Seluge: Secure and DoS-Resistant Code Dissemination in Wireless Sensor Networks*

Sangwon Hyun, Peng Ning, An Liu
North Carolina State University

Wenliang Du
Syracuse University

Abstract

Wireless sensor networks are considered ideal candidates for a wide range of applications, such as industry monitoring, data acquisition in hazardous environments, and military operations. It is desirable and sometimes necessary to reprogram sensor nodes through wireless links after deployment, due to, for example, the need of removing bugs and adding new functionalities. The process of propagating a new code image to the nodes in a wireless sensor network is referred to as code dissemination. *This paper presents the design, implementation, and evaluation of an efficient, secure, robust, and DoS-resistant code dissemination system named* Seluge *for wireless sensor networks. Seluge is a secure extension to* Deluge*, an open source, state-of-the-art code dissemination system for wireless sensor networks. It provides security protections for code dissemination, including the integrity protection of code images and immunity from, to the best of our knowledge, all DoS attacks that exploit code dissemination protocols. Seluge is superior to all previous attempts for secure code dissemination, and is the only solution that seamlessly integrates the security mechanisms and the Deluge efficient propagation strategies. Besides the theoretical analysis that demonstrates the security and performance of Seluge, this paper also reports the experimental evaluation of Seluge in a network of MicaZ motes, which shows the efficiency of Seluge in practice.*

1 Introduction

A wireless sensor network is expected to consist of a potentially large number of low-cost, low-power, and multi-functional sensor nodes that communicate over short distances through wireless links. Due to the potential to provide fine-grained sensing and actuation at a reasonable cost, wireless sensor networks are considered ideal candidates for a wide range of applications, such as industry monitoring and military operations.

It is desirable and sometimes necessary to reprogram sensor nodes through wireless links after they are deployed, due to, for example, the need of removing bugs and adding new functionalities. The process of propagating a new code image to the nodes in a network is referred to as *code dissemination*. A few code dissemination protocols (e.g., [7, 13, 16, 22, 27, 28]) have been developed recently to propagate new code images using the ad-hoc wireless network formed by the sensor nodes. In particular, Deluge [13] uses an epidemic protocol [18] for efficient advertisement of meta data and spatial multiplexing for efficient propagation of code images. Deluge is generally accepted as the state of the art for code dissemination in wireless sensor networks, and has been included in recent TinyOS distributions [4].

In hostile environments where there may be malicious attacks against wireless sensor networks, code dissemination faces threats from both external attackers and potentially compromised nodes. For example, the adversary may attempt to modify or replace the real code image being propagated to sensor nodes, introducing malicious code into the sensor network. As another example, the adversary may inject bogus code dissemination packets and force normal sensor nodes to verify and/or forward them, thus exhausting their limited battery power.

1.1 Related Work

Several recent works have attempted to provide secure code dissemination for wireless sensor networks [8, 10, 17]. All these approaches are extensions to Deluge [13]. Lanigan et al. proposed a protocol named Sluice [17] to integrate signature and cryptographic hash functions to provide efficient authentication for code dissemination. This approach follows Deluge to divide each code image into pages. The hash image of each page is included in the previous page, while the hash image of the first page is signed and included in a signature packet. This approach, however, is vulnerable to Denial of Service (DoS) attacks. This is because in Sluice, a node can only perform authentication when an entire page is received, and thus it cannot authenticate a packet immediately after it is received. To exploit this property, the

*This work is supported by the National Science Foundation under grants CNS-0721424 and CAREER-0447761, and by the Army Research Office under grant W911NF-05-1-0247. The contents of this paper do not necessarily reflect the position or the policies of the US Government.

978-0-7695-3157-1/08 $25.00 © 2008 IEEE
DOI 10.1109/IPSN.2008.12

adversary may send a large number of bogus packets during code dissemination; a sensor node, upon receiving a packet, cannot tell if it is authentic or not, and has to save it if possible. As a result, the adversary can force the sensor nodes to save bogus packets but drop some authentic ones.

A scheme similar to Sluice was independently proposed in [10]. In this approach, the hash image of each code dissemination packet is included in the previous packet, and the hash image of the first packet is signed and included in an advertisement packet. A code dissemination packet can be authenticated only when the packet immediately before it has been received and authenticated. The authors of [10] proposed to optimistically store out-of-order packets. Unfortunately, this decision opens a door to the same DoS attacks, since the adversary can again send a lot of bogus packets to exhaust receivers' buffers. Not storing out-of-order packets is certainly an option to avoid the DoS attacks. However, this will lead to inefficient code dissemination: Losing one packet at a node will result in the retransmission of all the later packets in the same page.

Deng et al. proposed a scheme to improve the DoS-resilience of secure code dissemination by using Merkle hash tree [8]. Besides having the hash image of each page in the previous one, it also uses a Merkle hash tree to allow each packet to be immediately authenticated upon receipt. However, this approach adds additional overhead due to the transmission of a Merkle hash tree for *every* page [8]. Moreover, it distributes each hash tree in a level-by-level fashion; only after every packet in one level is received and verified can the packets in the next level be requested. This indeed disables the efficient page-by-page propagation in Deluge, leading to higher propagation delays.

Though all existing approaches [8, 10, 17] are based on Deluge [13], none of them provide a satisfactory solution to the authentication of advertisement and Selective Negative Acknowledgment (SNACK) packets. (In Deluge, such packets are used to advertise new data and facilitate the request and retransmission of packets.) Indeed, the approaches in [10, 17] overlooked the authentication of such packets. As a result, the adversary can forge such packets and exploit the Deluge epidemic and suppression mechanisms to launch DoS attacks. For example, the adversary may repeatedly request packets from a node to exhaust its battery power. Moreover, the adversary may request packets from a non-existing node; due to the Deluge suppression mechanism, others that overhear this request will not send request for packets (to real nodes). The authors of [8] discussed the authentication of SNACK packets, but overlooked the issue of advertisement packets.

Another critical issue common to these approaches [8, 10, 17] is the vulnerability to DoS attacks against the signatures used to bootstrap secure code dissemination. The approaches in [10, 17] did not consider such threats at all. As a result, the adversary can broadcast packets with bogus signatures, and force all the receivers to perform expensive signature verifications. The approach in [8] proposes to release previously undisclosed values in a one-way hash chain to mitigate such DoS attacks. However, this method is vulnerable to online attackers. Once the adversary overhears a hash value during a legitimate code dissemination, it can reuse the value to send forged signatures to other regions of the network. Due to the multi-hop and low-bandwidth nature of wireless sensor networks, the adversary has sufficient time to launch DoS attacks against many sensor nodes.

1.2 Proposed Approach

In this paper, we present the design, development, and evaluation of an efficient, secure, robust, and DoS-resistant code dissemination system named *Seluge* for wireless sensor networks. Seluge is an extension to Deluge [13], an open source code dissemination system included in TinyOS [4]. It inherits the efficiency and robustness properties from Deluge, and at the same time provides security protections for code dissemination, including the integrity protection of code images and resistance to the following three classes of DoS attacks: (1) DoS attacks against signature packets; (2) DoS attacks against code dissemination packets; and (3) DoS attacks against maintenance packets. To the best of our knowledge, these are all the DoS attacks that manipulate code dissemination protocols.

The key contribution of Seluge is a novel way to organize the packets used to distribute new code images. By carefully arranging code dissemination data items and their hash images in packets, Seluge provides *immediate authentication* of each packet upon receipt, without disrupting the efficient propagation mechanisms used by Deluge. Thus, it can defeat the DoS attacks exploiting authentication delays.

Seluge properly authenticates advertisement and SNACK packets. As a result, it can prevent DoS attacks exploiting the Deluge epidemic propagation and suppression mechanisms.

Seluge uses a signature to bootstrap the authentication of a new code image. However, unlike the previous attempts, Seluge uses a weak authentication along with the signature. This weak authentication mechanism has nice properties: It can be efficiently verified by a regular sensor node, but it takes a computationally powerful attacker a substantial amount of time to forge a weak authenticator. Moreover, it cannot be pre-computed. Thus, this weak authentication mechanism provides an effective filter of forged signatures. As a result, Seluge is not subject to the same DoS attacks against signature verifications as the previous approaches.

Note that Seluge does not deal with DoS attacks not targeted at code dissemination protocols (e.g., jamming attacks). We need other mechanisms to handle those threats.

Compared with the previous attempts [8, 10, 17], Seluge

not only provides integrity protection for code images, but is also resistant to various DoS attacks that exploit code dissemination protocols. Indeed, Seluge is superior to all the previous solutions [8, 10, 17], and is the only solution that seamlessly integrates the security mechanisms and the efficient Deluge propagation strategies.

1.3 Organization

The rest of this paper is organized as follows. The next section clarifies our assumptions and the threats to code dissemination in wireless sensor networks. Section 3 gives a brief overview of Deluge. Section 4 presents the techniques used in Seluge for secure and DoS-resistant code dissemination. Section 5 provides theoretical analysis of the security and performance of Seluge. Section 6 describes the implementation and experimental evaluation of Seluge in a network of MicaZ motes. Section 7 concludes this paper and points out some future research directions.

2 Assumptions and Threat Model

Assumptions: We assume the source of the code images, i.e., the *base station*, is a powerful node (e.g., a laptop PC) with sufficient energy supply. We assume that sensor nodes are resource constrained. A sensor node may be able to perform a limited number of public key cryptographic operations. For example, a MicaZ mote can perform a 160-bit ECC signature verification operation in about 2.43 seconds using TinyECC [19]. However, a node cannot afford performing many such operations due to the intensive computation and energy consumption. We assume the packet size is large enough to hold a signature and other information required by a signature packet. This can be satisfied on sensor platforms with IEEE 802.15.4 compliant radios [15], where the maximum payload size is 102 bytes. We assume each node has enough memory to store the disseminated code image (e.g., using the measurement flash).

We assume Deluge as the underlying code dissemination protocol. We assume the base station has a private and public key pair, and each sensor node in the network is preconfigured with the base station's public key. We also assume sensor nodes are able to establish pairwise keys between neighbor nodes, for example, using one of the existing schemes (e.g., [6, 9, 20].

Threat Model: We assume the adversary has access to computationally resourceful nodes such as laptops. The adversary may launch both *external* and *insider* attacks. In external attacks, the adversary does not control any valid node in the network. The adversary may attempt to eavesdrop for sensitive information, inject forged messages, replay previously intercepted messages, and impersonate valid sensor nodes. Moreover, the adversary may fake non-existing links by launching wormhole attacks [12]. The adversary may use Sybil attacks [23], where one node presents multiple identities to defeat typical fault tolerant mechanisms. The adversary may launch DoS attacks by, for example, forging a large number of signature packets or exploiting weaknesses of the code dissemination protocol. The adversary may jam the communication channel; however, we assume that the adversary cannot constantly jam the communication channel without being detected and removed.

The adversary may compromise some nodes to attack the rest of the network. We call such attacks *insider attacks*, since the compromised nodes are considered a part of the network before they are identified and removed. However, we assume that the majority of the nodes are not compromised. The adversary may exploit the compromised nodes in arbitrary ways to attack the remaining nodes. For example, the adversary may instruct the compromised nodes to intercept sensitive information even if the messages are encrypted, (selectively) drop packets, and launch Sybil attacks [23]. The adversary may also instruct the compromised nodes not to cooperate with others, inject false data, and exploit specific weaknesses of various protocols. However, we assume the base station cannot be compromised.

With those capabilities, the adversary attempts to disseminate illegal code images into the sensor network using the code dissemination mechanism, or launch DoS attacks to consume the limited resources (e.g., battery power, memory) on select sensor nodes.

3 Background: Deluge Overview

Deluge [13] is an open source code dissemination system for wireless sensor networks running TinyOS [4]. Deluge uses a page-by-page dissemination strategy. A code image is divided into fixed-size pages, and each page is further split into same-size packets. The pages for a code image is delivered in a sequential order. After completely receiving a page, a node advertises the availability of the newly received page, and may transmit the corresponding packets upon request. On the other hand, a receiver requests a page only after having received all packets in previous pages.

Deluge uses an epidemic protocol [18] for efficient advertisement of code meta data [13]. Each node periodically advertises the version of its code image and the number of pages it has for that version. For energy efficiency, the advertisement rate is dynamically adjusted: If a node discovers its own advertisement is different from those received from others, it increases its advertisement rate. Otherwise, it decreases the rate. As a result, Deluge can achieve rapid propagation during dissemination of a new code image, but consumes little resource in steady state.

Once a node finds out from an advertisement packet that a neighbor node has the page it needs, it uses Selective Negative Acknowledgment (SNACK) packets to request transmission from this neighbor. Each SNACK packet contains a requested page number and a bit vector indicating the re-

quested packets. Upon receiving several request packets for the same page, a node computes the union of the requested packets, and transmits the packets in a round-robin fashion.

Deluge uses various message suppression mechanisms for efficiency reasons. To reduce redundant advertisements, each node suppresses its own advertisement if the number of overheard advertisement packets having the same information is over a predefined threshold. Moreover, if a node overhears request (or data) packets for a page that it is about to request or has already received, it suppresses its own request packet. Similarly, if a node overhears request (or data) packets for the pages with smaller indices than that of the page it is currently transmitting, the node suppresses transmission of the subsequent data packets. By using such suppression mechanisms, Deluge increases the possibility that nodes in the same region wait for the same page and consequently maximizes the effect of overhearing.

4 Design of Seluge

Seluge relies on Deluge [13] for efficiency (via epidemic propagation and suppression) and robustness (via SNACK). To defend against the security threats against code dissemination, Seluge further adds three layers of protection: (1) Immediate authentication of code dissemination packets, (2) authentication of page advertisement and SNACK packets, and (3) anti-DoS protection for signature packets. The key contribution of Seluge is that it provides authentication and DoS-resistant protections by efficiently using cryptographic primitives, and at the same time still allows the efficient code dissemination mechanisms in Deluge.

We use the following notations: $H(\cdot)$ denotes a cryptographic hash function; $Sig(M)$ denotes the signature of M signed by the base station; $C \parallel D$ denotes the concatenation of C and D; and $|E|$ denotes the size of E in byte.

4.1 Immediate Authentication of Code Dissemination Packets

Following Deluge [13], we partition the code image to be disseminated into fixed-size pages. (For simplicity, we assume all pages have the same size. Our description can be modified slightly to accommodate the cases where the last page has a smaller size.) Assume there are P pages, denoted as page 1 through page P. We split each page i ($1 \le i \le P$) into N fixed-size packets, denoted as $Pkt_{i,1}$ through $Pkt_{i,N}$.

As discussed earlier, due to the Deluge page-by-page dissemination strategy, only after successfully receiving all packets in the current page does a node request the next page from a sender. We exploit this property to enable immediate authentication of each received packet at a node.

Construction of Code Dissemination Packets: Figure 1 illustrates our authentication scheme for the code image to be disseminated. We append the hash image of each

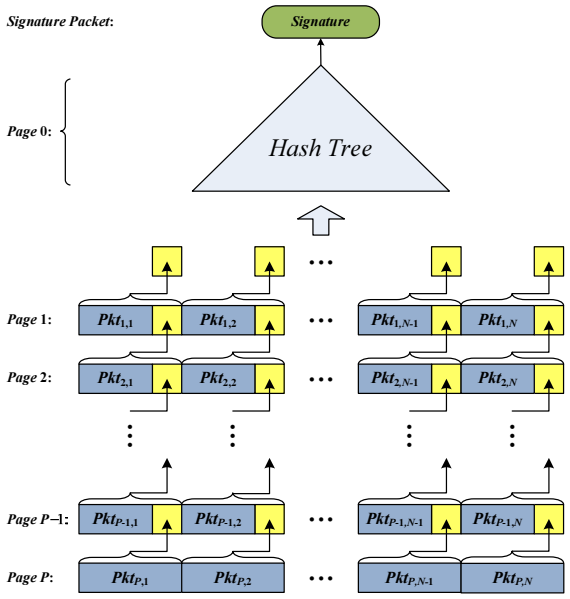

Figure 1. Authentication of code images.

packet in page P to the corresponding packet in page $P-1$. For example, the hash image of packet $Pkt_{P,1}$, $H(Pkt_{P,1})$, is included in packet $Pkt_{P-1,1}$. We then include the hash image of each packet in page $P-1$ in the corresponding packet in page $P-2$. This process continues until we finish hashing all the packets in page 2 and including their hash images in the corresponding packets in page 1.

As shown in Figure 1, we use Merkle hash tree [21] to facilitate the authentication of the hash images of the packets in page 1. We refer to the packets related to this Merkle hash tree collectively as *page 0*. Figure 2 illustrates the construction of page 0 and its packets. Specifically, we concatenate the hash images of the page 1 packets to form $HashValues = H(Pkt_{1,1}) \parallel \cdots \parallel H(Pkt_{1,N})$, and then fragment $HashValues$ into $M = 2^k$ pieces, where k is the minimum value that satisfies

$$\frac{N \cdot |H(\cdot)|}{2^k} + k \cdot |H(\cdot)| \le \text{Maximum payload size.} \quad (1)$$

(The condition in Equation 1 is to make sure each leaf node and its authentication path can be transmitted in one packet.) We denote the resulting fragments as $V_{0,1}$, $V_{0,2}$, ..., $V_{0,M}$, and construct a Merkle hash tree using $V_{0,1}$, $V_{0,2}$, ..., and $V_{0,M}$ as leaf nodes [21]. Figure 2 shows the construction of the Merkle hash tree when $N = 48$ and $M = 8$. Specifically, we compute $e_i = H(V_{0,i})$ $(i = 1, 2, ..., M)$, and build a binary tree by computing internal nodes from adjacent children nodes. Each internal node is the hash image of the two children nodes. For example, in Figure 2, $e_{1-2} = H(e_1||e_2)$, and $e_{1-4} = H(e_{1-2}||e_{3-4})$.

We then construct M packets using this Merkle hash tree. Specifically, we construct one packet $Pkt_{0,i}$ for each

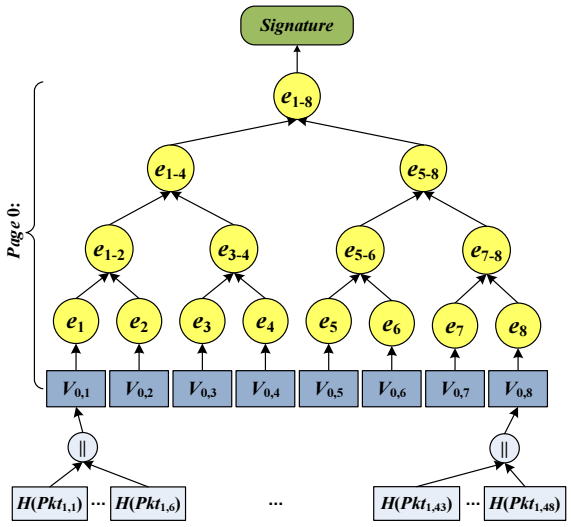

Figure 2. The Merkle hash tree constructed for page 0 when $M = 8$ and $N = 48$. Packet $Pkt_{0,i}$ ($i = 1, ..., M$) consists of $V_{0,i}$ and the values in its authentication path. For example, packet $Pkt_{0,1}$ consists of $V_{0,1}$, e_2, e_{3-4}, and e_{5-8}.

$V_{0,i}$, where $i = 1, 2, ..., M$; each packet $Pkt_{0,i}$ consists of $V_{0,i}$ and the values in its authentication path (i.e., the siblings of the nodes in the path from $V_{0,i}$ to the root) in the Merkle hash tree. For example, packet $Pkt_{0,1}$ consists of $V_{0,1}$, e_2, e_{3-4}, and e_{5-8} in Figure 2.

We include the root of the Merkle hash tree, the meta data about the code image (e.g., version number, size), and a signature over all of them in a *signature packet*. For the sake of presentation, we refer to the packets in page 1 through page P as *data packets*, and the packets in page 0 as *hash packets*. In addition to the payload discussed earlier, each packet also has a header describing auxiliary information about the code image, pages, and packets.

The reader may have noticed that we must choose parameters N and M carefully to ensure that the hash packets are small enough to be transmitted in wireless sensor networks, as indicated in Equation 1. This is indeed well accommodated by the current generation of sensor platforms that use IEEE 802.15.4 compliant radios [15], such as MicaZ [2] and TelosB [3] motes. For example, we may set the number of packets per page $N = 48$, following the default configuration in Deluge. We may use the 64-bit truncation of SHA-1 as $H(x)$, which provides sufficient pre-image resistance and has been used previously (e.g., [10]). Moreover, we may set the number of leaf nodes in the Merkle hash tree $M = 8$. As a result, each hash packet consists of 6 hash images of page 1 packets (48 bytes) and 3 hash images (24 bytes) in the authentication path in the Merkle hash tree. The total payload size is 72 bytes, smaller than the 102 bytes maximum payload size in the IEEE 802.15.4 standard [15].

Transmission & Authentication of Code Dissemination Packets: We rely on the underlying Deluge protocol to distribute packets for a given code image (in a *page-by-page* fashion). The additional capability provided by our packet construction is the *immediate authentication* of packets received by each node. This property is critical to sensor nodes in order to prevent DoS attacks aimed at exhausting receivers' buffers. Note that the approaches in [10, 17] do not provide this property, and thus are vulnerable to such DoS attacks. Though not vulnerable to such DoS attacks, the approach in [8] is much less efficient than Seluge.

The base station first broadcasts the signature packet, which serves as the advertisement of the new code image. Upon receiving a signature packet, each node verifies the signature to authenticate the root of the Merkle hash tree constructed for page 0. This root allows the node to authenticate each hash packet in page 0 upon receipt, using the values in the authentication path included in the same packet. For example, in Figure 2, if e_{1-8} has been authenticated in the signature packet, upon receiving a packet consisting of $V_{0,1}$, e_2, e_{3-4}, and e_{5-8}, a node can immediately verify whether $H(H(H(H(V_{0,1})||e_2)||e_{3-4})||e_{5-8}) = e_{1-8}$. If yes, the received packet is accepted; otherwise, it must be a forged packet and should be discarded right away.

Since the hash packets include all the hash images of the page 1 packets, successful receipt of them allows the node to further authenticate page 1 packets immediately upon receiving them. To continue the above example, the receipt of an authenticated $V_{0,1}$, where $V_{0,1} = H(Pkt_{1,1})||\cdots||H(Pkt_{1,6})$, implies the correct receipt of $H(Pkt_{1,1})$, ..., $H(Pkt_{1,6})$. Thus, when this node receives packet $Pkt_{1,i}$ ($i = 1, ..., N$) in page 1, it can immediately verify whether hashing $Pkt_{1,i}$ results in $H(Pkt_{1,i})$ and decide whether the received packet should be accepted or discarded. Following the same reasoning, receipt of the packets in page i ($i = 0, 1, ..., P - 1$) allows a node to authenticate all the packets in page $i + 1$ independently and immediately after those packets are received.

4.2 Authentication of Page Advertisement and SNACK Packets

As discussed in Section 1.1, several efficient code propagation mechanisms used by Deluge are vulnerable to exploits. The root cause of these vulnerabilities is the lack of authentication. We have provided immediate authentication of code packets in the previous subsection. However, the adversary can still exploit page advertisement and SNACK packets. Thus, advertisement and request packets must be authenticated as well. Because of the heavy use of overhearing and suppression, such authentication must be (local) broadcast authentication, i.e., a node can authenticate

any packet transmitted by its neighbors.

We use *cluster keys* for local broadcast authentication. (This was first mentioned in [8]; however, the authors of [8] did not give specific details.) Specifically, each node generates a per-node cluster key, which is intended to authenticate all the advertisement and SNACK packets transmitted from itself. When a node is deployed, it notifies its neighbors through periodic hello packets. We assume sensor nodes can establish pairwise keys between neighbor nodes using an existing scheme (e.g., [6, 9, 20]). Upon receiving a hello packet from a new neighbor, after a random delay, each node replies with its cluster key to the sender encrypted using their pairwise key. Moreover, a node that just sends a cluster key to a new neighbor also broadcasts a hello packet so that the new neighbor can reply with its own cluster key.

For each outgoing page advertisement or SNACK packet, the sender includes a unique sequence number (to prevent replay attacks), and authenticates the packet using its cluster key. Each node stores the cluster keys of its neighbors. For each incoming page advertisement or SNACK packet, a node uses the sender's cluster key to verify its integrity. A node simply discards unauthenticated or duplicate packets. The limitation of this approach is that it cannot uniquely identify senders. As a result, a compromised node can pretend to be its neighbors using their cluster keys.

In an earlier version of this paper [14], we discussed a complementary approach that uses μTESLA for local broadcast authentication. The μTESLA approach provides *true* broadcast authentication of page advertisement and SNACK packets. However, due to the use of μTESLA, there has to be either receiver side or sender side delay [25, 26]. The current version of Seluge adopts the cluster key approach due to its simplicity. We will explore the μTESLA approach in our future work.

4.3 Mitigating DoS Attacks against Signature Packets

All the previous secure code dissemination schemes [8, 10, 17] as well as Seluge use a signature to bootstrap the authentication of a code image. This signature is vulnerable to DoS attacks: The adversary can inject bogus signature packets into the network, force the nodes that receive such packets to perform expensive signature verifications, and eventually exhaust their limited battery power.

Seluge adapts a recently developed weak authentication mechanism called *message specific puzzles* [24] to mitigate such DoS attacks [24]. This method has a setup phase before the deployment of sensor networks. During setup, the base station generates a one-way key chain consisting of K_0, K_1, ..., K_n, where $K_i = H(K_{i+1})$ ($i = n-1, n-2, ..., 0$) and $H(\cdot)$ is a cryptographic hash function. This is done by randomly selecting K_n and repeatedly performing hash function H to K_n, as shown in Figure 3. The base station then pre-distributes the *key chain commitment* K_0 to all sensor nodes before deployment. The keys $K_1, K_2, ..., K_n$ are called *puzzle keys*, and the puzzle key K_i is used for the ith version of the disseminated code image.

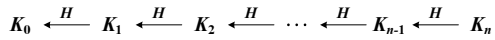

Figure 3. One-way key chain for puzzle keys.

We use message specific puzzles to provide another layer of protection for the signature packet of each code image. For each version i, we use the puzzle key K_i to generate a puzzle. Consider the signature packet of version i code image, denoted as $i||M_i||Sig(i||M_i)$, where i is the version number, M_i represents the collection of the other fields in the signature packet, and $Sig(i||M_i)$ is the signature generated by the base station. The signature packet $i||M_i||Sig(i||M_i)$ and the puzzle key K_i constitute a message specific puzzle. A valid solution P_i is such a value that after applying the hash function H to $i||M_i||Sig(i||M_i)||K_i||P_i$, the first l bits of the resulting image are all "0", as illustrated in Figure 4. The parameter l determines the strength of the puzzle. Before transmitting the signature packet, the base station first tries to solve the puzzle by finding the puzzle solution P_i. Then the base station broadcasts the final signature packet $i||M_i||Sig(i||M_i)||K_i||P_i$.

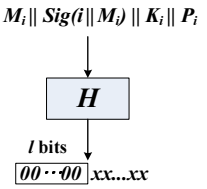

Figure 4. Message specific puzzles.

Upon receiving a signature packet, each node first verifies that the puzzle key is valid using H and K_0 (or a previously verified puzzle key) and that the puzzle key has not been used along with a valid signature before. Only when this verification is successful does the node verify the puzzle solution. If the puzzle solution is invalid, the receiver will simply drop the signature packet. Thus, without first solving some message specific puzzles with a fresh puzzle key, the adversary cannot force a node to verify signatures in forged packets.

Message specific puzzles can effectively mitigate DoS attacks against signature packets in code dissemination. The puzzle solution in each signature packet can be efficiently verified by a regular sensor node through a few hash function operations and comparisons. However, a puzzle solution can only be found through brute-force search due to the one-way property of the hash function. Moreover, though it takes the same effort for both the base station

and the adversary to solve a puzzle formed by a signature packet, the base station has a clear advantage over the adversary due to the prior knowledge of the puzzle keys: The base station has enough time to solve a puzzle off-line before disseminating a new code image. In contrast, the adversary has to solve puzzles after seeing a puzzle key but before the puzzle key becomes invalid when the signature packet reaches the target sensor nodes. Thus, with an appropriate puzzle strength, the message specific puzzle mechanism substantially increases the difficulty of launching DoS attacks against signature packets. Detailed analysis of message specific puzzles can be found in [24].

5 Analysis

5.1 Security Analysis

Integrity of Code Images: In Seluge, the trusted base station uses a digital signature to authenticate the root of the Merkle hash tree in page 0, with the private key only known to itself. All the sensor nodes know the public key of the base station, and thus can verify the signature. Under the assumption that the adversary cannot compromise the base station, it is guaranteed that all sensor nodes can authenticate any received signature packet as well as the root of the Merkle hash tree contained there. This means that all the nodes can authenticate the hash packets in page 0 once they receive such packets, based on the security of Merkle hash tree [21]. The hash packets include the hash images of the data packets in page 1. Thus, after verifying the hash packets, a sensor node can easily verify the data packets in page 1 based on the one-way property of cryptographic hash functions. Likewise, once verifying the data packets in page i, a sensor node can easily authenticate the data packets in page $i+1$, where $i = 1, 2, ..., P-1$. In summary, if the adversary injects a forged or modified code image, each receiving node can detect it easily because of the (immediate) authentication of code dissemination packets.

Resistance to DoS Attacks: As discussed in Section 1.1, there are three types of DoS attacks against Deluge based code dissemination: (1) DoS attacks exploiting authentication delays, (2) DoS attacks exploiting the expensive signature verifications, and (3) DoS attacks exploiting the Deluge propagation and suppression mechanisms.

Seluge is resistant to all three types of DoS attacks from *external* attackers. Due to the page-by-page dissemination strategy, upon receiving a packet, each node should have already received its hash image in the corresponding packet of the previous page (or in a hash packet in page 0). Thus, it can immediately authenticate any packet it receives in the current page, and successfully defeat DoS attacks exploiting authentication delays. Moreover, because of the use of message specific puzzles [24], each node can perform a few efficient hash function operations and comparisons to detect fake signature packets. Thus, Seluge provides resistance to DoS attacks that send fake signature packets. Finally, Seluge uses cluster keys to authenticate every advertisement or SNACK packet. As a result, an external attacker cannot convince regular nodes to misuse the propagation or suppression mechanisms.

Seluge can successfully defeat the first two types of DoS attacks even if there are compromised nodes. Indeed, without the private key and the unreleased puzzle keys on the base station, even an inside attacker cannot forge any code dissemination packets. However, Seluge cannot entirely prevent compromised nodes from launching the third type of DoS attacks that exploit the Deluge propagation and suppression mechanisms. A compromised node may misuse Deluge propagation and suppression mechanisms to mislead its neighbors. Fortunately, such DoS attacks are hard to coordinate and easy to detect, and the impacts are local to the compromised nodes. We will investigate how to detect such misbehaving nodes in the future.

5.2 Performance Analysis

Communication Overhead: We denote the code image as CI, and the maximum payload size per packet as $|payload|$. We first analyze the communication overhead to set up cluster keys between neighbor nodes, and then derive the communication overhead to disseminate a code image. We omit the analysis of overhead due to advertisement and request packets, because it remains the same as in Deluge.

As described in Section 4.2, each new node periodically broadcasts hello packets for a while to notify its neighbors. When a node hears a hello packet from a new neighbor, it sends its encrypted cluster key to the sender, and requests the sender's encrypted cluster key back. This cluster key exchange phase is performed for a limited period of time for each node. Thus, the total number of hello and cluster key packets that each node transmits is limited, though it varies depending on the actual deployment parameters.

The communication cost for propagating a code image includes a *signature packet*, *hash packets*, and *data packets*.

The total number of data packets depends on P, the number of the pages. Given the parameters N (number of packets per page), $|payload|$ (payload size available for code), and $|H(\cdot)|$ (size of a hash image), P can be determined as follows: Since each packet in all the pages except for the last one should deliver a single hash value, each packet in page 1 to page $P-1$ has $|payload| - |H(\cdot)|$ bytes available for code, and each packet in page P has $|payload|$ bytes all available for code. Thus, we can calculate $P - 1 = \lceil \frac{|CI| - N \cdot |payload|}{N \cdot (|payload| - |H(\cdot)|)} \rceil$. The number of packets in the last page can be calculated as $\lceil \frac{|CI| - N(P-1)(|payload| - |H(\cdot)|)}{|payload|} \rceil$, which may be less than N. Thus, the total number of data packets is $(N(P-1) + \lceil \frac{|CI| - N(P-1)(|payload| - |H(\cdot)|)}{|payload|} \rceil)$, where $P - 1 = \lceil \frac{|CI| - N \cdot |payload|}{N \cdot (|payload| - |H(\cdot)|)} \rceil$.

Now consider the number of hash packets in page 0, which is the number of leaves in the Merkle hash tree. As discussed in Section 4.1, the number of hash packets is $M = 2^k (k \geq 1)$, where k is the minimum value that satisfies the following inequality: $\frac{N \cdot |H(\cdot)|}{2^k} + k \cdot |H(\cdot)| \leq |payload|$.

Storage Overhead: Now we analyze the maximum buffer size required on each node. In Seluge, each node needs to authenticate advertisement and request packets from its neighbors using the right cluster keys. Thus, each node should store these cluster keys. Suppose each node keeps at most m cluster keys for its neighbors. Moreover, each node needs to store the hash images of the packets in the page to be received; such hash images are distributed in the corresponding packets in the previous page. Note that once a packet is received correctly, its hash image can be discarded, and the buffer entry can be reused for the hash image of the packet in the next page. Thus, each sensor node needs to have buffer for at most N hash images. In total, the maximum buffer size required by Seluge on each sensor node is $m \times |K_c| + N \times |H(\cdot)|$, where $|K_c|$ denotes the size of a cluster key.

Computation Overhead: Now we analyze the computation cost that Seluge requires on regular sensor nodes. Let us first consider attack-free cases. For each cluster key message, a sender adds message integrity code (MIC) for authentication and then encrypts the cluster key. A received cluster key packet is decrypted first and then verified with the MIC in the packet. Therefore, one MIC generation (or verification) and one encryption (or decryption) per transmission of a cluster key message are required on the sender (or receiver). Each advertisement or SNACK packet requires a MIC for authentication, and thus a sender (or a receiver) needs to generate (or verify) a MIC.

Consider the computation required to authenticate one code image. For each signature packet, two hash operations are needed to verify the puzzle key and the puzzle solution, respectively, and a signature verification operation is performed. Each hash packet is verified by $\log M + 1 = k + 1$ hash operations, and the hash packets together requires $M(\log M + 1)$ hash operations. Each of the remaining data packets is verified by a single hash operation. We already analyzed the total number of data packets earlier in this subsection. Thus, in attack-free situations, the total computation cost required to verify a single code image includes one signature verification and $(2 + M(\log M + 1) + N(P-1) + \lceil \frac{|CI| - N(P-1)(|payload| - |H(\cdot)|)}{|payload|} \rceil)$ hash operations, where $P - 1 = \lceil \frac{|CI| - N \cdot |payload|}{N \cdot (|payload| - |H(\cdot)|)} \rceil$.

When there are attacks, a node being attacked must perform more computation. The actual computation depends on the volume of the attacks. However, as discussed in Section 5.1, the extra computations are mostly those that can be efficiently performed, such as hash operations.

5.3 Comparison with Previous Approaches

Comparison with Sluice [17] and Berkeley approach [10]: Sluice and the Berkeley approach have similar constructions as well as similar properties. They can prevent malicious code images from being accepted at sensor nodes. However, both of them are vulnerable to several types of DoS attacks. First, they are both vulnerable to DoS attacks exploiting authentication delays. As discussed in Section 1.1, the adversary can send a large number of bogus packets to exhaust the buffers at receiving nodes. Second, both of them overlooked the authentication of advertisement and SNACK packets. The adversary can attack sensor nodes by misusing the Deluge propagation and suppression mechanisms. Finally, there is no protection for the signature packet in either approach. This allows the adversary to exhaust the battery power on sensor nodes by sending a large number of forged signature packets. In contrast, as discussed in Section 5.1, Seluge can guarantee the code image integrity and deal with all these attacks.

Comparison with Colorado approach [8]: The Colorado approach can provide code image integrity protection. In addition, it allows each code packet to be immediately authenticated upon receipt, and thus is not vulnerable to DoS attacks exploiting authentication delays. Though the Colorado approach achieves the same property as Seluge, it is much less efficient than Seluge. The Colorado approach uses a per-page Merkle hash tree; a node transmits a request packet for each level in the tree, and waits for the packets only at the requested level. This essentially disrupts the efficient page-by-page propagation mechanism used by Deluge. As a result, this approach adds not only additional packets to transmit, but also additional propagation delay. In contrast, Seluge seamlessly integrates the Deluge page-by-page propagation mechanism.

As discussed in Section 1.1, the Colorado approach is vulnerable to online DoS attacks against the signature packets. Moreover, though the Colorado approach discussed the possibility of authenticating SNACK packets to partially address the DoS attacks exploiting the Deluge propagation and suppression mechanisms, it overlooked the authentication requirements for advertisement packets. Therefore, it is still vulnerable to similar DoS attacks. In contrast, Seluge can handle both types of DoS attacks gracefully.

6 Implementation and Experiments

6.1 Implementation

We implement Seluge as an extension to Deluge 2.0 in the current TinyOS distribution. Our implementation has both base station side and sensor side programs. The base station side programs are Java programs expected to run on

a PC. They extend the Deluge Java tools to construct and inject new code dissemination packets into sensor networks. The sensor side program is written in nesC [11] and runs on regular sensor nodes.

We use the 64-bit truncation of SHA-1 as the hash function H. It provides sufficient pre-image resistance, and has been used previously (e.g., [10]). For digital signatures, we use ECDSA over the 160-bit elliptic curve `secp160k1`, which is defined in [5]. On the base station side, we use the JCE provider in the Bouncy Castle Crypto APIs [1] for hash function, key generation, and signature generation operations. On each sensor node, we integrate TinyECC [19] into Seluge to perform hash function and signature verification operations. Moreover, we use the hardware cryptographic support in the CC2420 radio component on MicaZ motes [2] for symmetric cryptographic operations, including the encryption (using AES) and authentication (using CBC-MAC) of cluster keys, and the authentication of page advertisement and SNACK packets.

We add the following functionalities in the Java tools on the base station side: Computation of the hash images of the data packets from the last to the first page, construction of the page 0 Merkle hash tree and then the hash packets from the hash images of the page 1 packets, and generation of the signature packet from the root of the Merkle hash tree and the meta data of the code image (e.g., version number, size). We include the message specific puzzle mechanism developed in [24] in both the Java tools and the sensor programs.

We add a `PacketVerifier` module into the Deluge nesC library to perform verification of signature packets (including both puzzle and signature verification), hash packets, and data packets. The commitment of the puzzle key chain used in message specific puzzles and the public key of the base station, which are generated by the Java tools, are pre-distributed to all nodes. The pairwise keys used to distribute cluster keys are also pre-distributed to all nodes.

Table 1. Code size (bytes) on MicaZ.

	ROM	RAM
Deluge	22,226	1,123
Seluge	45,258	2,278
TinyECC in Seluge	13,044	426

Table 1 shows the ROM and RAM usage of Seluge on MicaZ motes. The code size of Deluge and that of TinyECC are also included for reference purposes. It is easy to see that Seluge increases both the ROM and RAM consumption compared with Deluge, and the majority of the ROM increase is due to TinyECC.

6.2 Experimental Evaluation

We have provided theoretical analysis of the security and performance properties of Seluge in Section 5. In this subsection, we report the experimental evaluation of Seluge in a network of MicaZ motes [2]. For comparison purposes, we performed the same set of experiments with Deluge [13]. Moreover, as discussed in Section 1.1, the Berkeley approach [10] can be revised to mitigate the DoS attacks against dissemination packets. We obtained the source code from the authors of [10], made the revision, and used it in our experiments. Finally, we implemented the Colorado approach [8] and included it in our experimental evaluation. However, we did not include Sluice [17], since it offers much weaker security properties than the other approaches.

We use two performance metrics in our evaluation: *Propagation delay* and *communication overhead*. The propagation delay is the time required to finish disseminating a code image to all the nodes in the network. As mentioned in [13], for performance reasons, Deluge requires that every node keep its radio on. Thus, the propagation delay is closely related to the energy consumption required by code dissemination. The communication overhead is measured as the total number of packets transmitted by all the nodes during a code dissemination, which is also related to radio power consumption. Moreover, we also examine the propagation dynamics on individual nodes during the code dissemination to understand how each node receives different pages of the code image.

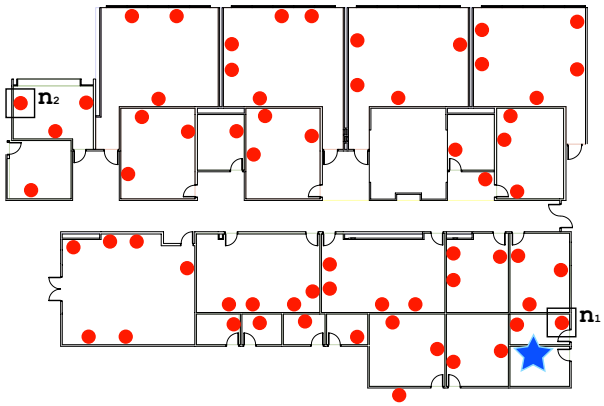

Figure 5. The testbed (65 MicaZ motes; 152.5 feet $\times$ 97 feet).

We perform the experiments in a testbed of 65 MicaZ motes. Figure 5 shows the layout of the testbed. The sensor nodes are deployed in 25 rooms, including offices, labs, server rooms, and corridors, covering an area of 152.5 $\times$ 97 square feet. We equip each node with an Ethernet programming board, which provides remote access to the node. We only use the programming boards to gather evaluation results from the nodes; they do not interfere with the radio communication between sensor nodes at all. We set the transmission power level of the radio module (CC2420) as -3dBm to increase the hop distance of the network.

Similar to Deluge, we need to configure a number of pa-

rameters for Seluge before code dissemination. We divide each code page into 48 packets, as the default setting in Deluge. To integrate the security mechanisms and the Deluge propagation mechanisms, we have to make certain changes to some Deluge parameters. Deluge uses a 2ms gap between two data packet transmissions. However, a SHA-1 hash verification operation takes about 15ms. Thus, we increase the transmission gap from 2ms to 17ms to accommodate this time requirement. Moreover, we increase the SNACK packet delay from 256ms to 1 second, so that a requesting node gives the sender enough time to transmit all the requested packets. Due to the dependency between the SNACK delay and advertisement delay, we also change the lower bound of the advertisement period to 2 seconds. The upper bound of the advertisement period remains the same default value of 60 seconds as in Deluge.

In these experiments, we use two different packet payload sizes, 102 bytes and 62 bytes, to examine the performance in different situations. (Note that the maximum payload size in IEEE 802.15.4 [15] is 102 bytes.) To investigate and compare the impact of disseminated code size on performance, we use four different code image sizes: 10K bytes, 20K bytes, 30K bytes, and 40K bytes. In each experiment, we inject a new code image at the star-shaped node located at the bottom-right corner in Figure 5. For each test case, we perform the same experiment 20 times and take an average over them.

6.2.1 Propagation Delay

Figure 6 shows the propagation delays of these schemes in the experiments. As the code image size increases, the propagation delays of all schemes increase almost linearly. Since the number of packets required for a given code image increases as the packet payload size decreases, for all approaches, the propagation delays for 62 bytes payload size are longer than those for 102 bytes payload size.

Let us first compare the propagation delays in Seluge and the Colorado approach. For all the code image sizes, the propagation delays in Seluge are much less than those in the Colorado approach, and the gap between them becomes larger as the code image size increases. Among all the experiments, the average propagation delay of the Colorado approach is 63% longer than that of Seluge. When the packet payload size is 62 bytes, it takes the Colorado approach 30% to 82% longer time than Seluge to disseminate a code image. In the worst case, when the code image size is 20K bytes, the delay of the Colorado approach is about 82% longer than that of Seluge. Similarly, when the packet payload size is 102 bytes, it takes the Colorado approach 51% to 73% more time to finish disseminating a code image, where the worst case (i.e., 73%) happens when the code image size is 30K bytes.

As we explained earlier, the main reason for this performance difference is that the Colorado approach propagates

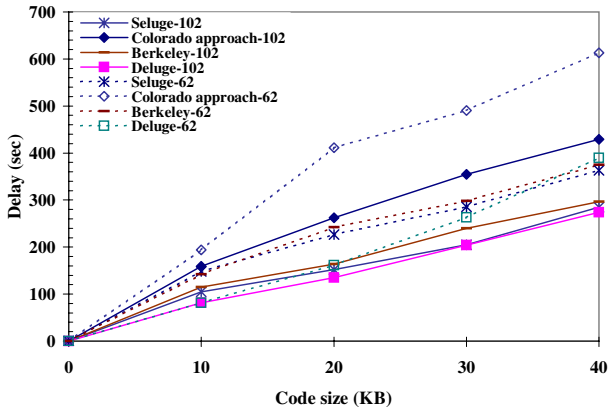

Figure 6. Propagation delay. (The Berkeley approach does not protect maintenance packets. When this mechanism was disabled in Seluge, the propagation delay was reduced by 30 – 146 seconds.)

each code page and the corresponding per-page Merkle hash tree in a level-by-level fashion. This approach increases the interaction between a sending node and its receivers, and disrupts the page-by-page propagation in Deluge. In contrast, Seluge integrates the authentication and DoS-resistance mechanisms seamlessly with the Deluge page-by-page propagation and suppression mechanisms.

Let us now compare Seluge with Deluge. In all the experiments, Seluge introduces on average 21% longer propagation time than Deluge. When the packet payload size is 62 bytes, it takes Seluge 9% to 82% longer time than Deluge. In the case, when the code image size is 40K bytes, the delay of Seluge is even about 7% shorter than that of Deluge. When the packet payload size is 102 bytes, the propagation delay of Seluge is 1% to 29% longer than Deluge. The worst case scenarios in both packet payload sizes happen when the code image size is 10K bytes.

The additional delay introduced by Seluge is due to the propagation and verification of the signature packet, the dissemination of the (additional) hash packets, and the increase of the number of the data packets due to the inclusion of hash images. Nevertheless, as shown in Figure 6, the additional propagation delay introduced by Seluge is much smaller than that by the Colorado approach.

The (revised) Berkeley approach introduces slightly longer delay than Seluge. However, the Berkeley approach does not provide authentication of Deluge maintenance packets. To get better understanding of the performance difference between Seluge and the Berkeley approach, we run another set of experiments using Seluge with cluster key based local authentication disabled. Our results indicate that without local authentication of maintenance packets, the propagation delay in Seluge can be reduced by 56,

106, 112, and 146 seconds for code sizes 10K, 20K, 30K, and 40K when the packet size is 62 bytes, and by 30, 55, 77, 132 seconds for code sizes 10K, 20K, 30K, and 40K when the packet size is 102 bytes.

These experimental results demonstrate that Seluge introduces much less propagation delay into code dissemination than the Colorado approach, in addition to the stronger security properties.

6.2.2 Communication Overhead

Figure 7 shows the communication overheads of all these schemes, which are measured as the total number of packets transmitted by all the nodes in each test case. For the communication overheads of Seluge and the Colorado approach, we consider SNACK packets, hash packets (called index packets in the Colorado approach [8]), and data packets because those three types of packets are additionally required for a dissemination. Likewise, we consider SNACK and data packets for the communication overhead of Deluge and the Berkeley approach. As in the evaluation results for propagation delays, for all approaches, the communication overheads increase approximately linearly as the code image size grows, and the communication overheads for 62 bytes payload size are larger than those for 102 bytes payload size. In all the experiments, the Berkeley approach has the largest communication overhead, the Colorado approach is ranked the second, and Seluge has slightly larger communication overhead than Deluge. In particular, the Berkeley approach has more than twice as much overhead than any other approach.

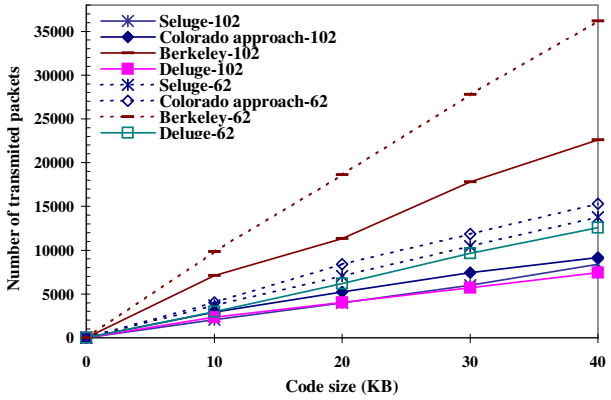

Figure 7. Communication overhead.

6.2.3 Propagation on Individual Nodes

We also investigate how code pages are propagated on individual nodes to get more insights. In the following, we select two nodes in the testbed, which are marked as n_1 and n_2 in Figure 5, to see how they receive code pages over time. (We select these two nodes to present, because n_1 is close to the source and n_2 is far away from the source. They are expected to have different situations during code dissemination.) In the following, we show the dynamic propagation features on these two nodes, using the test case where we inject a code image of 30K bytes with 102 bytes payload size.

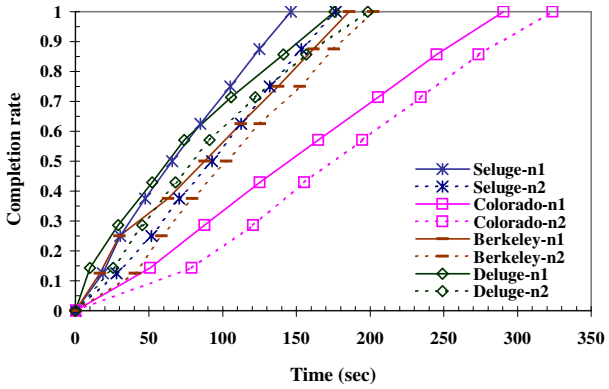

Figure 8. Dissemination progress over time on selected nodes.

Figure 8 shows the time points when n_1 or n_2 finishes receiving every page of the code image under all approaches. The x-axis represents the completion time for a page, and the y-axis represents the ratio of the number of completed pages to the total number of pages in a code image. As time goes on, n_1 and n_2 gradually complete the receiving of the code image. Due to the effect of spatial multiplexing, n_2 receives some pages of the code image before n_1 finishes to receive all the pages of the code image.

Figure 8 confirms at individual node level that Seluge allows much faster propagation than the Colorado approach and the Berkeley approach. Compared to Deluge, Seluge shows very similar dissemination dynamics on both of the nodes. Moreover, on these two nodes, Seluge even surpasses Deluge from certain time points and finally completes the code image earlier than Deluge, though it involves additional security mechanisms such as signature verification and hash packet distribution.

7 Conclusion

In this paper, we presented the design, implementation, and evaluation of Seluge. Besides the efficiency and robustness inherited from Deluge, Seluge provides security protections for code dissemination, including the integrity protection of code images and resistance to the DoS attacks that exploit the code dissemination protocol. Seluge is superior to all previous attempts for secure code dissemination, and is the only one that seamlessly integrates the security mechanisms and the Deluge efficient propagation strategies.

Acknowledgment

We would like to thank Andreas Terzis for shepherding our paper, Prabal Dutta and David Culler for helpful discus-

sions and providing source code for their work in [10], and the anonymous reviewers for their useful comments.

References

[1] Bouncy castle crypto apis. http://www.bouncycastle.org.

[2] MICAz: Wireless measurement system. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf.

[3] TelosB mote platform. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/TelosB_Datasheet.pdf.

[4] TinyOS: An open-source OS for the networked sensor regime. http://www.tinyos.net/.

[5] Certicom Research. Standards for efficient cryptography – SEC 2: Recommended elliptic curve domain parameters. http://www.secg.org/collateral/sec2_final.pdf, September 2000.

[6] H. Chan, A. Perrig, and D. Song. Random key predistribution schemes for sensor networks. In *IEEE Symposium on Research in Security and Privacy*, pages 197–213, 2003.

[7] Crossbow Technology Inc. Mote in-network programming user reference, 2003.

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

[9] W. Du, J. Deng, Y. S. Han, and P. Varshney. A pairwise key pre-distribution scheme for wireless sensor networks. In *Proceedings of 10th ACM Conference on Computer and Communications Security (CCS'03)*, pages 42–51, October 2003.

[10] P. K. Dutta, J. W. Hui, D. C. Chu, and D. E. Culler. Securing the deluge network programming system. In *Proceedings of the Fifth International Conference on Information Processing in Sensor Networks (IPSN '06)*, April 2006.

[11] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation (PLDI '03)*, June 2003.

[12] Y. Hu, A. Perrig, and D. Johnson. Packet leashes: A defense against wormhole attacks in wireless ad hoc networks. In *Proceedings of INFOCOM 2003*, April 2003.

[13] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys '04)*, November 2004.

[14] S. Hyun, P. Ning, A. Liu, and W. Du. Seluge: Secure and dos-resistant code dissemination in wireless sensor networks. Technical Report TR-2007-21, NC State University, Department of Computer Science.

[15] IEEE Std 802.15.4-2003. IEEE standard for information technology – telecommunications and information exchange between systems – local and metropolitan area networks – specific requirements – part 15.4: Wireless medium access control (MAC) and physical layer (PHY) specifications for low-rate wireless personal area networks (LR-WPANs).

[16] S. Kulkarni and L. Wang. MNP: multihop network reprogramming service for sensor networks. In *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS '05)*, pages 7–16, June 2005.

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

[18] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the 1st Symposium on Network System Design and Implementation (NSDI '04)*, March 2004.

[19] A. Liu, P. Kampanakis, and P. Ning. TinyECC: Elliptic curve cryptography for sensor networks (version 0.3). http://discovery.csc.ncsu.edu/software/TinyECC/.

[20] D. Liu and P. Ning. Establishing pairwise keys in distributed sensor networks. In *Proceedings of 10th ACM Conference on Computer and Communications Security (CCS'03)*, pages 52–61, October 2003.

[21] R. Merkle. Protocols for public key cryptosystems. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Apr 1980.

[22] V. Naik, A. Arora, P. Sinha, and H. Zhang. Sprinkler: A reliable and scalable data dissemination service for wireless embedded devices. In *Proceedings IEEE International Real-Time Systems Symposium*, pages 277–286, December 2005.

[23] J. Newsome, R. Shi, D. Song, and A. Perrig. The sybil attack in sensor networks: Analysis and defenses. In *Proceedings of IEEE International Conference on Information Processing in Sensor Networks (IPSN 2004)*, April 2004.

[24] P. Ning, A. Liu, and W. Du. Mitigating DoS attacks against broadcast authentication in wireless sensor networks. *ACM Transactions on Sensor Networks*, 4(1), February 2008.

[25] A. Perrig, R. Canetti, D. Song, and D. Tygar. Efficient authentication and signing of multicast streams over lossy channels. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, May 2000.

[26] A. Perrig, R. Canetti, D. Song, and D. Tygar. Efficient and secure source authentication for multicast. In *Proceedings of Network and Distributed System Security Symposium*, February 2001.

[27] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proceedings of the 2nd ACM International Conference on Wireless Sensor Networks and Applications (WSNA '03)*, pages 60–67, September 2003.

[28] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30, UCLA, Center for Embedded Networked Computing, November 2003.