A Practical Authentication Scheme for In-Network Programming in Wireless Sensor Networks

Ioannis Krontiris
Athens Information Technology
P.O.Box 68, 19.5 km Markopoulo Ave.
GR- 19002, Peania, Athens, Greece
ikro@ait.edu.gr

Tassos Dimitriou
Athens Information Technology
P.O.Box 68, 19.5 km Markopoulo Ave.
GR- 19002, Peania, Athens, Greece
tdim@ait.edu.gr

ABSTRACT

In-network programming has become a popular feature in sensor networks deployments, making it easy to update remotely the code running in sensor nodes. TinyOS includes Deluge as its default in-network programming system, but unfortunately it does not incorporate any security mechanism that ensures program authenticity. This makes it easy for an attacker to disseminate malicious code in the network and gain complete control or disrupt its functionality. We provide a protocol that yields source authentication in the group setting like a public-key signature scheme, only with signature and verification times much closer to those of a MAC. We show how Deluge can be augmented with our solution to give a secure and practical in-network programming system. Our implementation on the Mica2 platform allowed us to test the performance of this system on a real network of motes and prove its efficiency.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems

General Terms

Algorithms, Design, Performance, Security

Keywords

Sensor networks, In-network programming, Authentication

1. INTRODUCTION

The process of programming sensor nodes typically involves the development of the application in a PC and the loading of the program image to the node through the parallel or the serial port. The same process is repeated for all the nodes of the sensor network before deployment. However, after deployment, there is often the need to change the behavior of the nodes in order to adapt to new application requirements or new environmental conditions. This would require the effort of re-programming each individual node with the updated code and relocate it back to the deployment site. Network programming saves this effort by propagating the new code over the wireless link to the entire network, as soon as that code is loaded to only one node. Then, nodes reprogram themselves and start operating with the updated code.

As network programming simplifies things for legitimate users, it also simplifies things for attackers that want to disrupt the normal operation of the network or operate them for their own advantage. In currently deployed networks the nodes do not authenticate the source of the program; therefore an attacker could easily approach the deployment site and disseminate her own malicious/corrupted code in the network.

This possibility makes sensor networks deployments susceptible to outsider attacks. Besides loosing control of the network or getting back altered measurements, it is even possible that the network is reprogrammed with malicious code that has the same functionality as the legitimate code but also reports data to the adversary. In such a case legitimate users would never know that something is wrong. Hence, it is important that the sensor nodes can efficiently verify that the new code originates from a trusted source, namely the base station.

2. RELATED WORK

A recent work that proposes a solution for secure dissemination of code updates in sensor networks is described in [6]. The authors suggest the use of hash chains to efficiently authenticate each page of the program image. However, they are using cryptographic primitives to authenticate the initial commitment of the hash chain that are very demanding in time and computational power.

Another work on the same problem is described in [2], where the authors set the additional goal of DOS-resilience and therefore they need to authenticate each packet separately. To do so they construct a signed hash tree (similar to a Merkle tree) for *every* page in the program image, and they transmit these trees before the actual data. This increases considerably the overhead of packets sent and re-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
REALWSN'06, June 19, 2006, Uppsala, Sweden.
Copyright 2006 ACM 1-59593-431-6/06/0006 ...$5.00.

ceived by the motes. Moreover, due to memory constrains in the motes, these values need to be stored and loaded from the EEPROM, which is a very energy and time consuming operation.

Likewise, in [3] a reverse hash chain computed over the packets is also used. As the authors note in their experimental results, the overhead of authenticating each packet separately constrains the realizable radio throughput. Moreover, their signature verification at the motes is based on the RSA digital signature scheme, something that we have excluded from our design goals, due to intense computational requirements.

Authentication schemes for broadcasting messages in a sensor network that use only symmetric primitives exist, as for example the one proposed by Benenson et al. [1]. The authors keep the memory and computational overhead of their algorithm efficiently low. However they are concerned about the problem of authenticating broadcasted queries, which are normally less harmful messages with very small size, so their requirements are different than in our case and their solution cannot be applied here.

3. PROPOSED SOLUTION

To sign a program image we are following the approach by Gennaro and Rohatgi in [4] for signing digital streams. 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. In this way the sender needs to sign just the first block and then the properties of this signature will propagate to the rest of the stream through the "chaining" technique.

So, 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

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

and we attach each hash value h_i to page P_i, as shown in Figure 1. For the last hash value, $h_{N-1} = H(P_N)$. According to this scheme, we need to authenticate only h_0, which we will sign and release before the pages of the image.

To sign and verify h_0 we improve and extend the HORS r-times signature scheme [8]. Here we briefly describe the HORS scheme that is used to sign a message m. First, the signer generates a secret key SK that consists of t random values of l bits length. The public key PK is computed by applying a one-way function f to each of the values of the secret key and then distributing them to the intended receivers in an authenticated way. A message (in HORS) is signed according to the following steps:

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

$$H(m) = h_1 \| h_2 \| \ldots \| h_k.$$

2. Interpret each substring as integer. Use these integers to select a subset σ of k values out of the set SK.

3. This σ is the signature of the message m.

The recipients of the message m recompute the hash value of m, produce the same indices and pick the corresponding values of the set PK. Then they verify that the hash value of each member of the signature equals to the corresponding member of the public key PK. The signature is accepted if this is true for all k values.

Note that for each message that we sign a part of the secret key is leaked out. Some typical values for HORS are $l = 80$, $k = 16$ and $t = 1024$. In this case, assuming a hash output of 20 bytes, the public key will be $1024 \times 20 = 20,480$ bytes or 20 KB, which is not suitable for sensor nodes. Moreover, the security of the scheme and the size of the public key is directly related to the number of messages that we can sign.

To show this, 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 [8]) 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 Σ the attainable security level in bits, by equating the previous probability to $2^{-\Sigma}$, we see that Σ is given by

$$\Sigma = k(\log_2 t - \log_2 k - \log_2 r), \tag{1}$$

As an example, for $t = 1024$, $k = 16$ and $r = 4$ we get $\Sigma = 64$ bits of security. For $t = 65536$, $k = 8$ and $r = 32$ we get the same level of security but we can sign a lot more messages with the same key. However, since the number t of PK values determines the public/secret keys sizes, it is directly limited by the restrictions imposed by sensor networks capabilities.

We extend HORS in several ways: First the size of the public key is greatly reduced (to the size of a few hundred bytes) to fit in the memory of sensor nodes. Second, the number of messages that can be signed with the same instances of public/private key is increased. Finally, a tradeoff can be computed between signature size and public key size that would make our scheme appropriate for most sensor network applications.

3.1 Decreasing the public key size

We start our discussion by presenting how the public key size can be decreased. We distribute the values of the secret key into T Merkle trees [7], by separating these values into T groups, each with t/T values each. Then we use these values as leaves of the Merkle trees. The *roots* of the trees constitute the public key of our scheme (see Figure 2). To sign a message now we follow the HORS approach described previously except that the signature is made up by the selected secret values *along* with their corresponding authentication paths. The signature is accepted as authentic if upon evaluation of these paths the roots of the corresponding trees are produced.

As an example, let's apply to our scheme the same values we did for HORS, i.e., $l = 80$, $k = 16$ and $t = 1024$, and using a 20-byte hash output. If we construct 32 Merkle trees with 32 leaves each (so all 1024 secret values are covered), we will get 32 roots of trees, i.e., 640 bytes that will constitute our public key, compared to 20 KB we got from HORS. Also, from equation 1 we get 64 bits of security for $r = 4$ messages (program images). If we choose $l = 80$, $k = 8$ and $t = 65536$, and $r = 32$ we get the same level of security. In this case, by

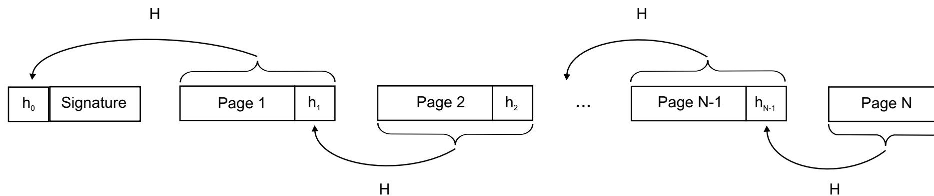

Figure 1: Page creation process for secure code updates.

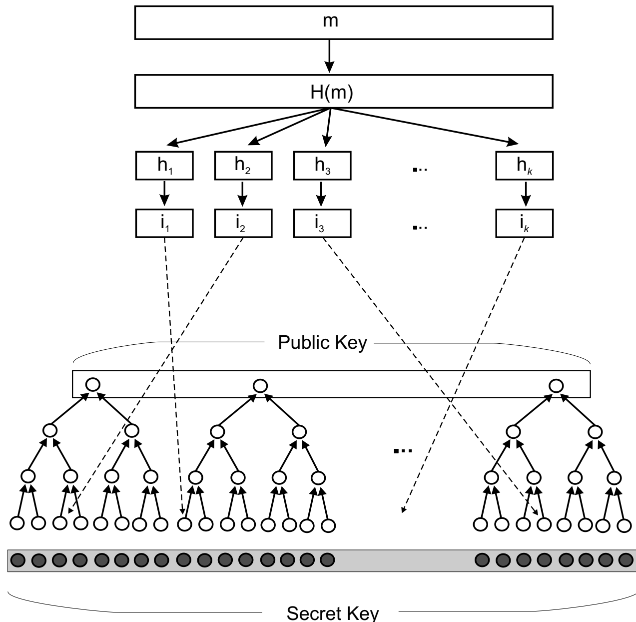

Figure 2: Signature construction.

constructing 64 Merkle trees of 1024 leaves each, the public key will become 1024 bytes, which is still an attractive value for use in sensor nodes.

3.2 Trading public key size for 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. This is true because the signature size depends on the length of the authentication paths, which is inversely proportional to number of trees (more trees means less secret values per tree and hence smaller height). To find this tradeoff between public/signature size let T denote the number of trees. Hence the public key size is simply

$$S_{PK} = |h|T, \tag{2}$$

where $|h|$ is the length of the output of the hash function 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.

To compute the signature size, we must recall that for a number of trees T 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 $\log t/T$. If we solve equation 1 for t, we get

$$t = 2^{\Sigma/k} kr$$

and so the length of the authentication paths becomes $\Sigma/k + \log(kr) - \log T$. The signature consists of k such authentication paths and since each path is made up of hash values, the signature size will be given by

$$S_{sig} = |h| \cdot (\Sigma + k \log_2(kr) - k \log_2 T). \tag{3}$$

From this equation it should be obvious that increasing the number of trees T (and hence the public key size) results in a decrease in the signature size.

4. BUILDING A SECURE IMAGE

Once the signature of the new program image has been created, we must prepare the image for dissemination. Deluge [5] first partitions the image into P pages, given as a parameter the page size S_{page}. Each page is further partitioned into N packets, which are transmitted to the sensors.

Figure 3 shows in detail the process of preparing a secure program image at the PC side for dissemination in the network. The boot image to be sent is made up by some metadata associated with it, information about the length of the image in bytes, and the image itself. Then, the boot image is partitioned into pages of fixed length (1081 bytes in this example), and a CRC is calculated for each one.

At the end of each page we add more bytes (equal to the packet size) where the hash value of the next page is stored. The hash value is 16 bytes, so it fits in a packet of 23 bytes (the default value for TinyOS packets). Therefore, we first partition the image into pages and then compute the hash chain in reverse from the last page to the first. Each value of the hash chain padded with 0s (to give 23 bytes) is then 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. The number of the extra pages is determined by the size of the signature (normally it should not be more than 2 pages). If S_{sig} denotes the size of the signature, then the number of the extra pages is $\lceil S_{sig}/S_{page} \rceil$

In our design we decided that these extra pages should be reserved explicitly for the signature and they should not contain any of the actual data. Therefore, we add some padding $0s$ at the end of the signature to fill up the page. Even though this may result in the transmission of some unnecessary packets, it keeps the algorithm at the node's side much simpler.

Figure 3 illustrates an example program image, where pages 0 and 1 are reserved for the signature. The original image is stored in pages $2 \dots P + 2$. Besides these two pages, the only extra information transmitted with the program image is the last packet of each page, containing the corresponding hash chain values.

The same mechanism is applied in order to update the

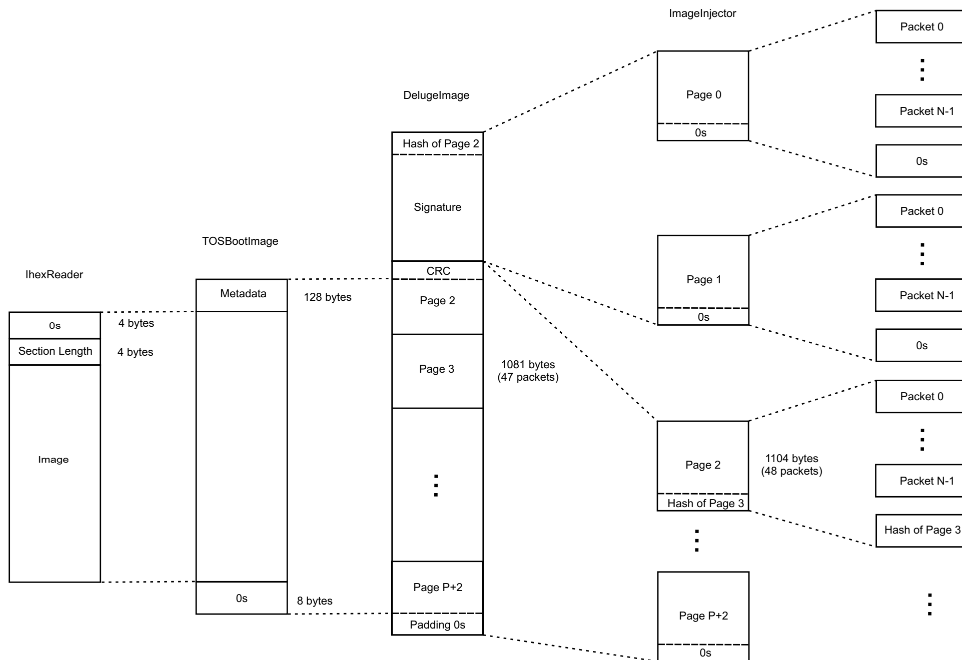

Figure 3: Partitioning of the program image into pages and packets before the dissemination into the network. Here each page is divided into $N = 48$ packets, the last one being the hash value of the next page. Also in this example the signature is 1408 bytes, so two extra pages are needed to transmit it.

public key. As we said, with any instance of the current secret key, we are allowed to sign r messages. After that, we need to produce new secret values, and therefore the public key will change as well. 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 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.

5. EVALUATION

We implemented our authentication protocol in TinyOS and incorporated it in standard Deluge 2.0 version, in order to measure its efficiency. We performed several experiments to evaluate the performance of our solution, using the same default values of packet payload and page size. In all of our experiments we used Blink as an example application to be disseminated into the network.

5.1 Memory requirements

In the secure variant of Deluge, when a mote receives a packet of the new program image, the first thing that it does is to store it at the external flash memory (EEPROM). At the end, the whole image will reside there along with all the extra information necessary to authenticate it. When a reboot command is issued, the mote will load only the useful information (i.e. the program image) into the program memory and start executing the code.

The dissemination 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 needs to verify that the hash value of the page it just received is the same as the corresponding value that came with the previous page.

To be able to hash the page, the mote needs to buffer it in RAM. That means a buffer equal to the page size is needed. The default value of Deluge for that parameter is 1104 bytes so it perfectly fits in a sensor node's memory (being 4KB for Mica2, or 10KB for Tmote). Of course the page size 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 rest of the operations evolved here have to do with verifying each path separately, so they do not impose any further memory requirements.

5.2 Time requirements

The overhead that the authentication scheme imposes on Deluge concerning the execution time can be traced 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.

The time needed to hash a page of default size (1104 bytes) in Mica2 platform is shown in Table 1 for two different hash functions, SHA-1 and MD5. Our implementation for both functions was based on publicly available code and no optimization was made for the specific hardware. Therefore, these values can be further improved, if some kind of optimization is implemented. In any case, it was obvious from

Table 1: **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

Table 2: Signature verification times for different signature sizes

Number of Trees T	Verification Time (ms)	Signature size (bytes)
4	269.1	2560
8	248.4	2240
16	227.7	1920
32	207	1600
64	186.3	1280
128	165.6	690

our experiments that we should incorporate MD5 in our implementation, since its time performance was considerably better than that of SHA-1.

For the signature verification, 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. Table 2 shows this relationship between signature size and verification time.

We measured 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. Table 3 shows our results for two different applications, Blink and Surge, both wired to Deluge. For this specific experiment we used $l = 80, k = 8, t = 65536, r = 32$, and $T = 32$. This setting gave a signature size of 1408 bytes and a public key size equal to 512 bytes. To interpret this table, lets take the example of Blink and calculate the average overhead per page. Deluge needs $2.07s$ per page, while for its secure variant it takes $2.21s$ per page. The number of pages are increased in the second case because of the signature and the inclusion of the hash values at the end of each page.

For the same parameters, we calculated the time needed to update the public key in one mote. The new public key (512 bytes) fits in one page, resulting in 3 pages to be transmitted in total (including the signature). The mote was able to verify the new key and update its table in 7.01 seconds.

Table 3: Completion times of Deluge and Secure Deluge for downloading different applications from the PC to a mote.

Application	Deluge version	Pages	Mean value of downloading time
Blink	Plain	23	47.75 s
	Secure	26	57.48 s
Surge	Plain	29	61.15 s
	Secure	32	70.88 s
Public Key Update	Secure	3	7.01 s

The time for a mote to receive a new image is subject to packet losses so it is not steady. That is why Table 3 shows the mean value of completion time taken over 15 repetitions. In Deluge, when the last packet of each page is received, a request message for any missing packets of that page is sent back to the sender. Then the sender retransmits these packets to complete the page. The same applies for the first two pages that contain the signature. Since the mote will authenticate a complete page, there is no need to change the secret key in order to retransmit a packet.

6. CONCLUSIONS

We present an efficient and practical scheme for authenticated in-network programming in wireless sensor networks. Our solution imposes asymmetric cryptography properties using *symmetric* cryptography primitives. The verification procedure at the motes is time and computational efficient, since it involves only hashing and comparison operations. We implemented our solution and integrated it in Deluge, showing that it can easily adapt to an existing in-network programming protocol. Our experiments proved that the memory and computational overhead imposed by our scheme is minimal and appropriate for applications running on sensor nodes.

7. REFERENCES

[1] Z. Benenson, L. Pimenidis, E. Hammerschmidt, F. C. Freiling, and S. Lucks. Authenticated query flooding in sensor networks. In *SEC 2006: Proceedings of the 21st IFIP International Information Security Conference*, May 2006.

[2] J. Deng, R. Han, and S. Mishra. Secure code distribution in dynamically programmable wireless sensor networks. Technical Report CU-CS-1000-05, University of Colorado, 2005.

[3] P. Dutta, J. Hui, D. Chu, and D. Culler. Securing the deluge network programming system. In *IPSN 2006: Proceeding of the 5th International Conference on Information Processing in Sensor Networks*, April 2006.

[4] R. Gennaro and P. Rohatgi. How to sign digital streams. *Information and Computation*, 165(1):100–116, 2001.

[5] 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*, pages 81–94, 2004.

[6] P. Lanigan, R. Gandhi, and P. Narasimhan. Sluice: Secure dissemination of code updates in sensor networks. In *Proceeding of the 26th International Conference on Distributed Computing Systems*, 2006.

[7] R. C. Merkle. A certified digital signature. In *CRYPTO '89: Proceedings on Advances in cryptology*, pages 218–238, New York, NY, USA, 1989.

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