

Secure Network Reprogramming for Sensor Networks

Patrick E. Lanigan

May 2006

Abstract

Network reprogramming allows for over-the-air application updates in sensor networks. 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. We describe the operation of a number of network reprogramming protocols that have emerged for the TinyOS sensor network operating system. We go on to discuss potential security issues that arise from the operation of network reprogramming protocols. Compromised nodes can subvert such protocols to result in the propagation and remote installation of malicious code. We propose a new mechanism, Sluice, that aims for the progressive, resource-sensitive verification of updates in sensor networks. Sluice ensures 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.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Organization	2
2	A Survey of Existing Protocols & Security Issues	3
2.1	Network Reprogramming Protocols	3
2.1.1	Overview of Existing Protocols	5
2.1.2	Evaluation & Comparison of Existing Protocols	9
2.2	Security	10
2.2.1	Attacks	11
2.2.2	Potential Approaches	12
3	Sluice: Enabling Secure Network Reprogramming	15
3.1	Overview	15
3.1.1	Assumptions	15
3.1.2	Threat Model	15
3.1.3	Properties	16
3.2	Approach	17
3.2.1	Sluice’s Progressive Verification	18
3.3	Implementation and Evaluation	19
3.3.1	Discussion of Sluice’s Properties	20
3.3.2	Metrics of Interest	21
3.3.3	Experiments	21
3.4	Related Work	23
4	Future Work	25
4.1	Enhanced Experimental Model	25
4.2	Enhanced Efficiency	25
4.3	Enhanced Threat Model	26
5	Conclusion	27

List of Figures

3.1	Illustration of Sluice in operation.	16
3.2	Sluice's hash chain construction.	17
3.3	Processing at the trusted source.	19
3.4	Processing at each receiving node.	19
3.5	Layout of testbed nodes.	21
3.6	Latency with respect to (i) update size and (ii) grid size.	23
3.7	The number of updated nodes in (i) 5x5 and (ii) 10x10 grid networks.	24

List of Tables

2.1 Summary of protocols. 9

Acknowledgements

I would like to thank my advisor, Priya Narasimhan, and my reader, Rajeev Gandhi, for their unending encouragement and guidance. I would also like to thank the Information Networking Institute staff; my INI colleagues – particularly Matt Doyle, Benoit Durandeu, Eric Hough, Matt Karr, Ed Kenney, Josh Little, Jed Mitten, Loren Robinson, and Dan Ruef; and my ECE colleagues – particularly Tudor Dumitras, Aaron Paulos, Soila Pertet, Joe Slember, Sonya Wierman, and Drew Williams. Finally, none of this would have been possible without the support of my family – particularly my parents Patrick & Beverly Lanigan, and my sister Lauren Lanigan.

I would like to acknowledge Jonathan Hui at the University of California, Berkeley, for providing the topology simulation files that Deluge was originally evaluated with. I would also like to acknowledge An Liu and Peng Ning at North Carolina State University for their help with TinyECC.

I would further like to acknowledge the following sponsors for funding this research. This work has been partially supported by the General Motors Collaborative Research Laboratory at CMU and the Army Research Office through grant number DAAD19-02-1-0389 (“Perpetually Available and Secure Information Systems”) to the Center for Computer and Communications Security at Carnegie Mellon University. Additional funding was provided to me by the Frank J. Marshall Graduate Fellowship.

Chapter 1

Introduction

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 [10], MNP [17], MOAP [32], 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.

1.1 Motivation

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 up-

date 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 work 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 [29] sensor-nodes running TinyOS [20], and (ii) through sample topologies simulated in TOSSIM [19], the TinyOS simulation environment.

1.2 Organization

The field of network reprogramming has been studied over the past few years, and Chapter 2 is an attempt to describe and assess the current state-of-the-art. We focus mainly on protocols designed for distributing monolithic application images, the model used by the TinyOS [20] operating system. Section 2.1 begins with a description of properties that can be used to characterize current network reprogramming protocols and an overview of common mechanisms that are used to build such protocols. A discussion of how these mechanisms are implemented in current protocols is found in Section 2.1.1. We evaluate and compare these existing protocols in Section 2.1.2.

As sensor networks pervade safety-critical environments, security and survivability will likely emerge as serious issues. Section 2.2.1 identifies some potential issues, and Section 2.2.2 discusses some of the approaches that we considered for resolving these issues, what we learnt from them, and why we decided *not* to adopt them.

Chapter 3 describes a new approach for verifying the authenticity and integrity of code updates in sensor networks. Section 3.1 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 3.2 provides the details of our approach, while Section 3.3 describes our current implementation and evaluation. We discuss related work in Section 3.4. Chapter 4 discusses potential future directions for this work, and Chapter 5 concludes.

Chapter 2

A Survey of Existing Protocols & Security Issues

2.1 Network Reprogramming Protocols

A generic network reprogramming protocol might operate as follows. Initially, all of the sensor nodes in a network are running the same version of some distributed application. This *primary application* defines the main functionality of a node, such as data collection, processing, and communication patterns. We differentiate between the primary application and the network reprogramming protocol, which operates as a *secondary service* alongside the primary application. At some point during the operation of the network, a new application image becomes available from an *originating source*. The originating source could be a base station, or a sensor node recently introduced into the network. We will refer to the set of data comprising a new application version as a *data object*. A data object is most commonly a monolithic application image, but could be any large piece of data required to upgrade an application. When two or more nodes have different versions of a data object, we say that they are *inconsistent*.

The originating source announces the availability of the new data object, and its neighboring nodes become aware of an inconsistency. The data object is then distributed from the originating source to each inconsistent neighbor. These neighbors are then *eligible* to become sources themselves, and the process is repeated until no inconsistencies remain in the network. The point at which a node is eligible to become a source varies with the protocol¹. A data object is *pending* while it is in the process of being distributed. After the data object has been fully received by a node, it is considered *complete*. A reboot is typically required to load and execute the new application image, after which the completed data object becomes the *current* data object.

Properties. Various properties have been proposed to characterize network reprogramming protocols.

- *Reliability.* Unlike some sensor-network applications where packet losses are tolerable, a data object must be received in its entirety in order for the data to be usable. Since network reprogramming protocols typically operate in an inherently lossy network, they must sustain packet loss to provide complete program images.
- *Consistency.* In order to ensure that all distributed nodes are network state, the entire data object should *eventually* reach all of the nodes in the network. Nodes entering the network

¹See the description of *pipelining*, later in this section.

during or after an upgrade should still be ultimately able to receive the latest version of a data object.

- *Small Memory Footprint.* The update protocol should itself have a small memory footprint so as not to significantly restrict the amount of program space and RAM available to applications on the sensor node.
- *Energy Efficient.* Because energy is often a limited resource in sensor networks, an update protocol should minimize its energy usage so that it does not significantly affect network lifetime.
- *Minimize Disruptions.* The update protocol itself is a secondary service. As such, its operation should not significantly disrupt the functions of any primary application.
- *Fault Tolerance.* Network programming protocols should be tolerant to node failures and node additions. Failed and incoming nodes should not adversely affect the propagation of data objects to established live nodes.

These properties can be classified as *required* or *desired*. *Required* properties are those that define the correctness of a given protocol. *Desired* properties are not necessary for correctness, but enhance the usefulness and practicality of a protocol [32]. Generally, *reliability* and *consistency* are required, while the other properties are desirable. Since certain desirable properties are at odds with others, existing protocols make various tradeoffs between them. A discussion of the tradeoffs made by specific protocols is found in Section 2.1.2.

Components. Existing update protocols are typically comprised of a few core components that provide basic functionality. These core components can be summarized as follows.

- *Preparation Mechanism.* Since data objects are often larger than a single packet, they must be prepared for distribution before being injected into a network. Some protocols also require that certain initialization steps be performed among nodes before the dissemination process begins.
- *Dissemination Mechanism.* The dissemination mechanism allows nodes to learn of inconsistencies among sensor nodes and enables data propagation across multiple hops.
- *Reliability Mechanism.* An update is not useful unless it is received in its entirety. Since sensor networks often run over characteristically lossy wireless links, it is necessary to include a reliability mechanism to ensure that lost packets are recovered and all of the data is received.

In addition to the core components, different optimizations can be employed to increase performance in accordance with the specific design goals of a particular protocol. A few of the more common optimizations are:

- *Message Suppression.* Since all of the sensor nodes in the network are involved in disseminating the same data object, duplicate control and data messages may be sent by multiple nodes. Duplicate messages can lead to poor performance due to collisions and wasted energy. Various protocols take advantage of the broadcast medium to suppress redundant messages.
- *Sender Selection.* Often, more than one node is capable of forwarding data in any given neighborhood. Sender-selection mechanisms limit the number of active senders in a neighborhood.

- *Pipelining.* Allowing nodes to forward data before an object is complete propagates data in a *pipelined* fashion across the network, which reduces dissemination latency².

2.1.1 Overview of Existing Protocols

2.1.1.1 XNP

XNP [11] was the original TinyOS over-the-air update protocol. It employed a simple flooding mechanism to distribute updates over a single hop. No support was included for propagating updates over a multi-hop network.

2.1.1.2 Reijers et al.

Preparation. In [30], Reijers et al. propose a `diff`-like³ scheme to reduce the amount of data that needs to be transmitted. Instead of disseminating an entire program image, the authors propose creating an *edit script* by comparing the current data object with the pending data object. The edit script consists of commands such as *copy*, *insert* and *repair*. The edit script is divided into packets, such that each packet describes a certain number of program bytes and can be processed independently of other packets.

Dissemination. While the preparation mechanism is well developed, [30] describes only a simple point-to-point distribution mechanism.

Reliability. Each packet contains the starting address of the block of program bytes it describes. Nodes use this address information to determine where edit-script data is missing. Since each packet contains all of the information required to build an arbitrary number of bytes and can be processed independently, packets can arrive in any order.

2.1.1.3 Multi-hop Over-the-Air Programming (MOAP)

Preparation. MOAP [32] divides a data object into packets, and these packets are distributed through the network. Once received, packets are placed in stable storage until the entire update has been completed.

Dissemination. MOAP uses a mechanism called *Ripple* to distribute code through the network on a “neighborhood-by-neighborhood” basis. Ripple uses a publish-subscribe mechanism similar to Directed Diffusion [12], where sources advertise updated code images to which other nodes register interest. After nodes receive a full image, they become publishers and may propagate the image to other nodes out of range of the original source. This process is applied iteratively until the update has propagated across the network.

Reliability. MOAP places the responsibility for detecting packet loss at the receiver and uses a sliding-window approach to keep track of lost packets. When a missing packet is detected, the receiver sends a unicast retransmission request. If the source does not respond within a certain amount of time, the receiver broadcasts a retransmission request to which all nodes within range

²We define *latency* to be the time from when the dissemination process begins at the originating source, until all live nodes have received the complete data object

³`diff` is a unix utility that identifies differences between two files. See <http://www.gnu.org/software/diffutils/>

reply. This allows the receiver to choose a new source in case the original source fails. Duplicate requests arriving at a source within a given time period are suppressed.

2.1.1.4 Deluge

Preparation. Deluge [10] introduced the notion of *pages* as a unit of transfer. An update is broken up into a number of fixed-size packets, which are then grouped into segments (or *pages*) of N packets each. Each page can then be compared against a previous version to determine whether it has changed. This metadata is encapsulated in an *age vector*, which allows nodes to determine when a page changed, and whether they need to request it.

Dissemination. Deluge employs a three-phase *advertisement-request-data* handshake to propagate updates between nodes.

In the *advertisement* phase, all nodes broadcast summaries that consist of a version number and the number of the highest page available for transfer. If a node learns of inconsistencies (i.e. a neighbor advertises a lower version number), it transmits an object profile containing the age vector and a version number. Deluge borrows techniques from Trickle [21] to suppress redundant control messages, while dynamic advertisement intervals allow adjustment of the advertisement rate during upgrades versus during steady state operation. When a node hears an advertisement for a page it does not have, it will transition to the *request* phase unless it has recently overheard (1) a previous request for, or data packet of, a lower numbered page or (2) a data packet of the desired page.

During the *request* phase, a node sends requests containing a bitmap of the packets it needs to complete a given page. A node will return to the advertisement phase once it receives all of packets that constitute the page, or the reception rate drops below a certain threshold.

A node enters the *data* phase when it receives a request for data. A node in the data phase is responsible for transmitting packets from a particular page. The node adds requested packets to a set, and transmits them in round-robin order. Packets requested in subsequent request messages are added to the set as well. After a packet is transmitted, it is removed from the set and the node returns to the advertisement phase when the set is empty.

Reliability. Cyclic redundancy checks (CRCs) at the packet- and page-levels help protect against data corruption. If a CRC fails, all of the data represented by it must be retransmitted.

Retransmissions are enabled by a bitmap that tracks packets received for a given page. As in [32], receivers are responsible for detecting their own losses and retransmission requests function as "selective negative acknowledgments" by including the page bitmap in messages during the *request* phase.

Optimizations. Deluge introduced the notion of "spatial multiplexing", or *pipelining*, whereby nodes are allowed to transmit page n as soon as pages $0 \dots n - 1$ are complete as opposed to waiting for an entire data object to be completed before forwarding data. Deluge also attempts to reduce the transfer of redundant data by proposing the use of an age vector, which allows nodes to know when pages have last changed. Pages that have not changed since the last update are not requested by nodes in the *request* phase.

2.1.1.5 Multi-hop Network Programming (MNP)

Preparation. As in Deluge, data objects are divided into fixed-size segments each containing an equal number of fixed-size packets.

Dissemination. MNP [17] is designed as a state machine that incorporates the dissemination and reliability mechanisms. The state machine consists of 5 basic functional states (IDLE, DOWNLOAD, ADVERTISE, FORWARD and SLEEP), as well as an ephemeral FAIL state from which nodes transition immediately back to IDLE. The dissemination mechanism also introduces a *parent / child* relationship between senders (the parents) and receivers (the children).

Nodes in the IDLE state listen for *advertisement* messages and respond with unicast *download requests*. Nodes in the ADVERTISE state send *advertisements* with the ID of the highest segment that they have available for transmission. If a *start download* message is received while a node is in either IDLE or ADVERTISE, the node sets its parent to be the source of the message and transitions to DOWNLOAD. Nodes in the DOWNLOAD state receive packets and write the data directly to stable storage, instead of storing the entire segment in a RAM buffer. If all packets have been received correctly when the node receives an *end download* message, it transitions to ADVERTISE. Otherwise, it transitions to FAIL.

Nodes in the ADVERTISE state listen for *download requests*. When an advertising node decides to become a sender, it broadcasts a *start download* message and transitions to the FORWARD state. In this state, the node broadcasts requested packets from a particular segment. Once all of the requested packets have been sent, the node broadcasts an *end download* message and goes to SLEEP for a predetermined length of time.

Reliability. MNP's reliability mechanism is very similar to Deluge's. MNP places the responsibility for detecting packet loss on the receiver and retains in memory a bitmap of the current segment. The bitmap tracks which packets in the current segment are missing and is included in download requests; when a node receives a download request, it performs the union of the new request with a set of outstanding requested packets to determine which packets it needs to forward.

MNP also provides an optional *query/update* phase, which adds two new states: QUERY and UPDATE. After a sender in the FORWARD state has finished forwarding all of its requested packets, it will enter the QUERY state instead of going to SLEEP. The sender broadcasts a *query* message, to which its children respond with unicast NACKs. This allows children to recover a small number of packets more quickly than immediately transitioning to FAIL and waiting for a new advertisement.

To handle failed senders, nodes in the DOWNLOAD and UPDATE states use timeouts to determine whether a parent is still alive. If a child does not hear from its parent for a predetermined period of time, the child node will transition to FAIL. Note that the parent-child relationship is strictly one-way; parents have no knowledge of their children.

Optimizations. MNP introduced a *sender-selection* mechanism that attempts to explicitly limit the number of nodes that are transmitting data in a particular broadcast neighborhood. Eligible source nodes (ie. nodes in the ADVERTISE state) compete with each other to become senders; nodes that lose go to SLEEP for a predetermined period of time. Specifically, potential senders keep track of the number of *download requests* they have received, and the node with the highest number of requests wins. Potential senders announce their local request-count in *advertisement* messages. When a node responds to an *advertisement* with a *download request*, it includes the request count of the advertising node in the request. This allows nodes to learn the request counts of other potential senders up to two hops away. Additionally, MNP takes advantage of a pipelining mechanism similar to that used by Deluge.

2.1.1.6 Infuse

Preparation. Infuse [16] splits the data object into fixed-size packets prior to dissemination. Infuse also requires a Time Division Multiple Access (TDMA) schedule to be known prior to dissemination.

Dissemination. Infuse introduces a TDMA-based approach to code dissemination. Infuse assumes TDMA slots are assigned by the base station and that each time slot is long enough to transmit a single packet. Each node groups its neighbors into *predecessors* and *successors*. Predecessors of a node are those neighbors who forward a majority of packets before the node. Successors of a node are neighbors who forward a majority of packets after the node. Nodes listen during their predecessor's time slots to receive packets, and forward packets during their own time slots.

Reliability. Infuse uses *go-back-n* with implicit acknowledgments, placing the responsibility for detecting and recovering from lost packets on the sender. After forwarding a packet, the sender listens in its successors' time slots. If a successor does not forward the packet, it is assumed to be lost and retransmitted in accordance with the *go-back-n* algorithm. If a node does not hear from a successor after a certain number of retransmission attempts, it assumes the successor has failed and no longer listens for implicit acknowledgments from it.

Optimizations. Infuse suggests a notion of *preferred predecessors* to reduce the number of predecessors receiving implicit acknowledgments from a single node. Nodes choose a predecessor from whom they prefer to recover lost packets, and piggyback this information on forwarded data packets. Once the preferred status of a node is known, non-preferred predecessors will only listen for implicit acknowledgments with a certain probability (to allow recovery from a failed preferred predecessor). The preferred predecessor continues to listen for implicit acknowledgments during its successors' slot, although the authors note that message receptions can be further reduced by having the preferred predecessor listen probabilistically as well.

2.1.1.7 Sprinkler

Preparation. Sprinkler [25] splits the data object into fixed-sized packets prior to dissemination. Clustering and coloring algorithms must be performed over the network to compute a connected dominating set (CDS) and TDMA schedule. Sprinkler assumes that nodes are grouped into clusters, and that a single node from each cluster is designated as the cluster head. A connected dominating set (CDS) is computed over cluster heads. Members of the CDS are designated as senders, and are the only nodes responsible for forwarding data. A D-2 coloring algorithm is then run over the CDS nodes to determine a TDMA transmission schedule.

Dissemination & Reliability. The Sprinkler dissemination mechanism has 2 phases, *streaming* and *recovery*, each of which employ reliability mechanisms. Each node in the network selects a neighboring node as its parent. Parents are responsible for recovering lost packets at their children; however, the mechanism for doing so differs according to the phase.

During the *streaming* phase, CDS nodes forward newly heard data packets during their respective TDMA slots. Negative acknowledgments (NACK) and parent IDs are piggybacked on these data packets. When a parent receives a NACK, it retransmits the corresponding packet in its next slot.

<i>Protocol</i>	<i>Dissemination</i>	<i>Data Granularity</i>	<i>Pipelining</i>	<i>Recovery</i>
Reijers et al.	point to point	packet	no	N/A
MOAP	publish/subscribe	packet	no	sliding window + unicast NACK
Deluge	three phase (adv-req-data)	page	yes	bitmap + unicast NACK
MNP	three phase (adv-req-data)	page	yes	bitmap + unicast NACK
Infuse	TDMA - parent / child	packet	yes	go-back-n + implicit ACK
Sprinkler	TDMA - parent / child	packet	yes	implicit & unicast NACK

Table 2.1: Summary of protocols.

At the end of the streaming phase, all of the CDS nodes have a complete data object, but non-CDS nodes may still be missing packets. During the *recovery* phase, non-CDS nodes unicast requests for missing packets to their parents. These requests continue to be sent at specific intervals until each non-CDS node has a complete data object. In this phase, messages can be sent by both CDS (recovery data packets) and non-CDS (recovery data requests) nodes, so the previously computed TDMA schedule cannot be used. Instead, Sprinkler specifies that a simple RTS/CTS mechanism be used for coordination.

Optimizations. Sprinkler introduced local algorithms to compute a connected dominating set (CDS) among clusters of nodes, and D-2 vertex coloring among cluster heads. These algorithms are used to determine the set of senders and their transmission schedules.

2.1.2 Evaluation & Comparison of Existing Protocols

One way to improve energy efficiency and end-to-end dissemination latency is to reduce the amount of data that needs to be transferred. Deluge attempts to reduce the size of data objects through the age vector. The age vector allows for nodes which may be multiple versions behind the pending data object. However, even small source-code changes can cause large shifts in binary data that span multiple pages. While Reijers et al. is very effective at reducing the amount of data to be transferred, it takes advantage of instruction-set details to build the edit script, and so is architecture dependent. A platform independent approach based on the `rsync` algorithm is presented by Jeong et al. [13]. This approach effectively handles code shifts and small binary changes, but presents a negligible speedup with large code changes. Both approaches assume nodes to be in a consistent state, and can only update nodes that are a single version behind the pending data object.

An important aspect of reliability mechanisms is keeping track of which packets a node has received correctly, and which packets have been lost. A simple solution would be to use a bitmap of all of the packets in a data object. However, this would require an inordinate amount of state to be stored in memory. Moreover, since dynamic memory allocation is typically not well supported on sensor nodes, it would be necessary to statically allocate a bitmap that is large enough for the largest possible data object. MOAP handles this through the use of a fixed-size sliding window. By splitting a data object into a number of fixed size pages, Deluge and MNP can use a small bitmap per page as opposed to a large bitmap per data object, thereby reducing the memory required to store recovery state. Also, since pages have a fixed number of packets, the page bitmap can be statically allocated without wasted space.

A page-based approach also allows Deluge and MNP to take advantage of pipelining to reduce latency. On the other hand, MOAP takes the position that reducing latency is less important than minimizing energy and memory usage, and makes tradeoffs accordingly. Not allowing nodes to become senders until they have received a complete update precludes the use of pipelining to decrease

latency. However, this reduces the protocol’s program complexity. Using an initial unicast retransmission request prevents duplicate replies without any additional suppression mechanisms. The tradeoff is a reduction in program complexity (no additional suppression code) and energy usage (no duplicate replies) for increased latency if the original source should fail.

Due to the hidden terminal problem⁴, Deluge displays dynamic propagation behavior where data propagates faster along the edges of a network than across a diagonal. MNP’s sender-selection mechanism effectively accounts for hidden terminals, so the dynamic propagation seen by Deluge is absent. The use of a `SLEEP` state in MNP allows nodes to avoid idle listening, thereby conserving energy.

Since Infuse and Sprinkler use a TDMA approach, the transmission schedule is deterministic and the hidden terminal problem does not apply. Hence, dynamic propagation effects are absent here as well. The use of implicit acknowledgments in Infuse reduces control-traffic overhead. Using TDMA also allows for reduced idle listening, since nodes only need to listen in their neighbor’s slots. While the use of preferred predecessors reduces the number of slots that a node must listen in, it introduces probabilistic recovery guarantees. Also, it may be difficult for nodes to determine their predecessors and successors in non-uniform networks.

While Sprinkler introduced novel ways to optimize the set of senders, the approach has a number of drawbacks. First, energy usage is not distributed evenly through the network. Because the set of senders (nodes in the CDS) is static, these nodes will be depleted of energy much faster than non-CDS nodes. Also, location information is required to compute the CDS and nodes must remain stationary.

While it is desirable that an update protocol not significantly impact the performance of the primary application, this property is not well evaluated in the protocols discussed so far. One way that protocols attempt to minimize disruptions is by minimizing dissemination latency, under the assumption that the sooner that dissemination completes, the sooner the nodes can get back to work. Deluge and MNP leverage a dynamic advertisement rate to allow for fast propagation (shorter advertisement period) during the upgrade process and reduced overhead (longer advertisement period) once the network reaches consistency.

2.2 Security

While network reprogramming has emerged as a necessary mechanism for maintaining sensor networks, current protocols have not been designed with security in mind. Consequently, it may be possible for an adversary to subvert the operation of such protocols to prevent the dissemination of needed updates, waste network resources, disrupt the operation of current applications or execute arbitrary malicious code. Moreover, an adversary may be able to leverage for malicious purposes the same mechanisms that contribute to the efficiency of current protocols. For example, pipelining allows correct nodes to quickly propagate *legitimate* data objects across a network. An adversary that is able to hijack this mechanism could use it to quickly propagate *malicious* data objects across the network instead.

⁴The hidden terminal problem refers to the situation where, given three nodes a , b and c , $a \leftrightarrow b$ and $b \leftrightarrow c$ can communicate but a and c are out of range from each other. Thus, if a and c can not sense the other is sending, they may both try transmitting to b at the same time, where the packets will collide and be lost.

2.2.1 Attacks

A discussion of attacks on sensor network routing protocols is presented by Karlof et al. [15]. Many of these attacks are applicable to network reprogramming protocols as well. However, network reprogramming protocols have several distinguishing characteristics.

- While routing protocols typically focus on finding a path through the network to a specific destination, *every* node is a target in network reprogramming. Thus, the operation of a network reprogramming protocol affects every single node in the network.
- Network reprogramming protocols affect the actual execution state of network nodes. A vulnerability in a network reprogramming protocol could compromise the execution state of a node.
- In many protocols, there is an implicit ordering whereby messages have to be received (i.e. pipelining). Disrupting this order could lead to the failure of the protocol.

Therefore, the nature of network reprogramming protocols makes the network vulnerable to new types of attacks and could allow for serious consequences in the event of an attack. Some of the attacks that we envisage are described below. While it is difficult to quantify the severity of these potential attacks, these issues should not be overlooked when designing secure network programming protocols.

Malicious Data Injection. Some form of advertisement is necessary for nodes to learn of inconsistencies in the network, and to determine a need to upgrade. Thus, the very nature of network reprogramming makes current protocols vulnerable to their most serious attack. If a malicious node is able to inject packets into a network, it may also be able to induce other nodes to download an arbitrary data object by advertising the availability of a bogus update. This not only wastes network resources, but it also may allow for the execution arbitrary code.

Deluge, for example, is designed so that any node can initiate an upgrade. If an adversary sends a malicious object profile containing a very high version number, other nodes will request the malicious code object. This code object could disable future reprogramming attempts and shut down the network, or simply retask it for the adversary's gain. Deluge's dual CRC checks do not provide any protection against this attack; they simply ensure that the malicious data object is not corrupted. Once the data object has propagated, the adversary could broadcast a reboot message to have network nodes load and execute the code. The malicious code could either place the network under the adversary's control, or turn off network reprogramming altogether, necessitating a manual maintenance to correct the affected nodes.

The advertisement and publishing mechanisms employed by MNP and MOAP can be likewise subverted. While Infuse and Sprinkler do not specify how nodes learn of inconsistencies, similar attacks are likely possible.

Timing/State Attacks. In protocols that rely on actions taken by other nodes to determine local state and timing, malicious nodes can broadcast fake information to disrupt the operation of other nodes. Deluge allows nodes to dynamically change their advertisement period during steady-state operation versus during an active update. This is accomplished by listening for inconsistencies in the network. If a malicious node falsely advertises an inconsistency, it will force neighboring nodes to send advertisements rapidly in an attempt to update the malicious node. This rapid advertisement wastes energy, and could have an adverse impact on the operation of the primary application.

Deluge is also designed so that a node will not request a needed page if the transfer of a lower numbered page is in progress. This is intended to prioritize lower numbered pages in order to limit interference caused by the transfer of different pages in a single neighborhood. However, this restriction could be abused by a node requesting low numbered pages with a high enough frequency that neighboring nodes never enter the request phase. These nodes will still be able to save overheard data packets, but the inability to request data will break the chain of propagation in certain network topologies.

Sybil Attacks. In a Sybil attack [4], a single node presents multiple identities to its neighbors. In the context of network reprogramming protocols, this could allow attacks against parent/child relationships. For example, Infuse allows a node to identify predecessors and successors by determining which nodes forward a majority of packets before or after it. A malicious node could pose as both a successor and a predecessor to another node by using different identities to forward a packet at different times.

Sinkhole Attacks. Sinkhole attacks allow malicious nodes to attract a disproportionate amount of traffic from neighboring nodes. This could allow an adversary to propagate malicious code more quickly, or to block the progress of a legitimate update.

An effective sinkhole attack is enabled by the sender-selection mechanism used in MNP. By advertising with an artificially high request counter, a malicious node is able to “cheat” in the sender-selection competition. All of the potential receivers in the malicious node’s neighborhood will assign it as their parent, putting other eligible senders to sleep. This causes a denial of service at other potential sender nodes and allows for a *selective forwarding* attack where the malicious node refuses to forward data to its children, preventing the propagation of an update to that particular neighborhood.

2.2.2 Potential 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 [31], 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 [14] 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 protect 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 [23, 33] 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 TESLA [27] and its sensor network counterpart, μ TESLA [28] – 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. TESLA broadcasts digitally signed keychain commitments, while μ TESLA unicasts commitments to each receiver using pre-distributed symmetric keys. This limits μ 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 [18] 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 $H^i(h_i)$. Note there needs to be a way 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 [7, 8, 22, 24, 34] 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 [34]. The authors propose securing the XNP [11] module by running TinyPK verification after initializing the XNP process. However, the verification protocol only seems to verify that the broadcaster is an

authorized member of the network. This does not address the issue of an already verified node becoming compromised.

Gura et al. have shown that elliptic curve cryptography (ECC) can provide substantial performance gains over RSA on constrained platforms [8]. ECC versions of cryptographic functions are becoming more common, e.g., ECDSA [1] is the elliptic curve digital signature algorithm. EecM [24] provides a Diffie-Helman (DH) key-distribution mechanism based on elliptic curves, but does not address digital signatures. Sizzle [7] implements SSL using ECC on the MICA2 [9] and Telos [29] 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 [22] provides a freely available ECC implementation for TinyOS that includes an ECDSA module. This module 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.

Chapter 3

Sluice: Enabling Secure Network Reprogramming

3.1 Overview

The primary focus of Sluice¹ 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 3.1.

3.1.1 Assumptions

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.

3.1.2 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*

¹Existing network-programming protocols, such as Deluge, Trickle, Sprinkler and Infuse, evoke a water/flow-based theme. Because we aim to control the flow of these protocols for their own good, we chose the suggestive name, *Sluice*. Sluices are man-made valve-like infrastructures that are typically used to regulate water levels and flow rates in rivers and canals for various purposes, such as irrigation and transportation.

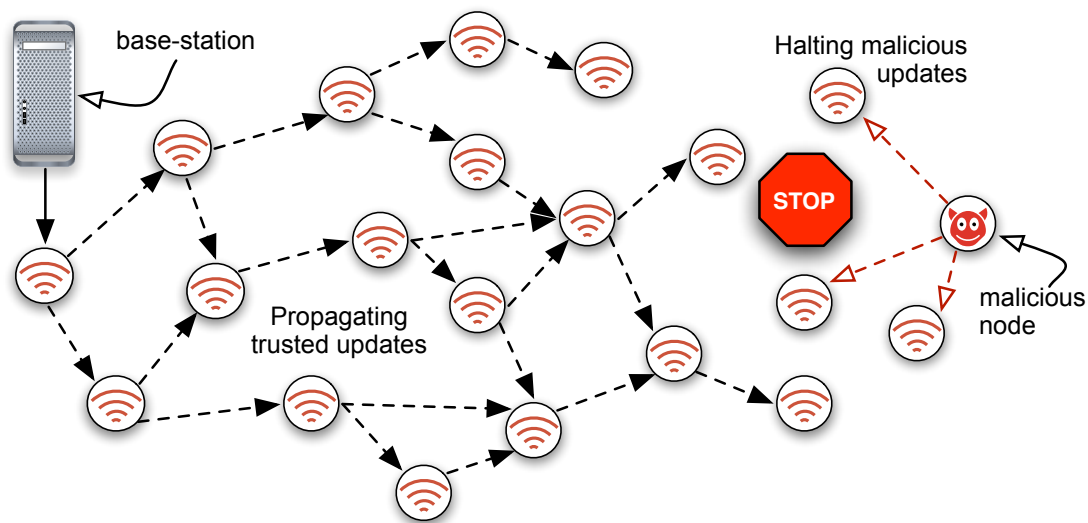


Figure 3.1: Illustration of Sluice in operation.

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-station, and the base-station 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.

3.1.3 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:

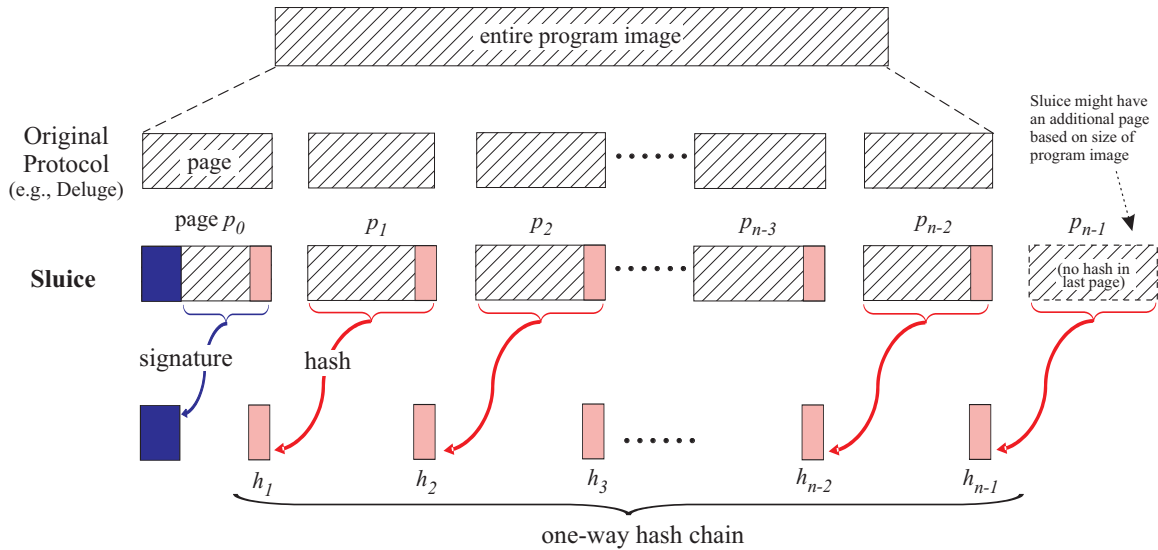


Figure 3.2: Sluice’s hash chain construction.

- **Completeness.** Every well-behaved node should eventually install a verified/trusted program update. This is a departure from previous work [10, 17, 32] 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.

3.2 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 [32], 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, which would negatively impact scalability in networks with a large diameter.

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.

3.2.1 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 [6]; 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 3.2 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, σ , 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 σ , 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 3.1.3 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 3.3). 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 signature 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 3.4).

```

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 3.3: 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
         $p' \leftarrow \text{HASH}(p_i)$ 
        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 3.4: Processing at each receiving node.

3.3 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, 26]. We use ECDSA [1] initialized with the `sec160k1` domain parameters specified in [3]. TinyECC [22] 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 [26]. We emphasize that these cryptographic libraries are off-the-shelf and *not* optimized for our platform.

We implemented Sluice as an extension to Deluge² [10], which defines 1104-byte pages by de-

²We have leveraged Deluge for our reference implementation because of its wide availability. However, our approach, as described in Section 3.2, is independent of Deluge, and can be used to extend *any* network reprogramming protocol, subject to the assumptions listed in Section 3.1.1

fault. 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 3.2). 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 3.2, 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.

3.3.1 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, 22], security parameters [3] and modes of operation [6]. 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.

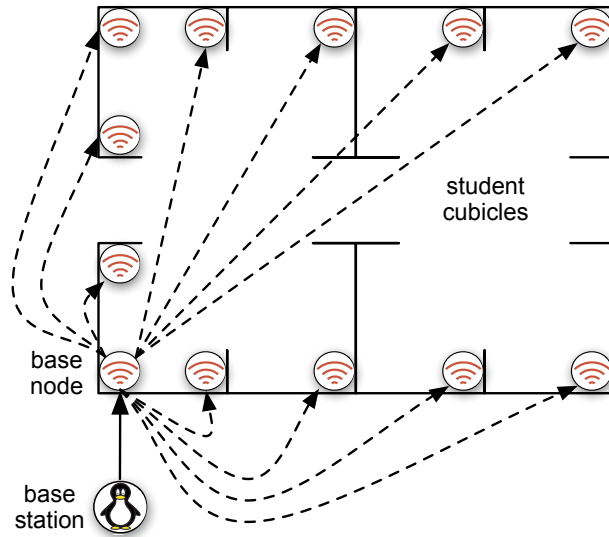


Figure 3.5: Layout of testbed nodes.

3.3.2 Metrics of Interest

We benchmark Sluice against its current foundation, Deluge, whose empirical evaluation is reported elsewhere [10]. 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.

3.3.3 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.

3.3.3.1 Testbed

Our current testbed (Figure 3.5) is comprised of 12 Tmote Sky nodes in an office setting. The Tmote Sky nodes are based on the Telos [29] architecture 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.

Theoretically, Sluice adds roughly $S_\sigma + n * S_h$ bytes to the size of an update, where S_σ 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 3.6. 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.

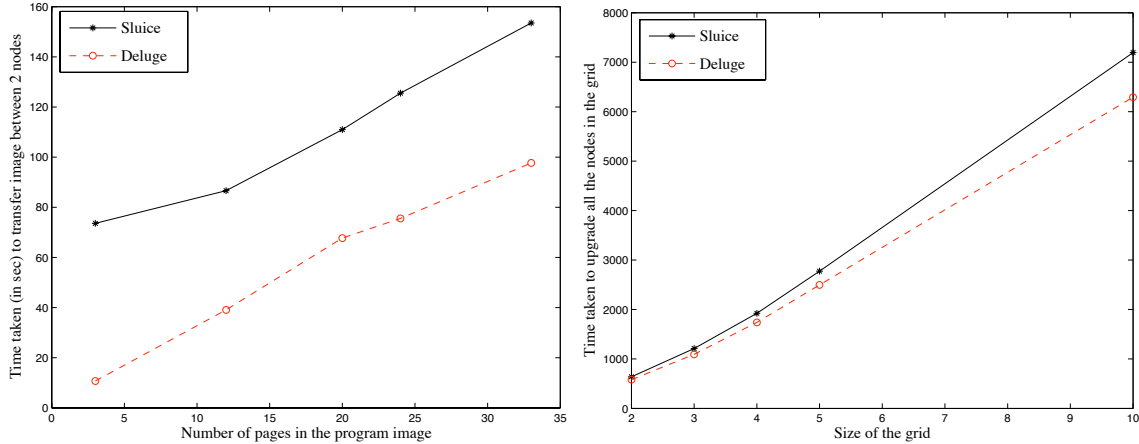


Figure 3.6: Latency with respect to (i) update size and (ii) grid size.

3.3.3.2 Simulation

We use TOSSIM [19] 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 update³. This page size is typical of what might be seen in actual applications. Due to simulation times on the order of hours⁴, we provide only single data points and discuss selected topologies.

Temporal Overhead. Figures 3.6 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 3.7, 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.

3.4 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

³The actual update disseminated was the SurgeTelos application, compiled with Sluice extensions.

⁴A 10x10 topology took us over 11 hours to complete on a 2.8 Ghz P4 machine.

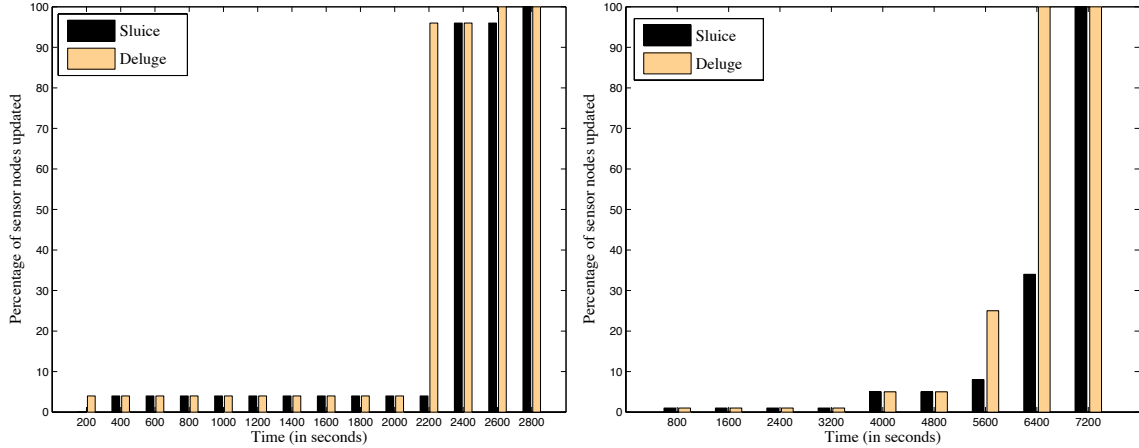


Figure 3.7: The number of updated nodes in (i) 5x5 and (ii) 10x10 grid networks.

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 [5]. 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. No supporting analysis is given by the authors regarding the security of the truncated hash. 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.

Chapter 4

Future Work

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.

4.1 Enhanced Experimental Model

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 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 3.3.

4.2 Enhanced Efficiency

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 through code optimization of the cryptographic implementations that we use. The TinyECC module includes AVR assembly-code optimizations for the 8-bit ATmega128 processor used in Mica2 series motes, which produces ECDSA verification times in the 13-second range. We believe that similar assembly-level improvements should be possible for the MSP430 microprocessor used in the Tmote Sky¹. The SHA-1 code was derived directly from publicly available reference code developed for 32-bit architectures, and would benefit from optimization as well. Our current implementation of Sluice, as described in this chapter, was *not an exercise in code optimization*. However, because Sluice is not dependent on any particular digital signature algorithm, other more efficient implementations [5, 7] 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

¹The authors of Sizzle report completing a full SSL handshake in under 4 seconds on the Telos platform using hand-written MSP assembly code. The ECDSA operations alone should take under 1 second.

a subsequent cryptographic verification will fail or succeed². 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.

4.3 Enhanced Threat Model

Currently, Sluice is effective at preventing nodes from propagating malicious updates. It is not designed to provide protection against denial-of-service. In a very simple denial-of-service attack, a malicious node could continually send corrupt pages and keep a node busy verifying bogus digital signatures. Also, Sluice does not provide explicit protection against replay attacks. Metadata (i.e. version numbers encapsulated in the advertisements) transmitted by the underlying protocol (e.g., Deluge) provides a level of protection against replay, but the advertisements must be tied to the image they represent to prevent a malicious node from advertising an old image with a newer version number. In our future work, we will aim to secure other aspects of the network reprogramming process, e.g., the integrity of advertisements and of requests for missing pages. Because protecting against many of these attacks would require more substantial changes to the underlying protocol, ultimately, we aim to move towards developing a new update protocol that is born resilient against a wide range of threats, including attacks that could partition/impede the update process.

²If a CRC fails, we can safely assume that the data is corrupt, and reject the page without proceeding to verify the hash or the digital signature. If the CRC succeeds, this is not conclusive – thus, any pages that pass the CRC check will still need to be verified against the digital signature or hash-chain, as appropriate, to provide authenticity and integrity.

Chapter 5

Conclusion

Current network reprogramming protocols are able to efficiently and reliably disseminate code updates in sensor networks. However, we identified a number of areas where these protocols may be vulnerable to attack. Sluice is 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.

Sluice does not provide protection against all of the vulnerabilities we identified. In order to provide protection against additional vulnerabilities, it may be necessary to rethink the way network reprogramming protocols operate and design a new breed of protocols that are born resilient against a wide range of attacks.

Bibliography

- [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, September 2000.
- [4] J. R. Douceur. The Sybil attack. In *International Workshop on Peer-to-Peer Systems*, pages 251–260, March 2002.
- [5] P. K. Dutta, J. W. Hui, D. C. Chu, and D. E. Culler. Securing the Deluge network programming system. In *Information Processing in Sensor Networks*, April 2006.
- [6] R. Gennaro and P. Rohatgi. How to sign digital streams. *Information and Computation*, 165(1):100–116, 2001.
- [7] 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.
- [8] 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*, pages 119–132, August 2004.
- [9] J. Hill and D. E. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, November 2002.
- [10] J. W. Hui and D. E. 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.
- [11] Crossbow Technology Inc. Mote in-network programming user reference, 2003.
- [12] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *International Conference on Mobile Computing and Networking*, pages 56–67, August 2000.
- [13] J. Jeong and D. E. Culler. Incremental network programming for wireless sensors. In *IEEE International Conference on Sensor and Ad Hoc Communications and Networks*, pages 25–33, October 2004.

- [14] 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.
- [15] C. Karlof and D. Wagner. Secure routing in wireless sensor networks: Attacks and countermeasures. *Elsevier's AdHoc Networks Journal*, 1(2–3):293–315, September 2003.
- [16] S. S. Kulkarni and M. Arumugam. Infuse: A TDMA based data dissemination protocol for sensor networks. Technical Report MSU-CSE-04-46, Department of Computer Science, Michigan State University, November 2004.
- [17] 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.
- [18] L. Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):770–772, November 1981.
- [19] P. Levis, N. Lee, M. Welsh, and D. E. Culler. TOSSIM: Accurate and scalable simulation of entire tinys applications. In *ACM International Conference on Embedded Networked Sensor Systems*, pages 126–137, November 2003.
- [20] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. E. Culler. The emergence of networking abstractions and techniques in TinyOS. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 1–14, March 2004.
- [21] P. Levis, N. Patel, D. E. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 15–28, March 2004.
- [22] A. Liu and P. Ning. TinyECC: Elliptic curve cryptography for sensor networks, September 2005.
- [23] 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.
- [24] 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.
- [25] V. Naik, A. Arora, P. Sinha, and H. Zhang. Sprinkler: A reliable and scalable data dissemination service for wireless embedded devices. In *IEEE International Real-Time Systems Symposium*, pages 277–286, December 2005.
- [26] National Institute of Standards and Technology. Secure hash standard, April 1997.
- [27] A. Perrig, R. Canetti, J. D. Tygar, and D. Song. The TESLA broadcast authentication protocol. *RSA CryptoBytes*, 5(Summer), 2002.
- [28] 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.
- [29] J. Polastre, R. Szewczyk, and D. E. Culler. Telos: Enabling ultra-low power wireless research. In *Information Processing in Sensor Networks*, pages 364–369, April 2005.

- [30] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *ACM International Conference on Wireless Sensor Networks and Applications*, pages 60–67, September 2003.
- [31] 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.
- [32] 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.
- [33] 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.
- [34] 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.