, thereby providing *correctness*.

*Progressive verification* is made possible through the page-wise hash-chain construction along with the update-wise digital signature. Our *efficiency* goal is met by our ensuring that the underlying network reprogramming protocol's efficiency mechanisms, e.g., pipelining, are not affected by Sluice – in fact, the need to maintain these efficiency mechanisms was factored into our design. Our *compatibility* goals are met by our ensuring that our approach is independent of Deluge, and can be used to secure *any* network reprogramming protocol.

Our design meets *resource-sensitivity* goals by amortizing the cost of a hash over an entire page, and the cost of a digital signature over an entire update. The overheads incurred by the implementation of Sluice can be reduced by using a more efficient implementation of cryptographic primitives.

## 6.2 Metrics of Interest

We benchmark Sluice against its current foundation, Deluge, whose empirical evaluation is reported elsewhere [8]. Unless otherwise noted, overhead refers to the additional cost of Sluice's security mechanisms over the default underlying network reprogramming protocol.

- **Spatial overhead** is comprised of (i) memory overhead (in ROM and RAM) required to support secure network reprogramming, and (ii) transmission overhead, i.e., the number of additional bytes transmitted in beyond the actual update payload.
- **Temporal overhead** is measured with respect to end-to-end dissemination latency. The end-to-end latency is defined as the period of time between the initial availability of a new update (i.e., when the first advertisement is sent) and the verification of the final page of that update at the last sensor.
- **Survivability** is the ability of the protocol to sustain operation in the presence of malicious nodes.

## 6.3 Experiments

We use a physical sensor testbed to evaluate the spatial overhead of Sluice, and to demonstrate the implementability of our approach. We also gathered basic testbed data to quantify the temporal overhead with respect to an increasing number of pages in an update. Simulation results allowed for further evaluation with regard to temporal overhead as the size of the network scales.

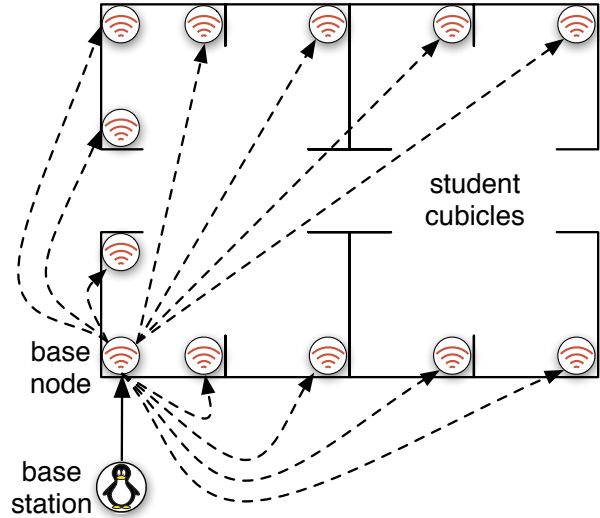


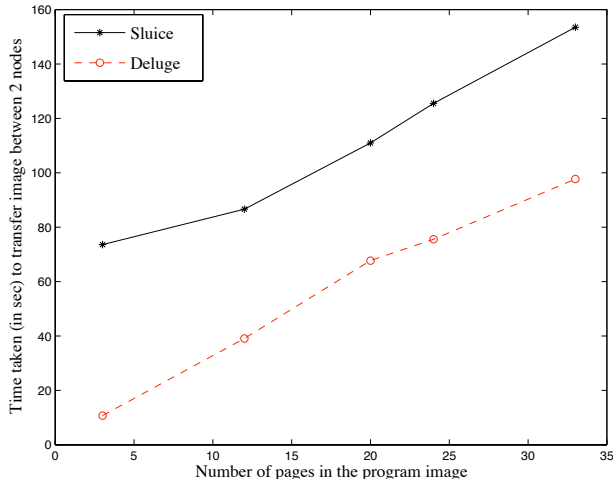
Figure 5. Layout of testbed nodes.

### 6.3.1 Testbed

Our current testbed (Figure 5) is comprised of 12 Tmote Sky nodes in an office setting. The Tmote Sky nodes have a 16-bit MSP430 microcontroller, 48KB of ROM and 10KB of RAM. The nodes communicate using 802.15.4 compatible radios transmitting at up to 250Kbit/s. A USB link to each node provides a back-channel for debugging and gathering data. Our trusted source (which we call the base-station) is located on a Pentium 4 desktop running Linux. In each of our testbed experiments, a single node is designated as the base-node. The modified `net.tinyos.tools.Deluge` Java application is used to connect to the base-node from the base-station over the USB back-channel and initiate the update process. The update propagates from the base-station to the base-node; upon verification, each page is further disseminated into the network. Note that the base-node is itself an untrusted node, and has no knowledge of the base-station's private key.

To evaluate the scalability of our extensions with respect to application size, we injected a number of existing TinyOS applications and measured end-to-end latency in each case. Due to the configuration (12 nodes in close proximity) of our testbed, it was difficult to emulate a multi-hop network. Therefore, we only present testbed results for sensors that are within broadcast range of one another.

**Spatial Overhead.** On the Tmote Sky platform, our current implementation of Sluice requires roughly 9kB ROM and 2kB RAM over Deluge. The increased ROM size comes mainly from the unoptimized cryptographic libraries. Most (1104 bytes) of the RAM overhead is incurred due to Sluice's buffering of pages prior to verification.



**Figure 6. Latency with respect to update size.**

Theoretically, Sluice adds roughly  $S_\sigma + n * S_h$  bytes to the size of an update, where  $S_\sigma$  is the size of a digital signature (44 bytes here),  $S_h$  is the size of a hash (20 bytes here), and  $n$  is the number of pages in the update, as generated by Sluice. However, in practice, this overhead can be masked by the fact that Deluge pads pages out to an even number of pages. If there is enough padding to hold the additional bytes due to Sluice, the transmitted update size will not change, and Sluice’s transmission overhead will be zero.

**Temporal Overhead.** Both Deluge and Sluice show generally linear scalability with respect to update size, as seen in Figure 6. For applications with a small number of pages, Sluice has significant overhead in terms of latency (585%). As the update size grows, Sluice’s relative latency decreases nearly ten-fold (to 57% for a 44-page update). Since the transmitted data overhead of Sluice with respect to Deluge is zero in this case (leveraging available padding bytes), we attribute the increased latency to long-running cryptographic operations. Through code instrumentation on our physical testbed, we determined that a single ECDSA digital-signature verification operation takes between 30 and 35 seconds. By comparison, a hash computation takes roughly only 0.2 seconds. Optimizing the digital signature and hash computations will significantly reduce this overhead.

**Survivability.** We have conducted tests to ensure that Sluice’s security mechanisms work, i.e., that a single malicious node is not able to (i) inject an arbitrary unsigned update, or (ii) to modify and subsequently re-inject a previously signed update. The base-node does not proceed with

the dissemination of an update that contains an invalid signature, and halts the dissemination process as soon as Sluice (running on the base-node) detects a mismatch in the the hash-chain. Network nodes exhibit similar behavior when an update is fed into the system by a “malicious” node (simulated by a separate base-node running Deluge to inject an unsigned or modified update). This behavior is representative of how a single broadcast neighborhood would react to an unauthorized update in a multi-hop setting.

### 6.3.2 Simulation

We use TOSSIM [12] to evaluate Sluice’s scalability with regard to the number of hops. TOSSIM is an event-based simulator that models TinyOS communication I/O events at the bit level, but does not model code execution time. To better simulate long running cryptographic operations, we manipulate the TOSSIM event queue to insert an artificial delay between the time that a cryptographic operation is started to the time when the calling module is notified of its completion. Using measurements gathered from our testbed implementation of Sluice, we assign a delay of 35 seconds for ECDSA verification and 0.2 seconds for hash computation. We do not consider malicious nodes in these simulations, and note that the results quantify the overhead of our security primitives alone.

Each topology is specified as a grid of  $N \times N$  nodes spaced at 15 meters, with lossy radio links modeled by specifying a a bit-error rate for each link. Each test simulated the dissemination of a 33-page update<sup>1</sup>. This page size is typical of what might be seen in actual applications. Due to simulation times on the order of hours<sup>2</sup>, we provide only single data points and discuss selected topologies.

**Temporal Overhead.** Figure 7 shows Sluice’s latency relative to Deluge as the size of the network grows. Because the page size is fixed to be the same for Deluge and Sluice, we attribute this latency to the long signature-verification time. In Figure 8, the propagation is shown as a function of time. This allows us to see the completion rate of Deluge and Sluice as each protocol’s respective “wavefront” works its way through the network, updating nodes in its path. The delay between the the progress of the two protocols is due to the verification times of Sluice. Thus, Sluice tends to “mirror” the overall trend of Deluge’s performance, while simultaneously exaggerating Deluge’s behavior. It should be possible to mitigate this “fun-house” effect by reducing the amount of time taken by the verification function.

<sup>1</sup>The actual update disseminated was the SurgeTelos application, compiled with Sluice extensions.

<sup>2</sup>A 10x10 topology took us over 11 hours to complete on a 2.8 Ghz P4 machine.

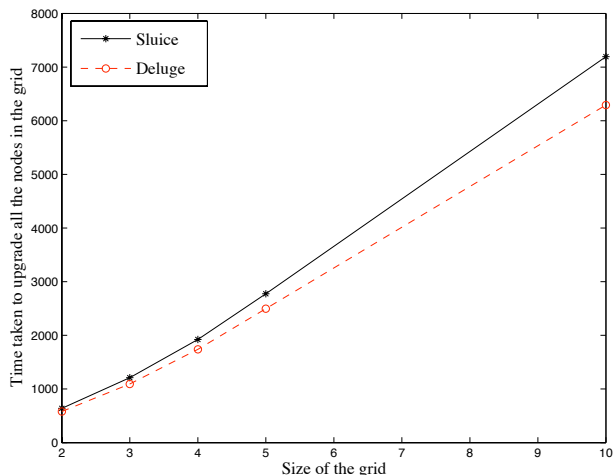


Figure 7. Latency with respect to grid size.

## 7 Related Work

Deluge provides limited integrity checking through the use of a 16-bit CRC on each page. The CRC provides no authentication, and does not guarantee update-level integrity since attackers can easily change the update in ways that are undetectable by the CRC mechanism. The developers of Deluge have recently and independently proposed an approach similar to ours [4]. Both approaches allow for the incremental verification of updates through the use of digital signatures and hash-chains. The primary difference is the granularity of the chaining mechanism. While we choose to amortize the cost of a hash by chaining across pages, Dutta et. al. adopt chains across individual packets. The advantage of a per-packet hash-chain strategy is that individual packet corruptions can be detected, so that a single bad packet will not trigger the retransmission of the entire page. However, this introduces a high per-packet overhead, which the authors attempt to reduce by truncating the generated 160-bit SHA-1 hash to 64 bits prior to transmission. Even with the truncated digest, the packet-chaining approach still adds 384 bytes of hash overhead per page, as opposed to Sluice’s 20 bytes per page, assuming the standard 48-packet pages that Deluge uses.

## 8 Looking Ahead

This work shows that Sluice is a viable mechanism to secure network reprogramming, but there is room for significant improvement in a number of areas.

Further testing in simulation and on actual hardware is warranted to evaluate Sluice’s performance and survivability at very large scales (>100 nodes). Our current experimental environment made it difficult to perform in-depth

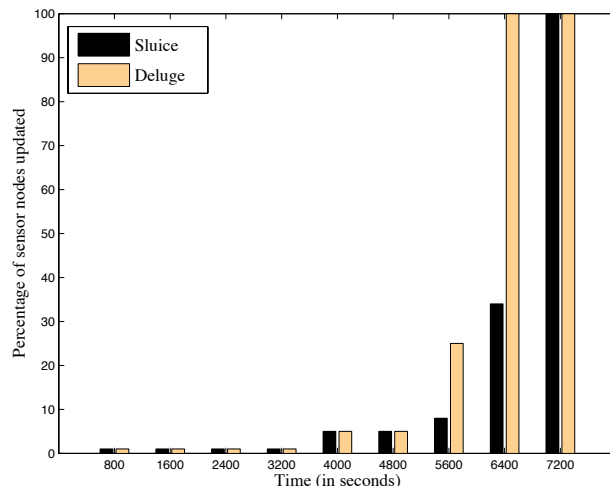


Figure 8. The number of updated nodes in a 10x10 grid network.

survivability analysis. We are currently in the process of expanding our sensor testbed to provide a 75+ node, multi-hop environment. Once this expansion is complete, the survivability of Sluice will be evaluated in the face of multiple malicious nodes. We also plan to reproduce and provide confidence intervals for experimental results presented in Section 6.

Our current implementation introduces a significant overhead in terms of update latency, largely due to the high signature verification time. We expect significant improvements can be made by optimizing the cryptographic libraries. Our current implementation of Sluice, as described in this paper, was *not an exercise in code optimization*. However, because Sluice is not dependent on any particular digital signature algorithm, other more efficient implementations [6, 4] could easily be substituted.

There are additional optimizations that we might make, in the interests of further resource conservation and efficiency. For example, our current mechanism does not distinguish between benign errors and malicious errors. Sluice prevents both types of errors from propagating, but benign errors are likely to be prevented by relatively straightforward CRC checks as well. We might be able to reduce the number of failed cryptographic operations by utilizing CRC values as a hint of whether a subsequent cryptographic verification will fail or succeed. This allows us to use a weaker, less expensive integrity check to form a first line of defense against benign errors, with the intent of reducing the number of wasteful cryptographic operations. In future work, we will also aim to secure other aspects of the network reprogramming process, e.g., the integrity of advertisements and of requests for missing pages.

## 9 Conclusion

In this paper, we presented Sluice, an extension to existing network reprogramming protocols that provides a number of security guarantees, including the prevention of malicious nodes from propagating or installing malicious updates on uncompromised nodes within the system. Sluice aims for the progressive, resource-sensitive verification of updates within sensor networks by exploiting a single digital signature per update, along with a hash-chain construction over pages of the update. We demonstrate the feasibility of our approach through an implementation of Sluice on a testbed of Telos sensor-nodes, along with a benchmarking of update latencies against its current underlying protocol, Deluge. With the lessons that we have learned in our implementation of Sluice, we intend to move towards understanding its trade-offs and capabilities better.

## 10 Acknowledgements

We gratefully acknowledge Jonathan Hui for providing us with the topology simulation files that Deluge was originally evaluated with. We also acknowledge An Liu and Peng Ning for their generous help in our experimentation with TinyECC. We acknowledge support through the NSF CAREER grant CCR-0238381 and the Army Research Office grant DAAD19-02-1-0389 (“Perpetually Available and Secure Information Systems”) to the Center for Computer and Communications Security at CMU.

## References

- [1] American National Standards Institute. ANSI X9.62-1998: Public key cryptography for the financial services industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), 1998.
- [2] Bouncy Castle Crypto APIs. <http://www.bouncycastle.org>.
- [3] Certicom Research. Standards for efficient cryptography - SEC2: Recommended elliptic curve domain parameters, 2000.
- [4] P. K. Dutta, J. W. Hui, D. C. Chu, and D. E. Culler. Securing the Deluge network programming system. In *The Fifth International Conference on Information Processing in Sensor Networks (IPSN'06)*, 2006.
- [5] R. Gennaro and P. Rohatgi. How to sign digital streams. *Information and Computation*, 165(1):100–116, 2001.
- [6] 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 *IEEE International Conference of Pervasive Computing and Communications*, pages 247–256, March 2005.
- [7] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In *International Workshop on Cryptographic Hardware and Embedded Systems*, volume 3156, pages 119–132, August 2004.
- [8] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *ACM International Conference on Embedded Networked Sensor Systems*, pages 81–94, November 2004.
- [9] C. Karlof, N. Sastry, and D. Wagner. TinySec: A link layer security architecture for wireless sensor networks. In *ACM International Conference on Embedded Networked Sensor Systems*, pages 162–175, November 2004.
- [10] S. S. Kulkarni and L. Wang. MNP: Multihop network programming for sensor networks. In *International Conference on Distributed Computing Systems*, pages 7–16, June 2005.
- [11] L. Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):770–772, November 1981.
- [12] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *ACM International Conference on Embedded Networked Sensor Systems*, pages 126–137, November 2003.
- [13] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The emergence of networking abstractions and techniques in TinyOS. In *Symposium on Networked System Design and Implementation*, pages 1–14, 2004.
- [14] A. Liu and P. Ning. TinyECC: Elliptic curve cryptography for sensor networks, September 2005.
- [15] D. Liu, P. Ning, and R. Li. Establishing pairwise keys in distributed sensor networks. *ACM Transactions on Information and System Security*, 8(1):41–77, February 2005.
- [16] D. J. Malan, M. Welsh, and M. D. Smith. A public-key infrastructure for key distribution in TinyOS based on elliptic curve cryptography. In *IEEE International Conference on Sensor and Ad Hoc Communications and Networks*, pages 71–81, October 2004.
- [17] National Institute of Standards and Technology. Secure hash standard, April 1997.
- [18] A. Perrig, R. Szewczyk, J. D. Tygar, V. Wen, and D. E. Culler. SPINS: Security protocols for sensor networks. *Wireless Networks*, 8(5):521–534, 2002.
- [19] A. Seshadri, A. Perrig, L. van Doorn, and P. K. Khosla. SWATT: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*, pages 272–282, May 2004.
- [20] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30, University of California, Los Angeles, Center for Embedded Networked Computing, November 2003.
- [21] A. Wacker, M. Knoll, T. Hieber, and K. Rothermel. A new approach for establishing pairwise keys for securing wireless sensor networks. In *ACM International Conference on Embedded Networked Sensor Systems*, pages 27–38, November 2005.
- [22] R. Watro, D. Kong, S. fen Cuti, C. Gardiner, C. Lynn, and P. Kruus. TinyPK: Securing sensor networks with public key technology. In *Workshop on Security of Ad Hoc and Sensor Networks*, pages 59–64, October 2004.