Secure Erasure and Code Update in Legacy Sensors

Ghassan O. Karame(✉) and Wenting Li

NEC Laboratories Europe, 69115 Heidelberg, Germany
{ghassan.karame,wenting.li}@neclab.eu

Abstract. Sensors require frequent over-the-air reprogramming to patch software errors, replace code, change sensor configuration, etc. Given their limited computational capability, one of the few workable techniques to secure code update in legacy sensors would be to execute Proofs of Secure Erasure (PoSE) which ensure that the sensor's memory is purged before sending the updated code. By doing so, the updated code can be loaded onto the sensor with the assurance that no other malicious code is being stored. Although current PoSE proposals rely on relatively simple cryptographic constructs, they still result in considerable energy and time overhead in existing legacy sensors.

In this paper, we propose a secure code update protocol which considerably reduces the overhead of existing proposals. Our proposal naturally combines PoSE with All or Nothing Transforms (AONT); we analyze the security of our scheme and evaluate its performance by means of implementation on MicaZ motes. Our prototype implementation only consumes 371 bytes of RAM in TinyOS2, and improves the time and energy overhead of existing proposals based on PoSE by almost 75 %.

Keywords: Secure code update · All or nothing transformations · Proofs of secure erasure

1 Introduction

Sensors and actuators require frequent over-the-air reprogramming to update their cryptographic credentials, patch software errors, change configuration, etc. Clearly, code update needs to be *securely* realized in order to ensure that the newly downloaded code is installed in its entirety and can be correctly executed in the installation environment with the assurance that no other malicious code is being stored.

The literature features a number of solutions based on device attestation to secure code execution in embedded devices [13,14,18,26,27]; however, recent studies show that existing (hardware and software-based) techniques are still far from being practical to be deployed in legacy sensors [22].

To remedy this, Perito and Tsudik [22] introduced the notion of Proofs of Secure Erasure (PoSE) in order to secure code update. PoSE enable a device to prove to a remote verifier that it has purged all of its memory. For example, in a

M. Conti et al. (Eds.): TRUST 2015, LNCS 9229, pp. 283–299, 2015.
DOI: 10.1007/978-3-319-22846-4_17

PoSE, the prover downloads large amounts of incompressible data which fills all of its writable memory contents, and then proves (e.g., using MACs, or Proofs of Data Possession schemes [5,15,29]) to a remote verifier that it has downloaded the data in its entirety. The main intuition here is that if the prover can attest that it is storing new data which covers all of its writable memory, then the prover must have purged its (old) memory contents. By doing so, PoSE can be used as a prelude to secure code update [22]; once all prior state has been erased, new code can be downloaded onto the device with the assurance that no other malware or malicious code is being stored.

Although existing PoSE schemes rely on relatively simple cryptographic constructs, such as MACs, these schemes still result in considerable energy and time overhead in existing low-cost sensors. For example, in MicaZ motes, the computation of HMACs based on SHA1 for 648 KB of data (which constitutes the total memory of a MicaZ mote) requires almost 90 s, and consumes $3.5\,\mu J$ per byte [22]. This is mainly due to the fact that legacy sensors are still not optimized to execute cryptographic algorithms.

In this paper, we address this problem and we propose an efficient PoSE scheme which considerably improves the verification overhead of existing PoSE proposals in legacy sensors. Our construction makes use of basic operations such as XORing and cyclic bitwise shifting; we show that our solution incurs moderate computational costs when compared to existing PoSE proposals—while ensuring secure memory erasure. We then extend our proposal and present a secure code update protocol, SUANT (Secure code Update based on All or Nothing Transforms), which naturally combines PoSE with an efficient all or nothing transform; SUANT requires the same number of communication rounds as PoSE, and results in a considerably smaller computational overhead when compared to existing secure code update protocols based on PoSE. We evaluate the performance of SUANT by means of implementation on MicaZ sensor nodes [2]. Our evaluation results show that our scheme only consumes 371 bytes of RAM, and incurs approximately 75 % energy and time savings when compared to the optimized secure code update protocol of [22].

The remainder of this paper is organized as follows. In Sect. 2, we present PoSE, and describe the building blocks that we will use throughout the paper. In Sect. 3, we introduce our proposals aimed at efficiently proving memory erasure and secure code update. In Sect. 4, we implement and evaluate our secure code update protocol using MicaZ motes. In Sect. 5, we overview related work in the area, and we conclude the paper in Sect. 6.

2 Background and Preliminaries

We start by outlining our system and attacker model. We then discuss the shortcomings of device attestation for sensors, introduce PoSE, and the building blocks that we will use throughout this paper.

2.1 Model

Our system consists of a verifier $\mathcal{V}$ and a resource-constrained prover $\mathcal{P}$. Here, $\mathcal{V}$ is interested in updating the code of $\mathcal{P}$; for that purpose, $\mathcal{V}$ transmits over the air the required code to be updated. To ensure that $\mathcal{P}$ can correctly execute the new code update, $\mathcal{V}$ requires a proof that $\mathcal{P}$ has correctly downloaded the code update, and does not host any malware in its memory. By memory, we refer to the entirety of the writable storage available to $\mathcal{P}$. We assume that $\mathcal{V}$ has a larger memory than $\mathcal{P}$, and knows the exact memory size of $\mathcal{P}$. This is a reasonable assumption since we consider the typical case where $\mathcal{P}$ is a sensor mote, whose total memory capacity is accurately reported in its datasheets.

We assume a computationally bounded adversary $\mathcal{A}$ which controls $\mathcal{P}$. Here, $\mathcal{A}$ is a program or a malware executing on $\mathcal{P}$, and has complete read/write access over the memory of $\mathcal{P}$. We assume, nevertheless, that $\mathcal{A}$ does not have write access to a small part of the read-only memory (ROM) of the device. Read-only memory can be instantiated in most embedded devices by locking parts of the device's memory. Writing to this memory portion without physically accessing the device is not possible.

Similar to existing software attestation protocols [22,26,27], we assume that $\mathcal{A}$ cannot modify the hardware configuration of $\mathcal{P}$, and can only communicate with the verifier (and no other external entity). Assuming wireless communication with the sensors, this can be practically enforced if the verifier actively jams the prover throughout their interaction phase. Jamming can be effectively realized by the verifier—without affecting the ability of the prover to interact with the verifier—by emitting signals with larger strength than the maximum threshold set at prover's side [20,22].

Notice that since $\mathcal{A}$ is restricted to $\mathcal{P}$'s running environment without any external help, then $\mathcal{A}$ is bounded by the computational and storage capabilities of the prover (i.e., by $\mathcal{P}$'s memory). Similar to existing protocols, we assume that the device authenticates the verifier prior to the start of the secure code update protocol. To this end, we assume, e.g., that the public key of the verifier and the authentication algorithm are stored in the ROM of the device. Throughout the rest of the paper, we do not focus on the overhead incurred by authenticating the verifier since this step is not particular to our protocols and applies to all similar protocols.

2.2 Remote Attestation

As mentioned earlier, device attestation constitutes one possible way to ensure that an embedded device is executing correct software. Device attestation can be categorized in two main branches: hardware-based attestation, and software-based attestation.

Hardware-based attestation leverages hardware support, such as TPM chips, ARM Trustzone [3], Intel SGX [4], to securely bootstrap a trusted measurement environment. Hardware-based attestation offers strong security guarantees but is unfortunately not yet supported on low-cost embedded devices [12]. On the other hand, software-based attestation [26,27] aims to verify the correctness of software executing on a device without the reliance on additional hardware support. Recently, several attacks have been reported against existing software-based attestation schemes [8,28,30].

In [13], Jakobsson and Johansson propose the reliance on a memory printing algorithm to practically enable software-based attestation. The proposed algorithm acquires a random seed from a secure source of (pseudo-)randomness located in the close proximity of the device (e.g., a SIM or a smart card), executes an expansion function using the seed as input to fill the RAM of the device, then mixes and shuffles the output of the expansion function. By doing so, if a malware is executing in the RAM of the device, then this will slow down the aforementioned process that requires all the RAM for fast computation—which would facilitate the detection of misbehavior. This solution is, however, unsuitable for legacy sensors which do not have any SIM/smart card slot and solely rely on the slow radio channel for communication. This renders the detection of delays originating from the execution in RAM a rather challenging task.

2.3 Proofs of Secure Erasure (PoSE)

In [22], Perito and Tsudik proposed proofs of secure erasure (PoSE) for resource-constrained sensor nodes. Their PoSE comprise two steps:

Step 1: Erase memory. The prover erasures all of its memory by downloading high entropy data (e.g., an encrypted stream of data) sent by the verifier. Here, the code must be large enough to fill all the writable memory of the prover.

Step 2: Proof of erasure. The prover attests to the verifier that it has stored all the downloaded code. This can be done e.g., by sending a MAC of the downloaded code to the verifier.

As shown in [22], the basic PoSE protocol described above can be transformed into a secure code update protocol by invoking an additional communication round between the prover $\mathcal{P}$ and the verifier $\mathcal{V}$. The resulting code update protocol is depicted in Fig. 1. Here, the verifier first chooses a random encryption key K' and encrypts the code[1] to be updated $P_1, \ldots, P_n$ using a semantically secure encryption function Enc. Upon reception of the ciphertext blocks $C_0, \ldots, C_n$, the prover uses the last m blocks as a MAC key K, constructs a MAC over the remaining ciphertext blocks $MAC(K, C_0, \ldots, C_{n-m})$, and sends the MAC to $\mathcal{V}$ who verifies it. If the verification passes, $\mathcal{V}$ sends to $\mathcal{P}$ key K' in order for $\mathcal{P}$ to decrypt $C_0, \ldots, C_n$ into the plaintext code.

The aforementioned PoSE protocol, and the corresponding secure code update protocol incur considerable communication and computational costs on the prover, namely:

[1] In case the code size to be updated is smaller than the total writable memory of the device, the verifier pads the code with zeros until it reaches the device's memory size.

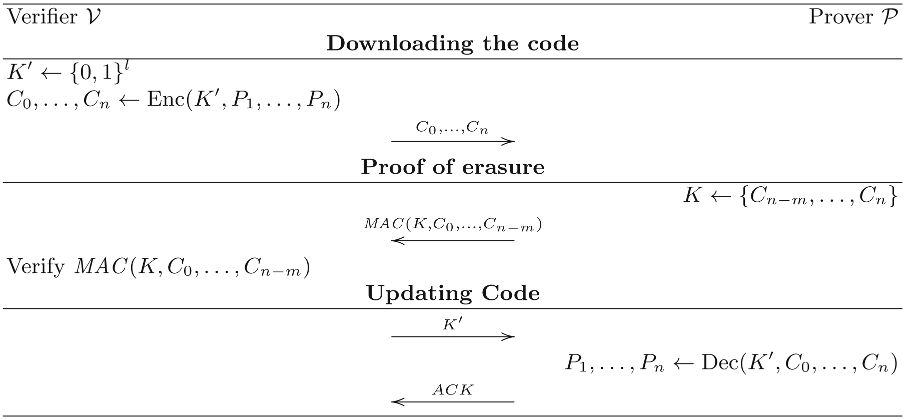

Verifier $\mathcal{V}$
Prover $\mathcal{P}$
Downloading the code
$K' \leftarrow \{0,1\}^l$
$C_0, \ldots, C_n \leftarrow \text{Enc}(K', P_1, \ldots, P_n)$
$\xrightarrow{C_0,\ldots,C_n}$
Proof of erasure
$K \leftarrow \{C_{n-m}, \ldots, C_n\}$
$\xleftarrow{MAC(K,C_0,\ldots,C_{n-m})}$
Verify $MAC(K, C_0, \ldots, C_{n-m})$
Updating Code
$\xrightarrow{K'}$
$P_1, \ldots, P_n \leftarrow \text{Dec}(K', C_0, \ldots, C_n)$
$\xleftarrow{ACK}$

Fig. 1. Summary of the optimized secure code update protocol of [22].

1. The prover needs to download data which is as large as its own memory (e.g., 648 KB in MicaZ nodes).
2. The prover needs to compute a MAC over the entire data to verify that the prover has indeed stored the downloaded bytes[2].
3. The secure code update protocol results in an additional communication round for the verifier to send the encryption key K' in case the PoSE protocol successfully completes.

One possible alternative to reduce the downloaded data size would be to make use of a secure expansion function which fills the entire prover's memory with high entropy data (similar to [13]). By doing so, it might be possible to detect if a malware still resides in the memory of the device by timing the verification step (i.e., I/O access vs. heavy computations). In an experiment that we conducted, we measured the time and energy required by a MicaZ sensor to erase its total memory by *(i)* downloading code from an external verifier over the radio channel as in [22] or by *(ii)* adapting the memory printing algorithm of [13] to fill the prover's memory. Our findings show that the former approach consumes 8.64 μJ/B at 8.86 KB/s,[3] while the latter solution consumes 35.74 μJ/B with a throughput of 0.77 KB/s. Therefore, filling the prover's memory with data downloaded from an external verifier emerges as the most workable mechanism to purge the memory of legacy sensors.

On the other hand, to reduce the computational costs of the PoSE protocol in Fig. 1, one alterative would be to selectively verify a fraction of the downloaded data (e.g., similar to POR/PDP [5,15,29]). This approach considerably speeds up the verification stage in PoSE, but still requires that the verifier verifies the

[2] As shown in [22], computing an HMAC-SHA1 over 648 KB of data in a MicaZ mote requires almost 90 s.

[3] The maximum claimed transmission throughput of TI-CC2420 radio chip used in MicaZ motes is 250 kbps, which translates to 31250 bytes/sec. However, our experiments show that the effective throughput is around 8860 bytes/sec using TinyOS 2.0.

integrity of a considerable fraction of the data in order to acquire reasonable guarantees that the prover did not erase a small part of its memory. Notably, assuming a block size of m bits, and that the prover did not erase c out of the total t blocks of data, then the verifier needs to selectively verify d blocks to achieve a detection probability of $1-(1-\frac{c}{t})^d$. As an example, in a MicaZ mote, the total memory is 648 KB; assuming $m = 128$ bit, then to detect that the prover did not keep 1,000 bits of its old code, the verifier would have to selectively check almost 30 % of the data blocks to achieve a detection probability close to 90 %.

2.4 All or Nothing Transforms (AONT)

An All or Nothing Transform (AONT) is a transform that outputs sequences of blocks such that given all but one of the output blocks, it is infeasible to compute any of the original input blocks [16]. An AONT is given by a pair of p.p.t. algorithms $(\mathbb{E}, \mathbb{D})$ where [10,16]:

$\mathbb{E}$ The encoding algorithm is a probabilistic algorithm which takes as input a message $x \in \{0,1\}^*$, and outputs ciphertext y.

$\mathbb{D}$ The decoding algorithm is a deterministic algorithm which takes as input ciphertext y, and outputs either a message $x \in \{0,1\}^*$ or $\perp$ to indicate that the input ciphertext is invalid.

To construct an AONT, Rivest [25] suggested the package transform which leverages a block cipher and maps m block strings to $m+1$ block strings. The first m output blocks are computed by encrypting the input blocks using a random key K. The last output block is computed by XORing K with the encryption of each of the previous output blocks, using a key K_0 that is publicly known.

Desai [10] proposed a faster version where the block cipher round which uses K_0 is skipped and the last output block is computed as the XOR of all the ciphertext blocks: That is, given block cipher F/F^{-1} and on input $x[1]\ldots x[m]$, Desai's transform outputs $y[1]\ldots y[n]$, with $n = m+1$, where:

$$y[i] = x[i] \oplus F_K(i),\ 1 \leq i \leq n-1,$$

$$y[n] = K \bigoplus_{i=1}^{n-1} y[i].$$

Notice that Desai's AONT leverages a block cipher to ensure that the output blocks have high entropy. In this paper, we leverage Desai's AONT to construct an efficient secure code update scheme for legacy sensors. By doing so, our construct requires that the prover fetches all the output code blocks in order to decode any part of the (plaintext) code; if the prover possesses all but one output block, then it is computationally infeasible for the prover to acquire any meaningful bit of information about any plaintext block. As we show later, this also removes the need for an additional communication round to transmit the code encryption key.

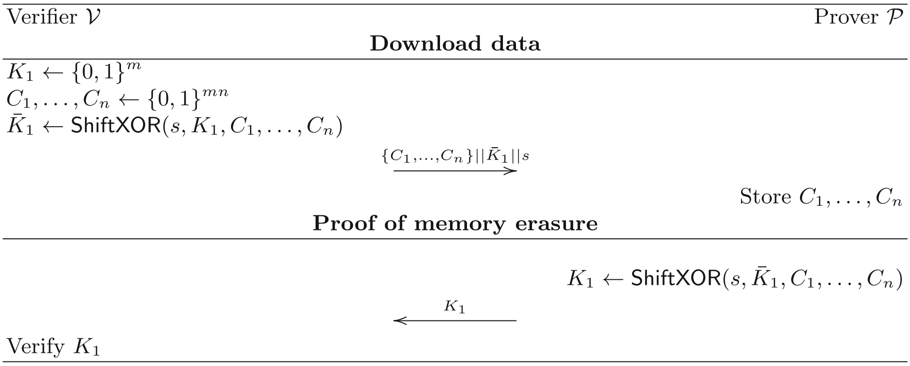

Verifier $\mathcal{V}$
Prover $\mathcal{P}$
Download data
$K_1 \leftarrow \{0,1\}^m$
$C_1, \ldots, C_n \leftarrow \{0,1\}^{mn}$
$\bar{K}_1 \leftarrow \mathsf{ShiftXOR}(s, K_1, C_1, \ldots, C_n)$
$\xrightarrow{\{C_1,\ldots,C_n\}||\bar{K}_1||s}$
Store $C_1, \ldots, C_n$
Proof of memory erasure
$K_1 \leftarrow \mathsf{ShiftXOR}(s, \bar{K}_1, C_1, \ldots, C_n)$
$\xleftarrow{K_1}$
Verify K_1

Fig. 2. Sketch of our lightweight PoSE scheme. Here, we assume that the code $C_1, \ldots, C_n$, of size mn, fills the total writable memory of $\mathcal{P}$ (with the exception of the minimum amount of volatile memory required to execute the PoSE scheme).

3 Lightweight Proofs of Secure Erasure and Code Update

In this section, we present and analyze our proposal for secure code update. We will do so incrementally, starting with an initial scheme which enables the construction of efficient proofs of secure memory erasure, and later extending it to construct our code update protocol, SUANT.

3.1 Lightweight Proofs of Secure Erasure

Our solution shares the same intuition with existing PoSE proposals [22]; namely, the prover fills its memory with high-entropy data acquired from the verifier and proves to the latter that it has stored all the downloaded data. As mentioned in Sect. 2.3, this alternative is more efficient than filling the memory of the prover using a local source of pseudo-randomness. The main difference between our proposal and the PoSE of [22] lies in the fact that, here, the data is specifically constructed in such a way that if the prover has stored that data in its entirety, then $\mathcal{P}$ can issue a compact proof of memory erasure—without the need to rely on MACs.

In our solution, this is achieved as follows: the verifier picks a random secret, divides the data into equal sized-blocks, and XORs the secret with (a function of) the data blocks which the prover is requested to download. The output of the XOR is appended and sent to the prover as the last data block. If the prover can correctly extract the secret inserted by the verifier, then this offers a strong proof that the prover has downloaded all the data sent by the verifier.

Notice that the straightforward XORing of the data blocks with the secret does not offer a proof of memory erasure, since a malicious prover can simply XOR all the downloaded blocks without the need to store them. Later on, after receiving the last block (which is the XOR of the secret with the remaining data blocks), the prover can correctly revert the secret without having to store all the downloaded data. Therefore, we require that the bits pertaining to different data block are pseudorandomly (cyclic) shifted before being XORed; here, we reveal the (pseudorandom) seed used in the shifting procedure at the very end of the data transmission. By doing so, our solution ensures that the advantage of an adversary in correctly computing the secret by performing intermediate results, or dropping a fraction of the data bits/blocks is negligible.

The detailed protocol of our PoSE scheme unfolds in Fig. 2. We stress that the code required to execute our PoSE scheme resides in a read-only part of the prover's memory (cf. Sect. 4); this does not give any advantage for the adversary to modify this code.

The verifier $\mathcal{V}$ first chooses n random data blocks $C_1, \ldots, C_n$ of length m bits each, such that mn fills the total *available* writable memory of the device. This corresponds to the total writable memory of the device excluding *(i)* the memory occupied by the code required to download and process the data, and *(ii)* the minimum amount of volatile memory necessary to execute the code of PoSE. $\mathcal{V}$ then chooses a random secret K_1 and a seed s of size m bits each, and executes the following ShiftXOR procedure:

```
procedure $\bar{K}_1 \leftarrow \text{ShiftXOR}(s, K_1, C_1, \ldots, C_n)$
    $S \leftarrow G(s)$
    $l = \log_2 m$
    $\bar{K}_1 \leftarrow K_1$
    for $i = 0 \ldots n-1$ do
        $c \leftarrow S_{il \rightarrow l(i+1)}$
        $\bar{K}_1 \leftarrow \bar{K}_1 \oplus \{C_{(i+1)}\}_{\gg c}$
    end for
end procedure
```

Here, $S_{x \rightarrow y}$ refers to the bit sequence of S indexed from position x to y, $X_{\gg y}$ refers to the bitwise cyclic shift of X by y positions, and $G : \{0,1\}^m \rightarrow \{0,1\}^{nl}$ is an expansion function. For example, G can be instantiated by iteratively applying a hash function using as input the seed and a counter until the required number of bits are reached. The verifier then sends $C_1, \ldots, C_n || \bar{K}_1 || s$ to the prover. Notice that the ShiftXOR procedure is symmetric. That is, K_1 can be obtained by computing ShiftXOR$(s, \bar{K}_1, C_1, \ldots, C_n)$.

Claim 1. *Assuming a secure cryptographic source of randomness on $\mathcal{V}$, the protocol of Fig. 2 enables the verifier to detect that an adversary has not erased any m bits in its memory with overwhelming probability.*

Proof Sketch. *Suppose that a malicious code of size $b > 0$ persists in the memory of $\mathcal{P}$ after the completion of our PoSE. Then, this means that the adversary was able to compute K_1 without storing all the downloaded data in its memory. Recall that we assume that $C_1, \ldots, C_n$ fill the total writable memory of the prover with the exception of the minimum amount of volatile memory required to run the PoSE code. Moreover, K_1 is generated from a cryptographically secure source of randomness and therefore cannot be easily guessed. Moreover, since s is*

Verifier $\mathcal{V}$		Prover $\mathcal{P}$
	Downloading data output by AONT	
$K_1 \leftarrow \{0,1\}^m$		
$P_1, \dots, P_n \leftarrow$ Code		
$\{C_1, \dots, C_n\} \leftarrow \mathbb{E}_{K_1}^{AONT}(P_1, \dots, P_n)$		
$\bar{K}_1 \leftarrow \mathsf{ShiftXOR}(s, K_1, C_1, \dots, C_n)$		
	$\xrightarrow{\{C_1,\dots,C_n\}\|\|\bar{K}_1\|\|s}$	
		Store $\{C_1, \dots, C_n\}$
	Proof of memory erasure and code update	
		$K_1 \leftarrow \mathsf{ShiftXOR}(s, \bar{K}_1, C_1, \dots, C_n)$
	$\xleftarrow{K_1}$	
Verify K_1		
		$\{P_1, \dots, P_n\} \leftarrow \mathbb{D}_{K_1}^{AONT}(C_1, \dots, C_n)$

Fig. 3. Sketch of SUANT. Our construct combines PoSE with Desai's AONT in order to reduce the communication rounds required to prove secure code update.

communicated to $\mathcal{P}$ at the very end of the transmission, then this precludes any straightforward pre-computation of K_1. Since the data is also generated from a secure source of randomness on $\mathcal{V}$, then its entropy also rules out any possibility of compression. Recall also that the adversary cannot modify the code required to execute our PoSE scheme, since this code resides in a read-only part of $\mathcal{P}$'s memory.

Without knowledge of s, notice that each bit of each data block can affect the outcome of any bit of K_1 (due to the $\mathsf{ShiftXOR}$ routine). That is, any intermediate processing on the received bits (e.g., dropping some bits, XORing bits) can affect any of the m bits of the computed response. In other words, if the adversary stores $b > 0$ m-bits blocks of malicious code after the completion of the protocol (e.g., by dropping bm bits of the received data), then the adversary has to guess the correct shifting applied to at least b blocks of received data. This guessing probability is bounded by $\max(m^{-b}, 2^{-m})$.

Assuming $m = 128$ bits, the probability that a malicious code of size 1000 bits persists in the memory of $\mathcal{P}$ is bounded by 2^{-56}. We contrast this to existing PoSE schemes based on selective checking, where the probability that the verifier detects that the adversary did not erase 1,000 bits of its old memory contents after checking the integrity of 30 % of the downloaded blocks is approximately 90 %, when the prover's memory size is 648 KB.

3.2 SUANT: Secure Code Update Based on AONT

We now show how to extend our aforementioned PoSE scheme in order to construct an efficient secure code update protocol.

Notice that extending a PoSE into a secure code update protocol can be easily realized by *(i)* first padding the code to be updated to reach the total memory size of the prover, *(ii)* encrypting the padded code, and *(iii)* executing PoSE is over encrypted code. However, as shown in [22], this results in an additional communication round between the prover and the verifier in order to enable the latter to communicate the encryption key once PoSE is completed. We point out that the decryption key should only be shared with $\mathcal{P}$ after the PoSE has correctly completed since, otherwise, there is a risk that malware acquires access to the newly updated code which might contain sensitive information (e.g., credentials).

In what follows, we offer a natural extension to our PoSE scheme which satisfies this requirement without incurring an additional communication round. Our extension, dubbed SUANT, combines PoSE with an AONT in order to ensure that only if the prover has downloaded and stored all the encrypted code update, then it can acquire the necessary decryption key to revert the encrypted code and update its code.

The detailed protocol of SUANT is depicted in Fig. 3. Here, the code to be updated is first encrypted using key K_1, which will be subsequently used as the secret XORed with the data blocks in the ShiftXOR procedure. From that point on, SUANT unfolds similarly to our PoSE protocol in Fig. 2. Recall here that the code required to execute our scheme resides in a read-only part of the prover's memory (cf. Sect. 4).

Notice that by first encrypting the code and then XORing all the ciphertext blocks with the encryption key, this exactly yields the AONT transform of Desai (cf. Sect. 2.4). One major difference between SUANT and Desai's AONT lies in the fact that the last output block is replaced with $\bar{K}_1$, as outputted by ShiftXOR, which corresponds to the XOR of the pseudorandomly shifted ciphertext blocks with K_1. As mentioned earlier, this prevents the adversary from computing intermediate XOR on the fly, without the need to store the downloaded blocks.

Claim 2. *Assuming a secure cryptographic source of randomness on $\mathcal{V}$, the protocol of Fig. 3 enables the verifier to detect that an adversary has not securely updated his code with overwhelming probability.*

Proof Sketch. *It is easy to see that, given our assumptions,* (i) *the prover has a fixed and known memory size, and* (ii) *the adversary cannot modify the hardware of the provers, the security of SUANT follows directly from Claim 1 (cf. Sect. 3.1) and from the security of Desai's transform [10].*

Namely, since the downloaded code has high entropy (recall that the code is encrypted), and fills the total memory of the prover, then this prevents any straightforward attack where the adversary e.g., compresses the code. Similarly, the adversary cannot hide malware in parts of the writable memory, since our code update fills the entire memory of the device, including the volatile memory (with the exception of the minimum amount of RAM required to execute SUANT). Moreover, the use of Desai's AONT also ensures that the prover cannot acquire any meaningful bit of plaintext code unless it has processed all the output ciphertext blocks [10].

Since s is communicated at the very end of the transmission, then the prover has to store all the blocks, in order to subsequently revert $\bar{K}_1$, compute K_1, and decode the encrypted blocks to acquire the code update. As shown in Claim 1, *the probability that a malicious code of size bm bits resides in the memory of $\mathcal{P}$ after the successful completion of* SUANT *is given by* $\max(m^{-b}, 2^{-m})$*, which corresponds to the probability that the adversary guesses the correct shifting of all the b blocks or the key* K_1*.*

Reducing I/O Costs in SUANT: The ShiftXOR routine employed by SUANT incurs high I/O costs since it requires access to each and every data block. Notice that this overhead can be reduced if ShiftXOR only operates on a randomly chosen fraction f of the blocks. Such an alternative approach ensures that the secret is XORed (line 7 of ShiftXOR) with a randomly selected fraction f of the data blocks and thus requires the prover to only fetch those blocks from memory—thus tremendously reducing I/O costs. Here, the advantage of the adversary in computing the correct key K_1 without storing any given b ciphertext blocks of size m is bounded by $\max(m^{-b}, (1-f)^b)$. For example, when $f = 0.5$, if the adversary does not delete 1,000 bits of its old code (e.g., and selectively deletes 8 ciphertext blocks with size $m = 128$ bits each), then the probability that she can correctly compute K_1 is bounded by 0.004. We evaluate the comparative performance of this approach in Sect. 4.

4 Implementation and Evaluation

In this section, we implement and evaluate SUANT in MicaZ motes. For comparison purposes, we also evaluate the secure code update protocol of [22].

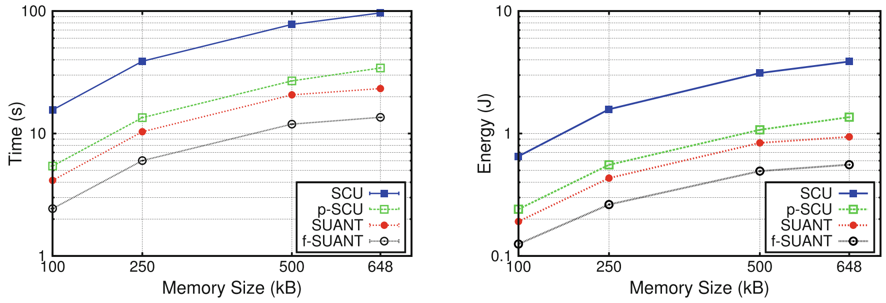
4.1 Implementation Setup

In order to evaluate the performance of our proposal in a realistic setting, we implemented SUANT on the ATMEGA128 micro-controller mounted on a MicaZ sensor. MicaZ [2] has a total memory of 648 KB, divided into 512 KB of external flash memory, 128 KB of internal flash, 4 KB of SRAM, and 4 KB of EEPROM. To access the on-chip memory, we made use of the *InternalFlashC* and *ProgFlashC*[4] modules from TinyOS bootloader (TOSBoot). The maximum transmission throughput of MicaZ is bounded by 250 kbps; however, our experiments suggest that only 30 % of this throughput can be effectively attained in a realistic scenario.

In addition to SUANT, we implemented f-SUANT, the optimized version of SUANT in which a fraction f of the data blocks are randomly processed by the ShiftXOR routine. In our implementation, we set $f = 0.5$; as mentioned earlier, this ensures the detection of a malicious prover which did not erase 1,000 bits (or more) of its old memory content with a probability of 0.996. For comparison purposes, we also implemented the optimized secure code update protocol of [22] (SCU) and a variant protocol adapted from [22] which replaces the verification of the entire downloaded code by a probabilistic verification (using MACs) of a fraction $p = 0.3$ of the downloaded data blocks[5]; we refer to this protocol by p-SCU.

[4] For that purpose, we extended the *ProgFlash* interface using AVR Libc.

[5] In this case, the probability to detect that a prover did not delete 1,000 bits of its old code is 0.9.



(a) Latency of the code update process w.r.t. the code size.

(b) Energy dissipated by MicaZ nodes during code update.

Fig. 4. Performance evaluation using MicaZ sensors. Each data point is averaged over 10 independent measurements; we do not include the corresponding 95 % confidence intervals due to their small size.

Table 1. Required code and volatile memory sizes.

	Total Memory (bytes)	RAM (bytes)	ROM (bytes)
SUANT	15,516	371	6822
***f*-SUANT**	15,718	384	6960
SCU	19,256	610	8562
***p*-SCU**	19,436	614	9722

Our implementation was integrated with TinyOS2. We relied on the TinySec and TinyECC libraries [17,19] to implement the cryptographic algorithms. We instantiated MACs using HMAC-SHA1, and made use of the Lehmer pseudo-random number generator [21]. In all the implemented schemes, we assume a fixed block size $m = 128$ bits.

Since all the implemented protocols require the initial transmission of a code of size mn bits and its decryption, we did not measure the overhead incurred by these processes. As shown in Sect. 2.3, our findings show that the code transmission to a MicaZ mote consumes 8.64 μJ/B at 8.86 KB/s.

In our experiments, we measured the time and energy that are consumed by the above mentioned four protocols, namely SUANT, f-SUANT, SCU, and p-SCU, in accessing and computing the memory blocks until the decryption key is obtained. To measure the energy consumption of the implemented protocols, we relied on the Avrora simulator [32] which provides an accurate cycle-based simulation of the ATMEGA128 micro-controller. All data points in our (latency) plots are averaged over 10 independent measurements; where appropriate, we also show the corresponding 95 % confidence intervals.

Ideally, the code update protocol should be stored in the ROM of the device to prevent tampering with the process. At present, many embedded devices support the use of mask ROM (e.g., the MSP430 micro-controller). In our case, we included the codes responsible for executing SUANT, f-SUANT, SCU, and p-SCU (respectively) in the internal flash of the MicaZ mote. Recall that ATmega128 allows part of its internal flash to be locked from writing—thus emulating a read-only memory. For instance, setting Boot Lock Bit 0 to '10' in ATmega128 and the BOOTSZ fuse to '00' on the bootloader section grants us an 8 KB equivalent of read-only memory in the internal flash [1]. Notice that, once locked, this memory can only be unlocked by means of physical access through the JTAG debugger.

4.2 Evaluation Results

Latency and Energy Overhead: In Fig. 4, we compare the latency and energy overhead incurred by SUANT and *f*-SUANT, when compared to SCU and *p*-SCU, with respect to the varied writable memory size of the device. Our results show that SUANT improves by more than 75 % the energy and time consumption of SCU, and results in more than 30 % energy and latency savings when compared to *p*-SCU. For example, to securely update code installed on MicaZ sensors whose total memory size is 648 KB, SUANT only requires 23.3 s and 0.94 joules, while SCU requires 96.6 s and 3.87 joules. *f*-SUANT further improves the performance of SUANT by reducing I/O costs by almost 50 %; our findings indicate that *f*-SUANT improves the latency and energy of *p*-SCU by almost 60 %. Recall that both SUANT and *f*-SUANT achieve higher detection probabilities when compared to *p*-SCU (which relies on selectively verifying 30 % of the downloaded code blocks).

Memory Usage: Table 1 summarizes the memory usage of SUANT and *f*-SUANT. Our results show that the total code size of SUANT (and *f*-SUANT) is almost 4 KB smaller than that of SCU and *p*-SCU. Moreover, SUANT almost halves the RAM consumption of SCU and only requires up to 371 bytes of RAM. These memory savings mainly originate from the fact that SUANT does not make use of cryptographic hashes and only relies on basic operations such as bitwise shifting and bitwise XORing—which consume less memory in legacy sensors. As shown in [22], HMAC-SHA1 alone occupies around 4500 bytes of ROM, and 120 bytes of RAM when loaded into memory. Since SUANT (and *f*-SUANT) leaves a smaller footprint in the RAM, this makes it harder for the attacker to compress the data and hide the malicious code—when compared to SCU.

Notice that since we integrated our implementation with TinyOS, the underlying code size for all protocols was larger than the maximum lockable memory in the bootloader section of the MicaZ mote. To remedy this, we can separate our codes into two parts: one part containing the memory accessing and computation routines (such as ShiftXOR) which we include in the bootloader section of the flash. The second part containing the necessary networking routines (such as the code required to send and receive bits) can be stored in the remaining part of the internal flash (i.e., in the application section). Recall that program code within the bootloader section has the capability to read/write to the entire internal flash memory through SPM (Store to Program Memory) instruction [1].

In Table 1, we show the minimum code size which should be included in read-only memory (labelled by "ROM") for all protocols; our results show that SUANT consumes a total of 6822 bytes of ROM—-almost 2 KB less ROM than SCU. Recall that the bootloader section is limited to 8 KB in size; this suggests that SUANT and f-SUANT can be directly integrated into the MicaZ motes using this approach. The critical parts of SCU (and p-SCU) on the other hand cannot fit entirely in the bootloader section in MicaZ.

5 Related Work

In this section, we overview related work in the area.

Securing Code-Update in Embedded Devices: Deng *et al.* [9] propose the use of Merkle hash trees and hash chains in order to authenticate code distribution in wireless sensor networks. In [11], Dutta *et al.* leverage authenticated streams in order to secure code update in the TinyOS network programming system. In [33], Ugus *et al.* propose to authenticate code updates using an efficient stateful verifier T-time signature scheme based on Merkle's one-time signature. However, these proposals do not aim at proving to a remote party that the code has been securely distributed and installed within the embedded device.

In [26], Seshadri *et al.* propose indisputable code execution in order to dynamically establish a trusted code base on remote untrusted wireless sensor nodes. In [13,14], the authors propose the reliance on a novel memory printing algorithm to practically enable software-based attestation. However, the proposed scheme relies on a trusted proxy that executes secure cryptographic algorithms, such as SIM card, that needs to be located in the close proximity of the device; clearly, this assumption cannot be met in existing sensor nodes.

A number of contributions address the problem of secure data deletion [6,23,24]; however, as far as we are aware, only few works consider the problem of securely deleting data in resource-constrained devices [22] and proving to a third party that data was securely deleted from these devices. In [22], Perito and Tsudik propose the notion of Proofs of Secure Erasure (PoSE) as an enabler of secure code update in embedded devices. In this paper, we borrow the notion of PoSE, and we propose lightweight PoSE and secure code update protocols that considerably improve the performance and energy consumption of existing proposals.

All or Nothing Transforms: All-or-nothing transforms (AONTs) were first introduced in [25] and later studied in [7,10,16]. The majority of AONTs leverage a secret key that is embedded in the output blocks. Once all output blocks are available, the key can be recovered and single blocks can be inverted. As such, AONT is not an encryption scheme and does not require the decoder to have any key material.

In [31], Stinson proposed a fast linear all or nothing transform based on matrix multiplication. Karame *et al.* showed in [16] that by first encrypting the data then post-processing it using an efficient Stinson-like transform over the field $\mathbb{F}^2$, one can construct an encryption mode which ensures that any single block of data cannot be decrypted unless the adversary has acquired almost all the ciphertext blocks and the encryption key.

6 Conclusion

In this paper, we tackled the problem of securing code update in legacy sensors. Here, code update needs to be securely realized in order to ensure that the newly downloaded code is installed in its entirety and can be correctly executed in the installation environment with the assurance that no other malicious code is being stored.

To this end, we proposed an efficient secure code update scheme, SUANT, which naturally combines PoSE with an efficient all or nothing transform inspired by Desai's transform [10]. We analyzed the security of SUANT, and we evaluated its performance by means of implementation on MicaZ sensor nodes [2]. Our evaluation results show that our scheme consumes a small footprint in RAM, and considerably improves the time and energy overhead of existing secure code update protocols.

References

1. ATmega128 Datasheet: Available from http://www.atmel.com/images/doc2467.pdf
2. MicaZ: Wireless Measurement System. http://www.openautomation.net/uploadsproductos/micaz_datasheet.pdf
3. Building a Secure System using TrustZone Technology (2009). http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf
4. Software Guard Extensions Programming Reference (2013). https://software.intel.com/sites/default/files/329298-001.pdf
5. Ateniese, G., Di Pietro, R., Mancini, L.V., Tsudik, G.: Scalable and efficient provable data possession. In: Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks, SecureComm 2008, pp. 9:1–9:10. ACM, New York, NY, USA (2008)
6. Bauer, S., Priyantha, N.B.: Secure data deletion for linux file systems. In: Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10, SSYM 2001. USENIX Association, Berkeley, CA, USA (2001)
7. Boyko, V.: On the security properties of OAEP as an all-or-nothing transform. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 503–518. Springer, Heidelberg (1999)
8. Castelluccia, C., Francillon, A., Perito, D., Soriente, C.: On the difficulty of software-based attestation of embedded devices. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS 2009, pp. 400–409. ACM, New York, NY, USA (2009)

9. Deng, J., Han, R., Mishra, S.: Secure code distribution in dynamically programmable wireless sensor networks. In: Proceedings of the 5th International Conference on Information Processing in Sensor Networks, IPSN 2006, pp. 292–300. ACM, New York, NY, USA (2006)
10. Desai, A.: The security of all-or-nothing encryption: protecting against exhaustive key search. In: Bellare, M. (ed.) CRYPTO 2000. LNCS, vol. 1880, pp. 359–375. Springer, Heidelberg (2000)
11. Dutta, P.K., Hui, J.W., Chu, D.C., Culler, D.E.: Securing the deluge network programming system. In: Proceedings of the 5th International Conference on Information Processing in Sensor Networks, IPSN 2006, pp. 326–333. ACM, New York, NY, USA (2006)
12. Eldefrawy, K., Francillon, A., Perito, D., Tsudik, G.: SMART: secure and minimal architecture for (establishing a dynamic) root of trust. In: NDSS 2012, 19th Annual Network and Distributed System Security Symposium, San Diego, USA, 5–8 February 2012
13. Jakobsson, M., Johansson, K.-A.: Practical and secure software-based attestation. In: LightSec (2011)
14. Jakobsson, M., Stewart, G.: Mobile malware: why the traditional AV paradigm is doomed, and how to use physics to detect undesirable routines. In: BlackHat (2013)
15. Juels, A., Jr., B.S.K.: PORs: proofs of retrievability for large files. In: ACM Conference on Computer and Communications Security, pp. 584–597 (2007)
16. Karame, G.O., Soriente, C., Lichota, K., Capkun, S.: Securing cloud data in the new attacker model. IACR Cryptology ePrint Archive 2014, p. 556 (2014)
17. Karlof, C., Sastry, N., Wagner, D.: Tinysec: a link layer security architecture for wireless sensor networks. In: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, SenSys 2004, pp. 162–175. ACM, New York, NY, USA (2004)
18. Koeberl, P., Schulz, S., Sadeghi, A.-R., Varadharajan, V.: Trustlite: a security architecture for tiny embedded devices. In: Proceedings of the Ninth European Conference on Computer Systems, EuroSys 2014, pp. 10:1–10:14. ACM, New York, NY, USA (2014)
19. Liu, A., Ning, P.: Tinyecc: a configurable library for elliptic curve cryptography in wireless sensor networks. In: Proceedings of the 7th International Conference on Information Processing in Sensor Networks, IPSN 2008, IEEE Computer Society, Washington, DC, USA (2008)
20. Martinovic, I., Pichota, P., Schmitt, J.B.: Jamming for good: a fresh approach to authentic communication in wsns. In: Proceedings of the Second ACM Conference on Wireless Network Security, WiSec 2009, pp. 161–168. ACM, New York, NY, USA (2009)
21. Payne, W.H., Rabung, J.R., Bogyo, T.P.: Coding the lehmer pseudo-random number generator. Commun. ACM **12**(2), 85–86 (1969)
22. Perito, D., Tsudik, G.: Secure code update for embedded devices via proofs of secure erasure. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 643–662. Springer, Heidelberg (2010)
23. Reardon, J., Basin, D., Capkun, S.: Sok: secure data deletion. In: Proceedings of the 2013 IEEE Symposium on Security and Privacy, SP 2013, pp. 301–315. IEEE Computer Society, Washington, DC, USA (2013)

24. Reardon, J., Ritzdorf, H., Basin, D., Capkun, S.: Secure data deletion from persistent media. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS 2013, pp. 271–284. ACM, New York, NY, USA (2013)
25. Rivest, R.L.: All-or-nothing encryption and the package transform. In: Biham, E. (ed.) FSE 1997. LNCS, vol. 1267, pp. 210–218. Springer, Heidelberg (1997)
26. Seshadri, A., Luk, M., Perrig, A., van Doorn, L., Khosla, P.: Scuba: secure code update by attestation in sensor networks. In: Proceedings of the 5th ACM Workshop on Wireless Security, WiSe 2006, pp. 85–94. ACM, New York, NY, USA (2006)
27. Seshadri, A., Perrig, A., Doorn, L.V., Khosla, P.: Swatt: software-based attestation for embedded devices. In: Proceedings of the IEEE Symposium on Security and Privacy (2004)
28. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS 2007, pp. 552–561. ACM, New York, NY, USA (2007)
29. Shacham, H., Waters, B.: Compact proofs of retrievability. In: Pieprzyk, J. (ed.) ASIACRYPT 2008. LNCS, vol. 5350, pp. 90–107. Springer, Heidelberg (2008)
30. Shankar, U., Chew, M., Tygar, J.D.: Side effects are not sufficient to authenticate software. In: Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13, SSYM 2004, pp. 7–7. USENIX Association, Berkeley, CA, USA (2004)
31. Stinson, D.R.: Something about all or nothing (transforms). Des. Codes Crypt. **22**(2), 133–138 (2001)
32. Titzer, B.L., Lee, D.K., Palsberg, J.: Avrora: scalable sensor network simulation with precise timing. In: Proceedings of the 4th International Symposium on Information Processing in Sensor Networks, IPSN 2005. IEEE Press, Piscataway, NJ, USA (2005)
33. Ugus, O., Westhoff, D., Bohli, J.-M.: A rom-friendly secure code update mechanism for wsns using a stateful-verifier t-time signature scheme. In: Proceedings of the Second ACM Conference on Wireless Network Security, WiSec 2009, pp. 29–40. ACM, New York, NY, USA (2009)