# Scatter – Secure Code Authentication for Efficient Reprogramming in Wireless Sensor Networks

**Abstract:** Currently proposed solutions to secure code dissemination in wireless sensor networks (WSNs) involve the use of expensive public–key digital signatures. In this work we present Scatter, a secure code dissemination scheme that enables sensor nodes to authenticate the program image efficiently. To achieve this, we use a scheme that offers source authentication in the group setting like a public–key signature scheme, but with signature and verification times much closer to those of a MAC. In this way, Scatter avoids the use of Elliptic Key Cryptography and manages to surpass all previous attempts for secure code dissemination in terms of energy consumption, memory and time efficiency. Besides the design and theoretical analysis of the solution, we also report the experimental evaluation of Scatter in two different hardware platforms, Mica2 and MicaZ, which proves its efficiency in practice.

**Keywords:** sensor networks; code dissemination; network reprogramming; security; authentication.

## 1 Introduction

Traditional methods of programming sensor nodes with a new binary require physical access to the nodes themselves: the application is developed in a PC and the resulting program image is loaded to the node through the parallel or the serial port. The same process has to be repeated for all the nodes, which then have to be re-deployed back to the field. Not only this solution does not scale to large number of geographically distributed nodes, but in many cases it might not be even possible to access the nodes. For these reasons, *code dissemination* have emerged as a solution to propagate the new program image remotely, over the wireless link to the entire network, in a multi-hop fashion. Each sensor node propagates any received updates to its immediate neighbors, which in turn do the same until the new binary reaches all the nodes. In-system programmability allow nodes to reprogram themselves and start operating with the updated code.

A number of code dissemination protocols suitable for sensor networks have been proposed (see Wang et al. (2006) for a comprehensive survey). They focus mainly on the reliability and the latency of the dissemination, but from a security point of view all of them assume that the nodes will behave in a legitimate way, meaning that no node can be compromised and controlled by an adversary. As a result nodes do not care about authenticating the source of the program. This fact allows an attacker to approach the deployment site and disseminate malicious or corrupted code in the network, reprogramming all the nodes at will. Thus, in the same way that a code dissemination protocol can help the network administrator with the program update procedure, it can also provide an easy way for an attacker to compromise the whole network by installing malicious code.

In this work we present Scatter, a scheme that secures code dissemination and allows sensor nodes to efficiently verify that the new code originates from a trusted source, namely the base station. With this security feature added, an attacker could not authenticate herself to the network, and therefore the nodes will reject malicious updates.

There have been several proposals to secure code dissemination for wireless sensor networks so far (Lanigan et al., 2006; Dutta et al., 2006; Deng et al., 2006; Shaheen et al., 2007; Liu et al., 2008; Tan et al., 2009). All of these approaches have in common that they employ either RSA or elliptic curve cryptography (ECC) to solve the problem and they concentrate on providing resilience against DOS attacks. So, they differ mainly in the granularity of data considered in the hash chains they employ.

The expensive signature overhead of public-key based solutions introduces a big delay in the protocol, uses a lot of memory resources on the nodes and consumes a lot of energy for complex computations. In this paper we show that the characteristics of code dissemination allow the use of $r$-time signature schemes, which decrease dramatically the signing and verification complexity, while maintaining the attractive characteristics of a public-key signature scheme. Therefore, in this paper we propose a construction of an $r$-time signature scheme for sensor networks and we introduce Scatter, a secure version of Deluge that integrates our solution.

In the remaining of this paper, we start in Section 2 by defining the concept of network reprogramming and giving some details about Deluge, a widely used protocol for this operation. In Section 3, we present the requirements and desirable features that a secure code dissemination protocol should exhibit, and in Section 4 we elaborate on the main approaches that exist and which one Scatter follows to be more efficient.

In Section 5, we present a $r$-time signature scheme appropriate for sensor networks, which is used by Scatter to authenticate program images. Section 6 describes the implementation and experimental evaluation of Scatter in different sensor platforms. Finally, Section 7 concludes this work.

## 2 Code Dissemination Protocols in WSNs

It is important to give some details about code dissemination protocols in sensor networks that will help us understand better the process of securing this procedure. In general, code dissemination is achieved by the following steps (see Figure 1):

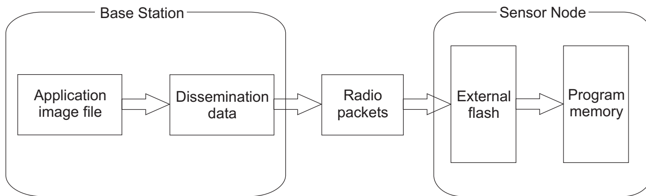



**Figure 1** Code dissemination process.

1. The base station reads the new application binary code and breaks it into packets to disseminate.
2. The base station sends the packets to the sensor nodes within communication range.
3. The nodes store the packets in the external flash memory after receiving them. They request retransmission of any missing packets.
4. The nodes forward the packets to any of their neighbors that have not received them, until all nodes get the new code.
5. After all packets have been received, the code lies in the external flash memory of the nodes. The nodes verify the program image and call the boot loader to transfer the program code to the program memory. Then the boot loader restarts the system and the new program begins execution.

Since Deluge (Hui and Culler, 2004) is one of the most commonly used protocols for code dissemination and has been included in recent TinyOS distributions (Hill et al., 2000), we use it as our base example in this paper.

Deluge propagates a program image by dividing it first into fixed-size pages and then uses a demand-response protocol to disseminate them in the network. As soon as a node receives a page, it makes it available to any of its neighbors that also need it. At the same time it sends a request to the sender in order to receive subsequent pages. In this way, Deluge supports a sort of *pipelining*: already received pages are forwarded further to the rest of the nodes while the program image is not yet complete and new pages keep coming in.

The integrity of each page is verified by using a 16-bit cyclic redundancy check (CRC) across both packets and pages. So, if a packet gets dropped or corrupted, the node requests from the sender to send it again until the page is completely and correctly received. This means that any authentication scheme added on top of Deluge does not need to be robust on packet loss. It can safely assume that packets always reach their destination. Also, note that since Deluge does not start receiving the next page, unless the previous one is completed, an authenticated broadcast protocol does not need to deal with out-of-order delivery of pages (even though packets may indeed arrive out-of-order).

## 3 Problem Definition and our Contribution

Our goal is to provide an efficient source authentication mechanism for broadcasting a program image from the base station to the sensor network. While the authentication mechanism should still allow efficient dissemination procedures, such as pipelining, it should also block malicious updates as early as possible.

The most natural solution for authenticated broadcasts is asymmetric cryptography, where messages are signed with a key known only to the sender. Everybody can verify the authenticity of the messages by using the corresponding public key, but no one can produce legitimate signed messages without the secret key.

However, public key schemes like RSA, have been long thought to be impractical for the limited computational, memory and energy resources of sensor nodes. Gura et al. (2004) have shown that Elliptic Curve Cryptography (ECC) can provide substantial performance gains over RSA for constrained hardware. For example, TinyECC (Liu and Ning, 2008) provides a freely available ECC implementation for TinyOS, which runs 12 to 16 seconds to verify a signature on MicaZ motes.

Besides computational overhead, public-key cryptography solutions also require a large percentage of the available memory resources. For example, TinyECC takes over 19.3 KB out of the 48 KB available in ROM of a standard MicaZ node (Liu and Ning, 2008).

In this paper we provide an efficient authentication scheme for disseminating a finite stream of data based on *symmetric* cryptography primitives, while maintaining at the same time the properties of asymmetric cryptography. In this way, we can avoid the use of ECC or RSA, and reach a much faster and energy efficient solution for authenticating the program image on the motes.

The solution satisfies the following requirements:

1. **Low computational cost**. Since our solutions requires only computations of a collision-resistant

hash function on the motes, it requires low computational overhead.

2. **Low verification time**. The rate at which a code segment is transmitted to the receiver is not reduced significantly by the authentication scheme.

3. **Low communication overhead**. The signature transmitted with data and the embedded hash values constitute a small percentage of the total bytes, imposing a low communication overhead.

4. **Low memory requirements**. The memory footprint of our code stored in ROM as well as any cryptographic material buffered in RAM are very small, given the mote's limited memory resources.

Moreover, since we are providing an authenticated broadcast scheme, we assure the following security requirements:

1. **Source authentication**. A mote is able to verify that a code update originates from a trusted source, i.e., the base station. This means that an attacker is not able to send malicious code in the network and reprogram the nodes.

2. **Node-compromise resilience**. In case an attacker compromises a node and read its cryptographic material, she is not able to reprogram any other non-compromised node with malicious code.

3. **DOS-attack resilience**. In case an attacker is trying to transmit malicious code to the network, any receiving node is able to realize this as soon as possible and stop receiving it or forwarding it to other nodes. For this purpose, Scatter realizes an incremental security verification mechanism that allows the dynamic loading of received pages.

## 4 Design Approaches and Related Work

As we said, asymmetric cryptography is the most natural solution to authenticated broadcasts. So, one approach would be to compute a digital signature for the whole program image. However, that would require the nodes to receive and buffer the entire image in order to verify the signature, which is clearly not possible for sensor nodes, given their limited memory resources. Moreover, the receiver needs to be able to "consume" each page it receives, i.e., send it to its own neighbors (pipelining) as well as store it to the external flash memory.

If we view program images as digital streams of data, we can apply the solution by Gennaro and Rohatgi (2001) in signing a program image. What they proposed is to divide the stream into blocks and embed some authentication information in each block. In particular, their idea is to embed in each block a hash of the following block. This way the sender needs to sign just the first hash value and then the properties of this signature will propagate to the rest of the stream through the "chaining" technique.

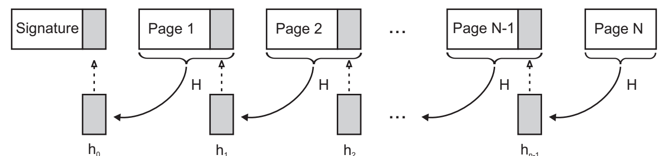



**Figure 2** Hash chain construction for the pages of a program image.

In our case, given a program image divided into $N$ fixed-size pages

$$P_1, P_2, \ldots, P_N$$

and a collision-resistant hash function $H$, we construct the hash chain

$$h_i = H(P_{i+1}||h_{i+1}), \quad i = 0 \ldots N-2$$

and we attach each hash value $h_i$ to page $P_i$ (see Figure 2), where "||" denotes concatenation. For the last hash value we set

$$h_{N-1} = H(P_N).$$

Now, we can sign only the hash chain commitment, $h_0$, and the nodes that receive and verify that signature will be able to authenticate all the pages by just computing their hash values and compare them with those transmitted.

### *4.1 DOS-attack Resilience*

One important design choice is whether to construct the one-way hash chain over the *pages* of the new program image or the *packets* that compose the pages. In Figure 2, we showed how the hash chain is applied to larger blocks, i.e. the pages, but alternatively the same method can be followed to apply the hash chain at a bigger granularity, i.e. the packets.

In the former case where the hash is computed over the whole page, the drawback is that the nodes need to receive the page before they are able to verify its hash. Since a page is usually composed of hundreds of packets, this gives an advantage to an attacker to launch a DOS attack, by sending a few malicious packets. After receiving the page, a node will realize that the hash verification fails and it will request the page again, as it will not know which specific packets caused this fail.

The alternative approach of computing the hash values for each packet is followed by both Dutta et al. (2006) and Deng et al. (2006). Since these schemes authenticate each packet separately, they can stop receiving them as soon as they find the first packet that failed to authenticate, saving the energy to receive the rest of the packets. However, this comes at a big price, as it increases the communication and computation

overhead of the scheme, cancelling the benefits that we got from resisting the DOS attack.

In particular, Deng et al. (2006) construct signed hash trees (similar to a Merkle tree) based on the hashes of each packet in the program image and they transmit these trees before the actual data. This increases the overhead of packets sent and received by the motes by about 28%. Moreover, due to memory constrains in the motes, these values need to be stored and loaded from the EEPROM each time a packet arrives, which is a very energy consuming operation for the motes (Stathopoulos et al., 2003).

Similarly, Dutta et al. (2006), compute the hash of each packet and place it in the previous packet. This increases the overall data that have to be transmitted besides the signature, compared to constructing and transmitting a hash value for each page. It also increases the computations a mote has to perform in order to verify all these hash values, besides the RSA signature verification that this scheme uses.

Another problem is that schemes, which try to provide protection against DOS attacks, do so only by considering the data packets, i.e. the packets that bare the new program image. However, code dissemination protocols are more complex than that. In order to provide some desirable properties, such as flexibility, efficiency and reliability of the code propagation, they also use some types of maintenance packets, like request (Req), advertisement (Adv) and acknowledgment (Ack) packets. It is easy for an adversary to exploit this kind of packets in order to launch a DOS attack, bypassing the protection measures of the above schemes. Zhang et al. recently showed five different types of DOS attacks made by malicious nodes exploiting maintenance packets.

So, overall, applying a page-level hashing offers better performance but less resilience under a DOS attack, while applying a packet-level hashing imposes high overhead in all cases, in order to offer partial resilience under this attack. In our scheme we have chosen the former and simpler approach of constructing the hash chain on the pages of the program image. This is also followed by other schemes, like Sluice (Lanigan et al., 2006). In this work we concentrate on the signature construction and verification, rather on the hash chain construction. Let us note that applying a different hash granularity is possible in Scatter, without affecting the signature scheme proposed.

### *4.2 Signature Scheme*

The most energy consuming operation in the process of secure code dissemination is the authentication of $h_0$, which is signed and released before the transmission of the pages. Therefore, choosing which security scheme to apply for this operation is important in the overall protocol's performance. As we said, all currently proposed solutions use a public key scheme like RSA or Elliptic Curve Cryptography (ECC) for signing the hash commitment. In the next section we are going to show a much more efficient way that can be applied for the case of code dissemination protocols.

We base our analysis on the fact that real world software updates in sensor networks do not constitute an everyday operation but rather they are performed occasionally. Therefore, we do not need to authenticate an unlimited number of broadcasts. We only need to be able to do so for a sufficiently large number of times. This fact allows us to use *r-time signature schemes*, which exhibit fast verification times. We describe analytically this solution in the next section. We first presented this scheme in 2006 (citation withheld), while later Ugus et al. (2009) independently proposed a similar solution.

## 5 Scatter's $r$-time Signature Scheme

An $r$-time signature scheme is similar to a public-key scheme in that it can be used to sign messages that can be verified using publicly known information. These $r$-time signatures decrease dramatically the signing and verification time compared to public-key signatures, however, one can only sign up to $r$ messages with a given key pair. After that, the security level drops below acceptable limits and a new key pair must be generated. But regarding code dissemination, if we can efficiently sign and verify, for example, $r = 32$ new program images before we need to redistribute a new public key, we have a tradeoff that makes this an attractive solution.

Recently, some signature schemes were proposed that seem attractive for sensor networks. For example, Reyzin and Reyzin (2002) introduced HORS, an $r$-time signature scheme with efficient signature and verification times. This scheme was further improved by Pieprzyk et al. (2003). Both of these $r$-times signature schemes can sign several messages with the same key with reasonable security before they can get compromised. However there are some drawbacks that prevent us from applying those schemes to sensor networks. The main one is the size of the public/secret key pair and the size of the generated signatures. For example, the HORS scheme uses a public key size of 20 KB for $r = 4$, which is not suitable for use in sensor networks. It also grows unacceptably high if we want to sign more messages (bigger $r$) and keep security at an acceptable level.

To overcome these drawbacks, we propose an $r$-time signature scheme, which is optimized for use in sensor networks. This scheme manages to drop the signature and public key sizes to values that are attractive for use in sensor nodes, and also reduces the signature verification time to that of a few hash operations.

Throughout our analysis we follow the same notation as in HORS. Table 1 summarizes the main parameters and their definitions. In what follows, we formally define our scheme.

Let $F$ be an $l$-bit one-way function. First we need to produce the secret and public key pair. To do this the signer applies the following steps:

**Table 1** Notations.

| *Notation* | *Definition* |
|---|---|
| $PK$ | Public Key |
| $S$ | Signature |
| $\|h\|$ | Size of hash values (bits) |
| $l$ | Size of secret values (bits) |
| $t$ | Total number of secret values |
| $k$ | Number of secret values in signature |
| $T$ | Number of public values |
| $r$ | $r$-subset-resilient |
| $\Sigma$ | Provided security level |

**Secret Key** Generate $t$ random $l$-bit quantities for the secret key:

$$SK = (s_1, \ldots, s_t).$$

**Public key** Compute the public key as follows: Generate $t$ hash values

$$(u_1, \ldots, u_t),$$

where

$$u_1 = F(s_1), \ldots, u_t = F(s_t).$$

Separate these values into $T$ groups, each with $t/T$ values. Use these values as leaves to construct $T$ Merkle trees, as shown in Figure 3. The roots of the trees constitute the public key $PK$ of our scheme.

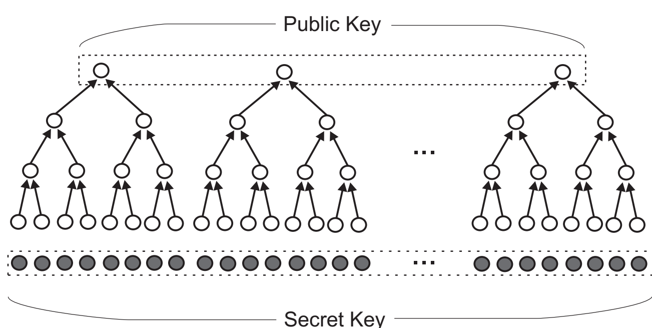



**Figure 3** Public key construction using Merkle trees.

A Merkle hash tree (Merkle, 1988) is a complete binary tree where each node is associated with a value, such that the value of each parent node is the hash function on the values of its children:

$$v(parent) = H(v(left)||v(right))$$

where the function $v$ here stands for the value of a node and $H$ for a hash function.

Using the Merkle trees we have achieved a public key size of a few hash values, i.e., the roots of the Merkle trees. These values need to be passed to all sensor nodes in an authenticated way. This can be done for example during initialization of sensor nodes.

Next we show how any message $m$ can be signed using this secret and public key pair. The procedure for this is described in the following steps (see also Figure 4):

1. Use a cryptographic hash function $H$ to convert the message to a fixed length output. Split the output into $k$ substrings of length $\log_2 t$ each.

2. Interpret each substring as integer in the range $[1 \ldots t]$. Use these integers $i_1, i_2, \ldots, i_k$ to select a subset $\sigma$ of $k$ values out of the set of secret values $SK = (s_1, \ldots, s_t)$.

3. The signature of the message $m$ is made up by the selected secret values along with their corresponding authentication paths.

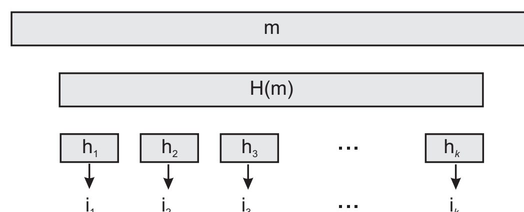



**Figure 4** Producing $k$ indices of secret values from a message $m$ of arbitrary length.

By *authentication path* of a secret value we mean the values of all the nodes that are siblings of nodes on the path between the leaf that represents the secret value and the root of the corresponding Merkle tree, as shown in Figure 5. Note that for each message that we sign, a part of the secret key is leaked out. That's why this process cannot be repeated *ad infinitum*.

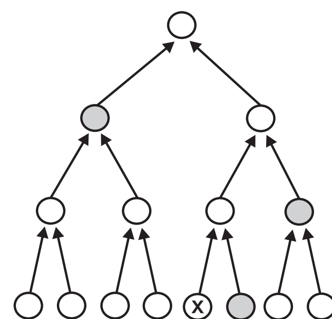



**Figure 5** Authentication path corresponding to a secrete value (marked leaf). The nodes, which are shaded grey, constitute the path.

A node that has received the message $m$ and wants to verify its signature, recomputes the hash value of the message, reproduces the same indices and picks the corresponding values of the set $PK$. Remember that the node has the public values $PK$ (roots of the Merkle trees), but not the Merkle trees. Then it evaluates each authentication path of the signature to reproduce

the root of the Merkle tree and compare it with the corresponding member of the public key $PK$ that it has in its memory. The signature is accepted, if this is true for all $k$ values. The detailed description of the algorithm is shown in Figure 6.

Let us emphasize that the verification of the signature at the sensor nodes requires only hash and comparison operations. Both of these operations can be performed very fast and efficiently in the nodes. For example, the time to hash one block using MD5 in a sensor node is 2.58 ms, as we will see in Section 6.2. The number of hash operations that will be needed depends on the number and the size of the authentication paths (or else, the size of the signature), which also determines the security level of the scheme.

The $r$-time signatures scheme we described can be tuned by setting various parameters, like the number of secret values $t$, the number of Merkle trees $T$, or the number of $k$ parts that we split the hash of the message $m$ (i.e, number of secret values that we release in the signature). The values that we choose for these parameters will determine the *signature size* and the *public key size*, two quantities that are important for the efficiency of the scheme, but also its *security level.* We study how these quantities are affected by the parameters of our scheme in the following sections.

### *5.1 Public Key and Signature Size*

There is a trade-off between the public key size and the signature size. The public key stored in each sensor node is given by the hash values residing at the roots of the trees. The more the number of the trees, the bigger the public key becomes but the smaller the signature size becomes. To see why, notice that the signature size depends on the length of the authentication paths, which are ultimately related to the height of the Merkle trees. More trees mean less secret values per tree and hence smaller height.

Ideally, we would like both of these sizes to be as small as possible. The signature is transmitted over the radio to be received and verified by the motes, so the smaller it is, the less energy and time will be needed for these operations. Similarly, the public key is stored in the memory (RAM) of the motes, to be used for the signature verification procedure. So, there is a limit on how large it can become. To calculate the formulas that give these two quantities, let $T$ denote the number of trees. Hence the public key size is simply

$$|PK| = |h|T, \qquad (1)$$

since every root contains a hash value of its children. We use the notation $|h|$ for the output of the hash function $H$ in bits. For example, $|h|$ can be equal to 128 bits in the case of MD5 or 160 bits in the case of SHA–1.

If we have $T$ Merkle trees and $t$ secret values, there can be at most $t/T$ values stored at the leaves of each tree. Thus the height of each tree (and the length of each authentication path) is simply $\log_2(t/T)$. The signature $S$ consists of $k$ such authentication paths, where each path is a sequence of hash values of $|h|$ bits. Thus the signature size is given by

$$|S| = |h|(k \log_2 \frac{t}{T}). \qquad (2)$$

This equation can be simplified further if we recall how the $k$ secret values are selected. The message $m$ to be authenticated is first hashed to obtain $H(m)$, a value that is $|h|$ bits long. Then these $|h|$ bits are broken into $k$ parts, where each part references one of the secret values. Thus the number of secret values $t$ must be equal to $2^{|h|/k}$, or equivalently

$$|h| = k \log_2 t. \qquad (3)$$

Combining Equations (3) and (2), we find that the signature size is given by

$$|S| = |h|(|h| - k \log_2 T). \qquad (4)$$

### *5.2 Security Level*

Next we calculate the security level of the scheme, since it is also affected by the values we choose for the parameters of the scheme. Let $r$ be equal to the number of messages that we allow to be signed with the current instance of the secret key. For an analysis (see also the work by Reyzin and Reyzin (2002)) we assume that the hash function $H$ behaves like a random oracle and that an adversary has obtained the signatures of $r$ messages using the same setting of secret/public key. Then the probability that an adversary can forge a message is simply the probability that after $rk$ values of the secret key have been released, $k$ elements are chosen at random that form a subset of the $rk$ values. The probability of this happening is $(rk/t)^k$. If we denote by $\Sigma$ the attainable security level in bits, by equating the previous probability to $2^{-\Sigma}$, we see that $\Sigma$ is given by

$$\Sigma = k(\log_2 t - \log_2 k - \log_2 r). \qquad (5)$$

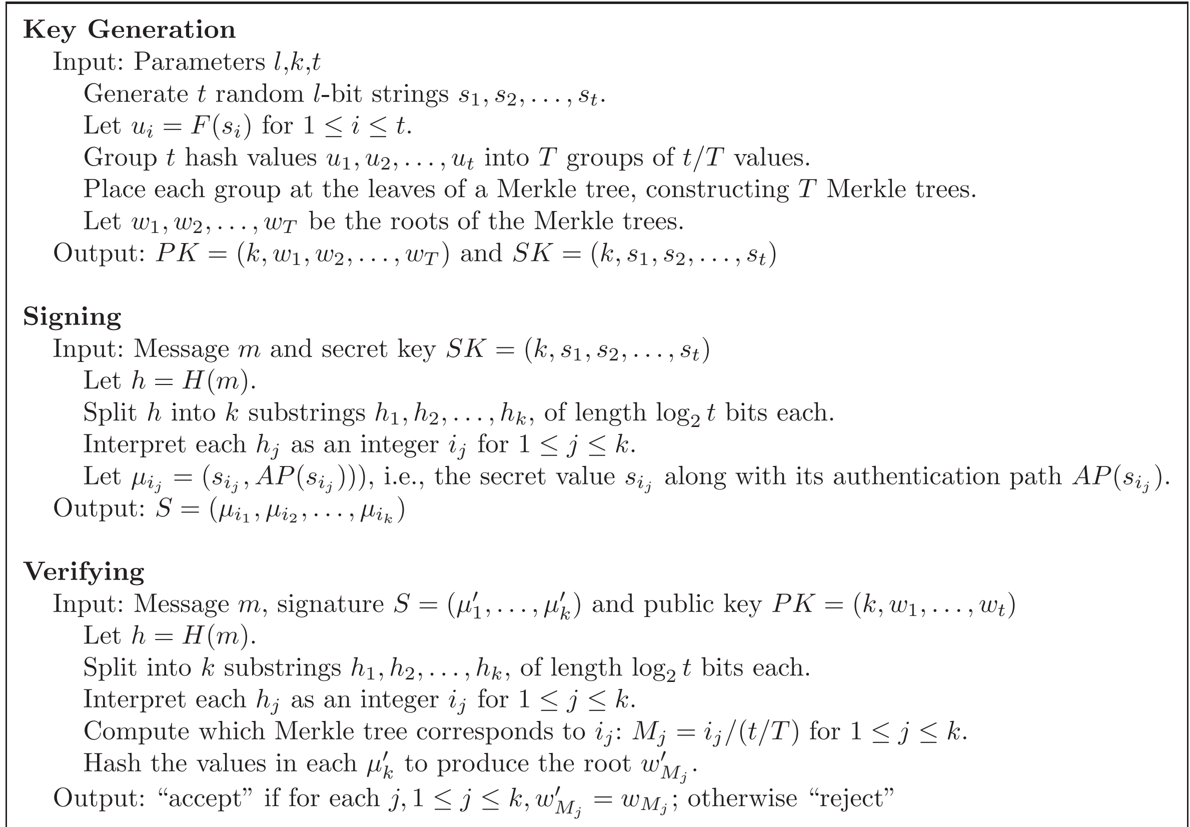
and by using Eq. (3), we get

$$\Sigma = k(|h|/k - log_2 k - log_2 r). \qquad (6)$$

### *5.3 Signature Verification Time*

To be able to decide on particular values for $r$, $k$ and $T$, we would also like to have an estimation of how these parameter affect the verification time of the signature. We implemented the $r$-time signature scheme in TinyOS and measured the verification times for various values of $T$ on the Mica2 platform. Figure 7 shows the results.

For the signature verification in Figure 7, the time needed is directly dependent on the signature size. The parameter determining the signature size is the number of Merkle trees $T$. As we build more trees on the secret values, their height gets smaller, and therefore the signature size is reduced. Consequently, the verification time at the mote's side is also reduced.



**Key Generation**

Input: Parameters $l$,$k$,$t$

Generate $t$ random $l$-bit strings $s_1, s_2, \ldots, s_t$.

Let $u_i = F(s_i)$ for $1 \le i \le t$.

Group $t$ hash values $u_1, u_2, \ldots, u_t$ into $T$ groups of $t/T$ values.

Place each group at the leaves of a Merkle tree, constructing $T$ Merkle trees.

Let $w_1, w_2, \ldots, w_T$ be the roots of the Merkle trees.

Output: $PK = (k, w_1, w_2, \ldots, w_T)$ and $SK = (k, s_1, s_2, \ldots, s_t)$

**Signing**

Input: Message $m$ and secret key $SK = (k, s_1, s_2, \ldots, s_t)$

Let $h = H(m)$.

Split $h$ into $k$ substrings $h_1, h_2, \ldots, h_k$, of length $\log_2 t$ bits each.

Interpret each $h_j$ as an integer $i_j$ for $1 \le j \le k$.

Let $\mu_{i_j} = (s_{i_j}, AP(s_{i_j})))$, i.e., the secret value $s_{i_j}$ along with its authentication path $AP(s_{i_j})$.

Output: $S = (\mu_{i_1}, \mu_{i_2}, \ldots, \mu_{i_k})$

**Verifying**

Input: Message $m$, signature $S = (\mu'_1, \ldots, \mu'_k)$ and public key $PK = (k, w_1, \ldots, w_t)$

Let $h = H(m)$.

Split into $k$ substrings $h_1, h_2, \ldots, h_k$, of length $\log_2 t$ bits each.

Interpret each $h_j$ as an integer $i_j$ for $1 \le j \le k$.

Compute which Merkle tree corresponds to $i_j$: $M_j = i_j/(t/T)$ for $1 \le j \le k$.

Hash the values in each $\mu'_k$ to produce the root $w'_{M_j}$.

Output: "accept" if for each $j, 1 \le j \le k, w'_{M_j} = w_{M_j}$; otherwise "reject"

**Figure 6** The $r$-time signature scheme. $F$ is a one-way function and $H$ is a hash function.

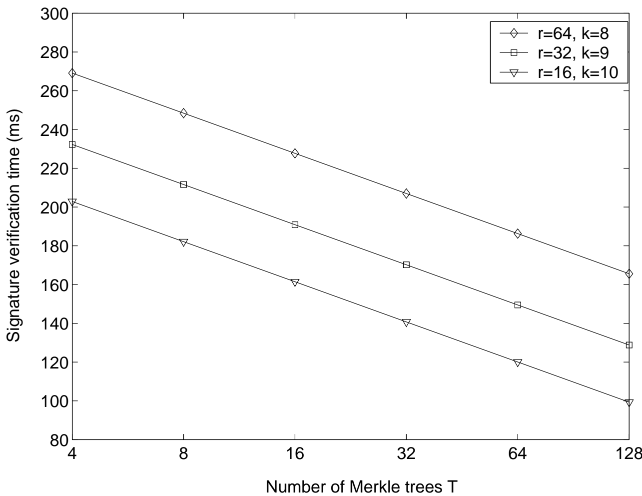



**Figure 7** Signature verification time as a function of $T$.

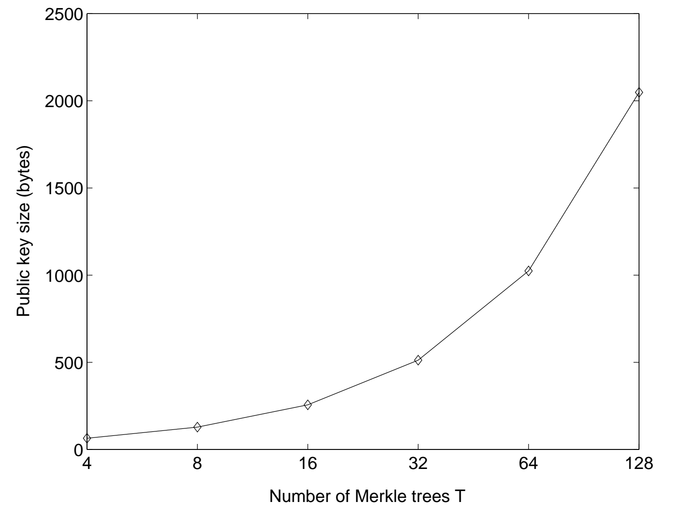



**Figure 8** Public key size as a function of $T$.

Figure 8 is a graphical representation of Eq. (1), using MD5 to produce the hash values, which means $|h| = 128$. Let us keep the public key size equal to approximately 1 KByte, so that it fits well in the memory of a typical Mica2 node (approximately 4 KB of RAM). So, we set $T = 64$. Figure 7 shows the verification time of the signature as a function of the number of Merkle trees $T$, for different values of $r$ (number of images that can be signed using the same keys) and $k$ (number of substrings that we split the hash value of the message into). Let us say we want to sign $r = 64$ messages before we need to update the keys. What would be a good value for $k$ to choose? Figures 9 and 10, show a graphical representation of Equations (4) and (6) respectively, for $T = 64$ and $|h| = 128$.

Observing Figures 9 and 10, a good value for $k$ could be $k = 8$. Then, for $r = 64$ we would get a security level of around 60 bits against passive adversaries, and a signature size of 1280 bits. The verification time of this signature, from Figure 7 would be equal to 186.3 ms.

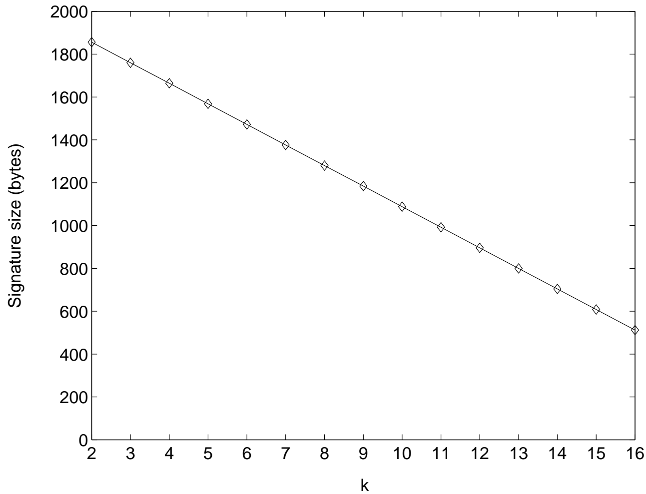



**Figure 9** Signature size as a function of $k$.

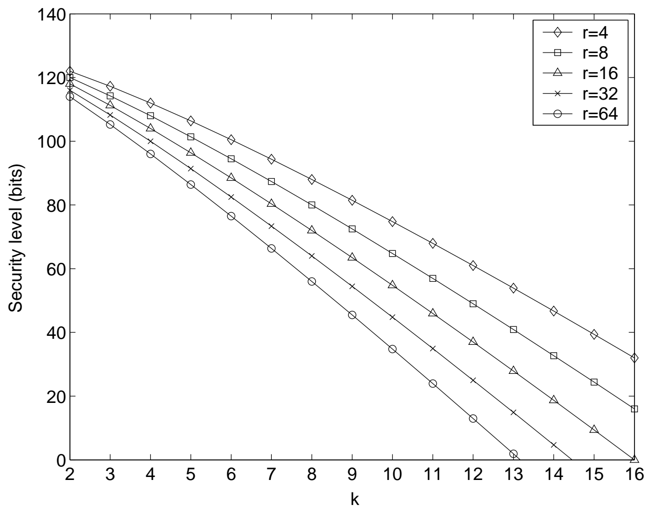



**Figure 10** Security level as a function of $k$.

Notice that we used standard implementations of hash functions, so we believe that this signature verification time can be improved even further using optimized code for the particular hardware of the motes.

We can see now the advantage of using this $r$-time signature scheme to authenticate a message in sensor networks. If we had chosen to use Elliptic Curve Cryptography, the signature verification would be much larger. For example, TinyECC (Liu and Ning, 2008) provides ECDSA (American National Standards Institute, 1998) verification functionality on the sensor nodes that takes between 12 and 16 seconds, which is highly inefficient compared to the verification times of the order of milliseconds, given in Figure 7.

To get a complete picture of the performance of a code dissemination protocol like Deluge that uses this $r$-time signature scheme to authenticate the broadcasted program images, we need first to show how the integration of the two schemes is done, and then measure the overall time to download and authenticate a complete image to a sensor node. We do this in the next section, presenting also some implementation details.

## 6 Implementation and Experimental Evaluation

### *6.1 Implementation*

We implement Scatter as an extension to Deluge in TinyOS distribution. Our implementation has two parts, one for the base station side and one for the sensor side. The former extends the Deluge Java tools to construct and inject new code dissemination packets into the sensor network. The latter is written in nesC (Gay et al., 2003) and runs on regular sensor nodes.

We add two main functionalities in the Java tools on the base station side: First, computation of the hash values of the image pages and second, signing of $h_0$, the public commitment of the hash chain we have built in order to authenticate the sequence of pages. In Section 5, we described how we can create a signature for a message $m$ using an $r$-time signature scheme. We apply this scheme to sign $h_0$.

Figure 11 shows in detail the process of preparing a secure program image at the PC side for dissemination in the network. A program image that Deluge transmits to the nodes includes some metadata associated with it, information about the length of the image in bytes, and the image itself. Deluge first partitions the image into $N$ pages, given as a parameter the page size $|Page|$. We keep the default value for page size, which is 1104 bytes. Each page is further partitioned into $P$ packets, which are transmitted to the sensors.

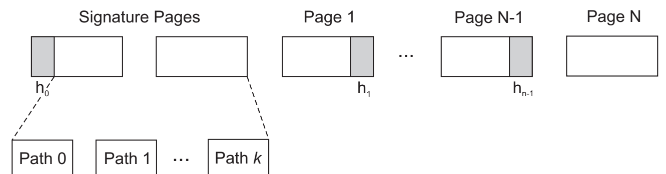



**Figure 11** Partitioning of the program image into pages before the dissemination into the network. This also shows the order at which a mote receives the signature, the pages and their hash values. Verification of the signature is possible by storing only one path at the time.

In Scatter, after we partition the image into pages, we compute a hash chain in reverse order from the last page to the first. We need to append these hash values at the end of the corresponding pages, so we have reserved the last packet of each page for that purpose. If we use MD5, a hash value is 16 bytes, so it fits in a packet of 23 bytes (the default value for TinyOS packets). So, each value of the hash chain padded with 0$s$ (to give 23 bytes) is attached at the end of the corresponding page.

The final step includes the addition of one or more pages at the beginning of the image that stores the first

value of the hash chain along with the signature we produced for it using the $r$-time signature scheme. The number of the extra pages is determined by the size of the signature, as this is given by Eq. (2). If $|S|$ denotes the size of the signature, then the number of the extra pages is $\lceil |S|/|Page| \rceil$. If necessary, we add some padding 0$s$ at the end of the signature to fill up the page.

Figure 11 illustrates how the secured program image is partitioned into pages before dissemination. In this typical example, the signature is stored in two extra pages at the beginning of the image. Besides these two pages, the only extra information transmitted is the last packet of each page, containing the corresponding hash chain value.

We now move to the side of the motes and describe the verification process for the signature and the authentication of each page. In particular, we are interested in the overhead posed by our security scheme in terms of memory and time.

#### *6.1.1 Memory Requirements*

The dissemination and authentication of the code is done in a per-page basis. As soon as the last packet of a page is received, the mote checks to see if it is complete and issues a request back to the sender for any missing packets, like in original Deluge. When the page is complete, a CRC check is done to verify its integrity. Then the mote verifies that the hash value of the page it just received is the same as the corresponding value that came with the previous page, before requesting the next page.

To be able to hash the page, the mote needs to buffer it in RAM. This requires a buffer equal to the page size. The default value of Deluge for that parameter is 1104 bytes, so it perfectly fits in a sensor node's memory (being 4 KB for Mica2, or 10 KB for Tmote). Of course, the page size is a parameter of Deluge that can easily be changed to any smaller value, if the final memory requirements exceed the available resources.

An exception for what is described above is the first two pages that the mote receives, which contain the signature of the hash chain commitment. They also need to be stored in EEPROM as the rest of the pages. However, the mote does not need to keep the whole page in RAM, but rather just each authentication path. This is because the signature is made up by authentication paths and the authentication of each path can be done independently by the others.

Referring to Figure 11, the mote first receives the hash value of Page 1. This will provide the indices to the public values. Then, the first authentication path of the signature will be received. The verification of that path evolves only a few hashing operations and a comparison of the result with the corresponding public value. This can be done fast enough by the mote, so that the path has been verified before the next path starts coming in. So, only a temporary storing is needed, equal to the size of a path, which dependents on the height of the Merkle trees at the base station.

**Table 2** Time performance of hash functions in Mica2 platform.

| *Function* | *Time to hash one block* | *Time to hash* 1104 *bytes* |
|---|---|---|
| SHA–1 | 7.56 ms | 131.7 ms |
| MD5 | 2.58 ms | 49.6 ms |

### *6.2 Experimental Evaluation*

In this subsection, we report the experimental evaluation of Scatter in two different hardware platforms, namely Mica2 and MicaZ. Both platforms have the same microcontroller, which offers 128 KB of data memory and 4 KB of program memory. The difference between the two is the RF transciever they use. Mica2 uses the Chipcon CC1000, while MicaZ uses the upgraded Chipcon CC2420, which implements the IEEE 802.15.4 standard.

For comparison purposes, we perform the same set of experiments for plain Deluge, Scatter as well as Seluge. We choose to compare Scatter with Seluge, because according to their experiments (Liu et al., 2008), their scheme outperforms the approach of Deng et al. (2006) and Dutta et al. (2006).

The main overhead in execution time that Scatter imposes on Deluge is due to the main two security operations evolved, i.e., hashing of the pages and verifying the signature. We measured that overhead for each operation separately, as well as the total overhead compared to plain Deluge over a complete image transfer, using our implementation in Mica2 motes.

We first examined the execution time needed to hash a page of default size (1104 bytes). This is shown in Table 2 for two different hash functions, SHA-1 and MD5. It is obvious from the results that the time performance of MD5 is considerably better than that of SHA-1. For both functions we used publicly available code and no optimization was made for the specific hardware. So, these values can be further improved.

We have already shown the signature verification times for Mica2 in Figure 7. So, now we show the overall time that it takes to download and authenticate a new program image on one mote from the PC and compared it with the time that it takes for plain Deluge. To investigate and compare the impact of dissemination code size on performance, we use six different code image sizes, ranging from 5 to 30 pages of 1104 bytes each. These correspond to images from 4.8 KB to 31 KB. Figure 12 shows our results. Note that the time for a mote to receive a new image is subject to packet losses, so we perform the same experiment 20 times and take an average over them. For this specific experiment we used $l = 80$, $k = 8$, $t = 65536$, $r = 32$, and $T = 32$. This

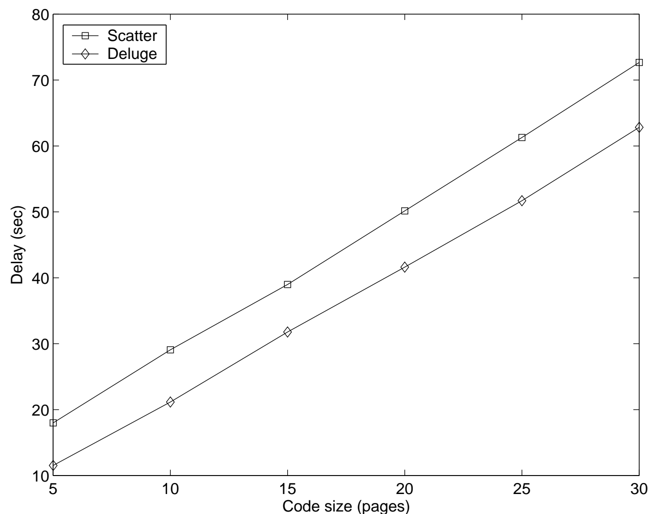



**Figure 12** Time taken to transfer an image to a Mica2 node.

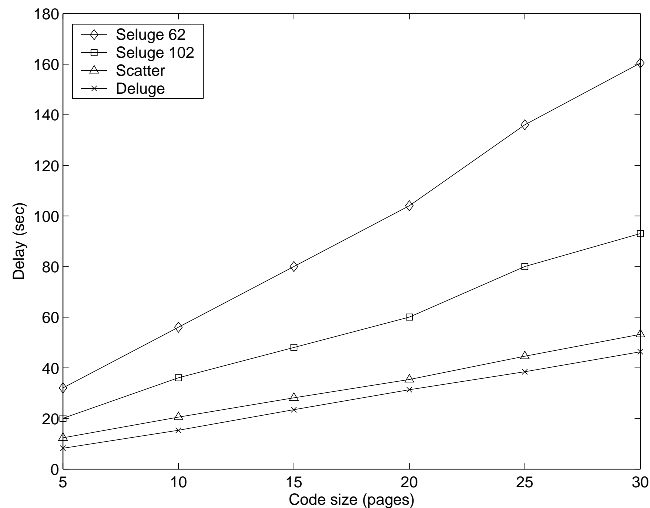



**Figure 13** Time taken to transfer an image to a MicaZ node.

setting gave a signature size of 1408 bytes and a public key size equal to 512 bytes.

An important observation that we can make from Figure 12 is that the overhead imposed on Deluge by Scatter is almost steady as the code size increases. This is because for applications that are larger by $p$ pages, the time overhead will increase by $p$ times the time for the sensor node to hash a page and compare it with the hash value included in the next page. This is insignificant compared to the overhead due to signature transfer and verification, which is independent on the program image size.

To compare Scatter with Seluge we run another set of experiments using MicaZ motes. This is because Seluge's implementation is based on CC2420 radio component on MicaZ to reduce its overhead. It uses the hardware cryptographic support available by that component for symmetric cryptographic operations and it also uses the larger packet payload sizes supported by IEEE 802.15.4, the standard implemented by CC2420. Seluge needs large packets to accommodate its hash values, but in this way it also decreases the total number of packets required for a given program image and therefore decreases the propagation delay.

Figure 13 shows the total delay to download different program images from the PC to a MicaZ mote, using Deluge, Scatter and Seluge. For Seluge we used two variants, one with packet size equal to 62 bytes and one with packet of 102 bytes. For Deluge and Scatter we used the default TinyOS packet size, which is 23. By using larger packet sizes for them also, it would only decrease the transfer time further.

Observing Figure 13, we can see the benefits from using an authentication scheme that is based only to hash operations, like Scatter, as opposed to a scheme that uses ECC based public key operations, like Seluge. Seluge uses TinyECC (Liu and Ning, 2008) on the motes, which provides an ECC implementation for TinyOS that includes an ECDSA (Elliptic Curve Digital Signature Algorithm) module. As a result, Seluge-62 introduces an average overhead of 255% compared to the completion time of Deluge and Seluge-102 brings it down to 114%. On the other hand, for Scatter we measure an average overhead of 24% compared to Deluge.

Let us also note that the lower transmission times observed in Figure 13 regarding Deluge and Scatter, compared to the corresponding measurements obtained for Mica2 in Figure 12, are due to the faster data rate of the CC2420 radio available in MicaZ. Mica2 uses the ChipCon CC1000 RF transceiver running at 38.9 kbps, while MicaZ features a much faster 250 kbps radio.

Finally, after using the key pair we have produced for authenticating new program images $r$ times ($r$ being for example 32 or 64), we can no longer use them, because the security level has dropped below acceptable levels. So we need to update them, and we use the same authentication scheme to distribute the new public key to the motes. To send this new public key to the motes we sign it using the current secret key, and send it to the motes just like if it was a new image (only much smaller). We embed a small bit pattern at the beginning to allow the motes realize that it is the new public key. In this way the motes will verify the new public key and start using it for the next images.

For $l = 80, k = 8, t = 65536, r = 32,$ and $T = 32$ we calculated the time needed to send a new public key in one mote in an authenticated way. The new public key (512 bytes) fits in one page, resulting in 3 pages to be transmitted in total (including 2 pages for the signature). Performing this experiment on Mica2, the mote was able to receive and verify the new public key in 7.01 seconds, which is a low price to pay, given that we need to perform this operation after $r = 32$ code disseminations.

## 7 Conclusions

In this paper we presented Scatter, an efficient and practical scheme for authenticated code dissemination in sensor networks. Scatter imposes asymmetric cryptography properties using *symmetric* cryptography primitives, outperforming other proposals that use ECC or RSA signature schemes. It minimizes the public key and signature sizes to values that are appropriate for sensor networks. The verification procedure at the motes is also time and computationally efficient, since it involves only hashing and comparison operations.

Scatter integrates these efficient security mechanisms with the propagation strategies of Deluge to provide a complete solution of secure code dissemination in wireless sensor networks. Our experiments with MicaZ and Mica2 motes demonstrate that Scatter is an efficient and practical solution, compared to other previous attempts, since it allows the fast code dissemination that Deluge offers, adding only a small overhead.

## References


American National Standards Institute (1998), 'ANSI X9.62-1998, public key cryptography for the financial services industry: The elliptic curve digital signature algorithm (ECDSA)'.

Deng, J., Han, R. and Mishra, S. (2006), Secure code distribution in dynamically programmable wireless sensor networks, *in* 'Proceedings of the fifth international conference on Information processing in sensor networks (IPSN '06)', pp. 292–300.

Dutta, P., Hui, J., Chu, D. and Culler, D. (2006), Securing the Deluge network programming system, *in* 'Proceeding of the 5th International Conference on Information Processing in Sensor Networks (IPSN 2006)', pp. 326–333.

Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E. and Culler, D. (2003), The nesC language: A holistic approach to networked embedded systems, *in* 'Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI '03)', ACM, New York, NY, USA, pp. 1–11.

Gennaro, R. and Rohatgi, P. (2001), "How to sign digital streams", *Information and Computation* , Vol. 165, Duluth, MN, USA, pp. 100–116.

Gura, N., Patel, A., Wander, A., Eberle, H. and Shantz, S. C. (2004), Comparing elliptic curve cryptography and RSA on 8-bit CPUs, *in* 'Cryptographic Hardware and Embedded Systems (CHES '04)', pp. 119–132.

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D. and Pister, K. (2000), "System architecture directions for networked sensors", *ACM SIGPLAN Notices* , Vol. 35, pp. 93–104.

Hui, J. W. and Culler, D. (2004), 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', pp. 81–94.

Lanigan, P. E., Gandhi, R. and Narasimhan, P. (2006), Sluice: Secure dissemination of code updates in sensor networks, *in* 'Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS '06)', p. 53.

Liu, A. and Ning, P. (2008), TinyECC: A configurable library for elliptic curve cryptography in wireless sensor networks, *in* 'Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN '08)', pp. 245–256.

Liu, A., Oh, Y.-H. and Ning, P. (2008), Secure and DoS-resistant code dissemination in wireless sensor networks using Seluge, *in* 'Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN '08)', pp. 561–562.

Merkle, R. (1988), A digital signature based on a conventional encryption function, *in* 'Advances in Cryptology – CRYPTO '87', Vol. 293 of *Lecture Notes in Computer Science*, Springer-Verlag, London, UK, pp. 369–378.

Pieprzyk, J., Wang, H. and Xing, C. (2003), Multiple-time signature schemes against adaptive chosen message attacks, *in* 'Selected Areas in Cryptography (SAC 2003)', Springer, pp. 88–100.

Reyzin, L. and Reyzin, N. (2002), Better than BiBa: Short one-time signatures with fast signing and verifying, *in* 'Proceedings of the 7th Australian Conference on Information Security and Privacy (ACISP '02)', Springer-Verlag, London, UK, pp. 144–153.

Shaheen, J., Ostry, D., Sivaraman, V. and Jha, S. (2007), Confidential and secure broadcast in wireless sensor networks, *in* 'IEEE Personal, Indoor, and Mobile Radio Communications (PIMRC '07)', Athens, Greece.

Stathopoulos, T., Heidemann, J. and Estrin, D. (2003), A remote code update mechanism for wireless sensor networks, Technical Report CENS-TR-30, University of California, Los Angeles, Center for Embedded Networked Computing.

Tan, H., Ostry, D., Zic, J. and Jha, S. (2009), A confidential and DoS-resistant multi-hop code dissemination protocol for wireless sensor networks, *in* 'Proceedings of the second ACM conference on Wireless network security (WiSec '09)', ACM, New York, NY, USA, pp. 245–252.

Ugus, O., Westhoff, D. and Bohli, J.-M. (2009), A ROM-friendly secure code update mechanism for WSNs using a stateful-verifier $\tau$-time signature scheme, *in* 'Proceedings of the second ACM conference on Wireless network security (WiSec '09)', ACM, New York, NY, USA, pp. 29–40.

Wang, Q., Zhu, Y. and Cheng, L. (2006), "Reprogramming wireless sensor networks: challenges and approaches", *IEEE Network* , Vol. 20, pp. 48–55.