Secure Code Update for Embedded Devices via Proofs of Secure Erasure*

Daniele Perito[1] and Gene Tsudik[2]

[1] INRIA Rhône-Alpes, France
[2] University of California, Irvine, USA

Abstract. Remote attestation is the process of verifying internal state of a remote embedded device. It is an important component of many security protocols and applications. Although previously proposed remote attestation techniques assisted by specialized secure hardware are effective, they not yet viable for low-cost embedded devices. One notable alternative is software-based attestation, that is both less costly and more efficient. However, recent results identified weaknesses in some proposed software-based methods, thus showing that security of remote software attestation remains a challenge.
Inspired by these developments, this paper explores an approach that relies neither on secure hardware nor on tight timing constraints typical of software-based technqiques. By taking advantage of the bounded memory/storage model of low-cost embedded devices and assuming a small amount of read-only memory (ROM), our approach involves a new primitive – Proofs of Secure Erasure (PoSE-s). We also show that, even though it is effective and provably secure, PoSE-based attestation is not cheap. However, it is particularly well-suited and practical for two other related tasks: secure code update and secure memory/storage erasure. We consider several flavors of PoSE-based protocols and demonstrate their feasibility in the context of existing commodity embedded devices.

1 Introduction

Embedded systems are encountered in many settings, ranging from mundane to critical. In particular, wireless sensor and actuator networks are used to control industrial systems as well as various utility distribution networks, such as electric power, water and fuel [24, 13]. They are also widely utilized in automotive, railroad and other transportation systems. In such environments, it is often imperative to verify the internal state of an embedded device to assure lack of spurious, malicious or simply residual code and/or data.

Attacks on individual devices can be perpetrated either physically [1] or remotely [30, 14, 15]. It is clearly desirable to detect and isolate (or at least restore) compromised nodes. One way to accomplish this is via *device attestation*,

a process whereby a trusted entity (e.g., a base station or a sink) verifies that an embedded device is indeed running the expected application code and, hence, has not been compromised. In recent years, several software-based attestation protocols have been proposed [27, 29, 31]. The goal of these protocols is to verify the trustworthiness of resource-constrained systems, without requiring dedicated tamper-resistant hardware or physical access to the device. Attestation based on tamper-resistant hardware [12], though effective [17], is not yet viable on low-cost commodity embedded devices. Furthermore, hardware attestation techniques, while having stronger security properties, ultimately rely on a per-device TPM and the availability of a trusted BIOS that begins attestation at boot time.

In contrast, remote software attestation typically involves a challenge-response interaction, whereby a trusted entity, the *verifier*, challenges a remote system, called the *prover*, to compute a cryptographic checksum of its internal state, i.e., code memory, registers and program counter. Depending on the specific scheme, the prover either computes this checksum using a fixed checksum routine and a nonce [29], or downloads a new routine from the verifier as part of the protocol [31]. The checksum routine sequentially updates the checksum value by loading and processing device memory blocks. Since the verifier is assumed to know the exact memory contents and hardware configuration of the prover, it can compute the expected checksum value and match it with the prover's response. If there is a match, the prover is assumed to be *clean*; otherwise, either it has been compromised or a fault has occurred. In either case, appropriate actions can be taken by the verifier.

Recently, several proposed software-based attestation schemes were shown to be vulnerable [9] to certain attacks, summarized in Section 2. These negative results show that software-based attestation remains to be an interesting and important research challenge.

To summarize, hardware-based attestation techniques are not quite practical for current and legacy low-cost embedded systems. Whereas, state-of-the-art in software-based attestation offers unclear (or, at best, *ad hoc*) security guarantees. These factors motivate us to look for alternative approaches. Specifically, in this paper we zoom out of just attestation and consider a broader issue of *secure remote code update*. To obtain it, we introduce a new cryptographic primitive called a *Proof of Secure Erasure* (PoSE). We suggest some simple PoSE constructs based on equally simple cryptographic building blocks. This allows us, in contrast to prior software-based attestation techniques, to obtain provable security guarantees, under reasonable assumptions.

Our approach can be used to obtain several related security properties for remote embedded devices. The most natural application is *secure memory erasure*. Embedded devices might collect sensitive or valuable data that – after being uploaded to a sink or a base station – must be securely erased. Also, if code resident on an embedded device is sensitive or proprietary, it might eventually need to be securely erased by a remote controller. We note that secure erasure may be used as a prelude to secure code update or attestation. This is because, after

secure erasure of all prior state, new (or old) code can be downloaded onto an embedded device with the assurance that no other code or data is being stored.

The intended **contribution** of this paper is three-fold:

1. We suggest a simple, novel and practical approach to secure erasure, code update and attestation that falls between (secure, but costly) hardware-based and (efficient, but uncertain in terms of security) software-based techniques.
2. We show that the problem of secure remote code update can be addressed using Proofs of Secure Erasure (PoSE-s).
3. We propose several PoSE variants and analyze their security as well as efficiency features. We also assess their viability on a commodity sensor platform.

Organization: Section 2 reviews related work. Next, Section 3 describes the envisaged network environment and states our assumptions. Section 4 presents our design rationale, followed by proposed protocols in Section 5. Implementation, experiments and performance issues are discussed in Section 6. Limitations and directions for future work are addressed in Section 7. An extension to support multi-device attestation is deferred to Appendix A.

2 Related work

We now summarize related work, which generally falls into either software- or hardware-based attestation methods. We also summarize some relevant cryptographic constructs.

2.1 Hardware attestation

Static Integrity Measures: Secure boot [2] was proposed to ensure a chain of trusted integrity checks, beginning at power-on with the BIOS and continuing until the kernel is loaded. These integrity checks compare the computation of a cryptographic hash function with a signed value associated with the checked component. If one of the checks fails, the system is rebooted and brought back to a known saved state.

Trusted Platform Module (TPM) is a secure coprocessor that stores an integrity measure of a system according to the specifications of the Trusted Computing Group (TCG) [34]. Upon boot, the control is passed to an immutable code base that computes a cryptographic hash of the BIOS, hash that is then securely stored in the TPM. Later, control is passed to the BIOS and the same procedure is applied recursively until the kernel is loaded. In contrast to secure boot, this approach does not detect integrity violations, instead the task is left to a remote verifier to check for integrity.

[25] proposed to extend the functionality of the TPM to maintain a chain of trust up to the application layer and system configuration. In order to do so,

they extend the Linux kernel to include a new system call that measures files and adds the checksum in a list stored by the kernel. The integrity of this list is then sealed in the TPM. A similar goal is pursued in NGSCB [12], that takes a more radical approach by partitioning a system in a trusted and an untrusted part, each running a separate operating system, where only the trusted part is checked.

Dynamic Integrity Measures: In [21], the use of TPM is extended to provide system integrity checks of run-time properties with ReDAS (**Re**mote **D**ynamic **A**ttestation **S**ystem). At every system call, a kernel module checks the integrity of constant properties of dynamic objects, e.g., invariant relations between the saved frame pointer and the caller's stack frame. Upon detection of an integrity violation, the kernel driver seals the information about the violation in the TPM. A remote verifier can ask the prover to send the sealed integrity measures and thus verify that no integrity violations occurred. However ReDAS only checks for violations of a subset of the invariant system properties and nothing prevents an adversary to succeed in subverting a system without modifying the properties checked by ReDAS. Extending the set of attested properties is difficult due to the increased number of false positives generated by this approach, for example in case of dynamic properties classified as invariants by mistake.

2.2 Software attestation

Most software-based techniques rely on challenge-response protocols that verify the integrity of code memory of a remote device: an attestation routine on the prover computes a checksum of its memory along with a challenge supplied by the verifier. In practice, memory words are read sequentially and fed into the attestation function. However, this simple approach does not guarantee that the attestation routine is computed faithfully by the prover. In other words, a prover can deviate (via some malicious code) from its expected behavior and still compute a correct checksum, even in the presence of some malicious memory content.

Time-based attestation: SWATT [29] is a technique that relies on response timing to identify compromised code: memory is traversed using a pseudo-random sequence of indexes generated from a unique seed sent by the verifier. If a compromised prover wants to pass attestation, it has to redirect some memory accesses to compute a correct checksum. These redirections are assumed to induce a remotely measurable delay in the attestation that can be used by the verifier to decide whether to trust the prover's response. The same concept is used in [27] where, the checksum calculation is extended to include also dynamic properties, e.g., the program counter or the status register. Furthermore the computation is optimised by having the checksum computed only on the attestation function itself.

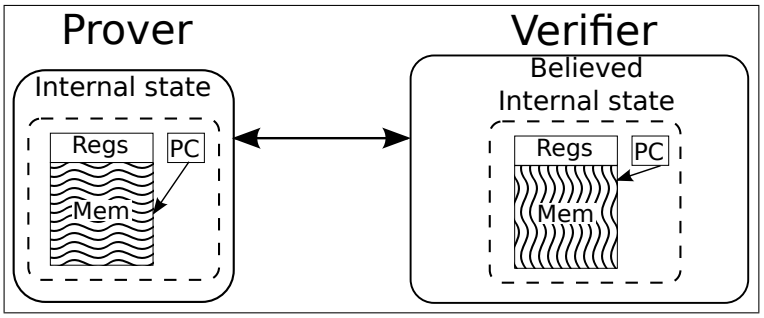

Fig. 1: Generic remote attestation.

Jakobsson, et al. [18] proposed an attestation scheme to detect malware on mobile phones. This attestation scheme relies on both careful response timing and memory filling. Timing is used to measure attestation computation as well as external memory access and wireless links. Security of this approach depends on a number of hardware-specific details (e.g., flash memory access time). Hence, formal guarantees and portability to different platforms appear difficult to achieve.

Memory-based attestation: In [35] sensors collaborate to attest the integrity of their peers. At deployment time, each empty node's memory is filled with randomness, that is supposed to prevent malicious software from being stored, without deleting some parts of the original memory. A similar approach is taken in [10], but, instead of relying on pre-deployed randomness, random values are generated using a PRF seeded by a challenge sent by the verifier and are used to fill the prover's memory. However, this does not assure compliance to the protocol of a malicious node that could trade computation for memory and still produce a valid checksum.

Gatzer et al. [16] suggest a method where random values are sent to a low-end embedded device (e.g., a SIM card) and then read back by the verifier, together with the attestation routine itself (called *Quine* in the paper). This construction, while quite valid, was only shown to be effective on an 8-bit Motorola MCU with an extremely simple instruction set. Also, this scheme applies only to RAM, whereas, we our approach aims to verify all memory/storage of an embedded device.

Attestation based on self-modifying code: [31] proposed to use a distinct attestation routine for each attestation instance. The routine is transferred right before the protocol is run and uses self-modifying code and obfuscation techniques to prevent static analysis. Combined with timing of responses, this makes it difficult for the adversary to reverse-engineer the attestation routine fast enough to cheat

the protocol and produce a valid (but forged) result. However, this approach relies on obfuscation techniques that are difficult to prove secure. Furthermore, some such techniques are difficult to implement on embedded systems, where code is stored in flash memory programmable only by pages.

Attacks: Recently, [9] demonstrated several flaws and attacks against some software attestation protocols. Attacks can be summarized as: failure to verify other memories apart from code memory (exploited through ROP attacks [30]); insufficient non-linearity in time-based attestation routines, which could be exploited to generate correct results over forged memory; failure to recognize that legitimate code memory can be compressed and thus save space for malicious code, while still remaining accessible for attestation. Also, [32] points out that side-effects, such as cache misses, are not sufficient to check software integrity using time-based approaches such as [20].

2.3 Provable Data Possession and Proofs of Retrievability

The problem at hand bears some resemblance to Provable Data Possession (PDP) [3, 4] and Proof of Retrievability (POR) schemes [19]. However, this resemblance is superficial. In settings envisaged by POR and PDP, a resource-poor client outsources a large amount of data to a server with an unlimited storage capacity. The main challenge is for a client to efficiently and repeatedly verify that the server indeed stores all of the client's data. This is markedly different from attestation where the prover (embedded device) must not only prove that it has the correct code, but also that it stores *nothing else*. Another major distinction is that, in POR and PDP, the verifier (client) is assumed not to keep a copy of its outsourced data. Whereas, in our setting, the verifier (base station) obviously keeps a copy of any code (and/or data) that embedded devices must store.

2.4 Memory-Bounded Adversary

Cryptographic literature contains a number of results on security in the presence of a memory-bounded adversary [8]. Our setting also features an adversary in the guise of a memory-limited prover. However, the memory-bounded adversary model involves two or more honest parties that aim to perform some secure computation. Whereas, in our case, the only honest party is the verifier and no secrets are being computed as part of the attestation process.

3 Assumptions and Adversary Model

Secure code update involves a *verifier* $\mathcal{V}$ and a *prover* $\mathcal{P}$. Internal state of $\mathcal{P}$ is represented by a tuple $S = (M, RG, pc)$ where M denotes $\mathcal{P}$'s memory of size n (in bits), $RG = rg_1, ..., rg_m$ is the set of registers and pc is the program counter.

We refer to S_P as the real internal state of the prover and S_V the internal state of the prover, as viewed by the verifier. Secure code update can be viewed as a means to ensure that $S_V = S_P$. Our notation is reflected in Table 1.

Table 1: Notation Summary.

$X \longleftarrow Y \ : \ Z$	Y sends message Z to X
$X_1, ..., X_t \Longleftarrow Y \ : \ Z$	Y multicasts message Z to $X_1, ..., X_t$
$\mathcal{V}$	Verifier
$\mathcal{P}$	Prover
ADV	Adversary
M	Prover's contiguous memory
$M[i]$	i-th bit in M $(0 \leq \ i < n)$
n	Bit-size of M
RG	Prover's registers $rg_1, ..., rg_m$
pc	Prover's program counter
$S_P = (M, R, pc)$	Prover's internal state
S_V	Verifier' view of Prover's internal state
$R_1...R_n$	Verifier's n-bit random challenge
$C_1...C_n$	n-bit program code (see below)
k	Security parameter
K	MAC key

$\mathcal{P}$ is assumed to be a generic embedded device – e.g., a sensor, an actuator or a computer peripheral – with limited memory and other forms of storage. For the ease of exposition, we assume that all $\mathcal{P}$'s storage is homogeneous and contiguous. (This assumption can be easily relaxed, as discussed in section 6.2) From here on, the term *"memory"* is used to denote all writable storage on the device. The verifier is a comparatively powerful computing device, e.g., a laptop-class machine.

Our protocol aims to ascertain the internal state of $\mathcal{P}$. The adversary is a program running in the prover's memory, e.g., a malware or a virus. Since the adversary executes on $\mathcal{P}$, it is bounded by the computational capabilities of the latter, i.e., memory size n.

We assume that the adversary cannot modify hardware configuration of $\mathcal{P}$[1], i.e., all anticipated attacks are software-based. The adversary has complete read-/write access to $\mathcal{P}$'s memory, including all cryptographic keys and code. However, in order to achieve provable security, our protocol relies on the availability of a small amount of Read-Only Memory (ROM) that the adversary can read, but not modify. Finally, the adversary can perform both passive (such as eavesdropping) and active (such as replay) attacks. An attack succeeds if the compromised

[1]In fact, one could easily prove that software attestation is in general impossible to achieve against hardware modifications.

$\mathcal{P}$ device passes the attestation protocol despite presence of malicious code or data.

We note that ROM is not unusual in commodity embedded systems. For example, the Atmel ATMEGA128 micro-controller allows a small portion of its flash memory to be designated as read-only. Writing to this memory portion via software becomes impossible and can only be enabled by physically accessing the micro-controller with an external debugger.

As in prior attestation literature, [10, 23, 26–29, 31, 35], we assume that the compromised prover device does not have any *real time* help. In other words, during attestation, it does not communicate with any other party, except the verifier. Put another way, the adversary maintains complete radio silence during attestation. In all other respects, the adversary's power is assumed to be unlimited.

4 Design Rationale

Our design rationale is simple and based on three premises:

- **First**, we broaden our scope beyond attestation, to include both secure memory erasure and secure code update. In the event that the updated code is the same as the prior code, secure code update yields secure code attestation. We thus consider secure code update to be a more general primitive than attestation.
- **Second**, we consider two ways of obtaining secure code update: (1) download new code to the device and then perform code attestation, or, (2) securely erase everything on the device and then download new code. The former brings us right back to the problematic software-based attestation, while the latter translates into a simpler problem of secure memory erasure, followed by the download of the new code. We naturally choose the latter.
  Correctness of this approach is intuitive: since the prover's memory is strictly limited, its secure erasure implies that no prior data or code is resident; except for a small amount of code in ROM, which is immutable. Because the adversary is assumed to be passive during code update, download of new code always succeeds, barring any communication errors.
- **Third**, based on the above, we do not aim to *detect* the presence of any malicious code or extraneous data on the prover. Instead, our goal is to make sure that, after erasure or secure code update, no malicious code or extraneous data remains.

Because our approach entails secure erasure of *all* memory, followed by the code download, it might appear to be very inefficient. However, as discussed in subsequent sections, we use the aforementioned approach as a base case that offers unconditional security. Thereafter, we consider ways of improving and optimizing the base case to obtain appreciably more practical solutions.

5 Secure Code Update

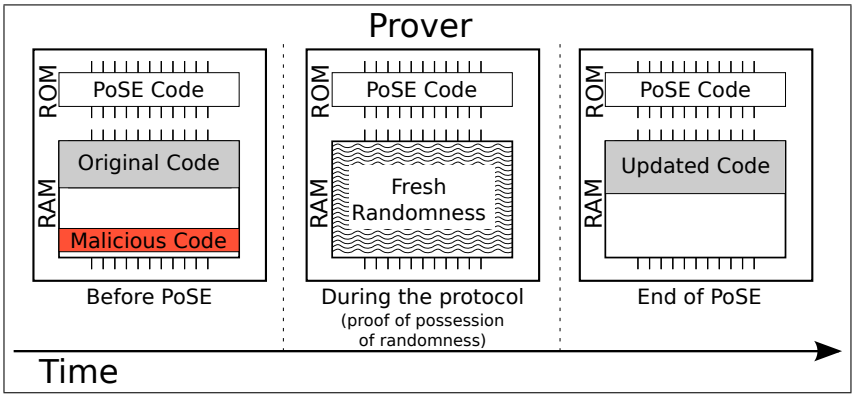

Fig. 2: Prover's Memory during Protocol Execution

The base case for our secure code update approach is depicted in Figure 3. It is essentially a four-round protocol, where:

- Rounds one and two comprise secure erasure of all writable memory contents.
- Rounds three and four represent code update.

Note that there is absolutely no interleaving between any adjacent rounds. The "evolution" of prover's memory during the protocol is shown in Figure 2.

$$
\begin{array}{lllll}
[1] & \mathcal{P} & \longleftarrow & \mathcal{V} & : \; R_1, ..., R_n \\
[2] & \mathcal{P} & \longrightarrow & \mathcal{V} & : \; R_1, ..., R_n \\
[3] & \mathcal{P} & \longleftarrow & \mathcal{V} & : \; C_1, ..., C_n \\
[4] & \mathcal{P} & \longrightarrow & \mathcal{V} & : \; \text{ACK or } H(C_1, ..., C_n)
\end{array}
$$

Fig. 3: Base Case Protocol

As mentioned earlier, we assume a small ROM unit on the prover. In the base case, ROM houses two functions: read-and-send and receive-and-write. During round one, receive-and-write is used to receive a random bit R_i and write it in location $M[i]$, for $0 \leq \ i < n$. At round two, read-and-send reads a bit from location $M[i]$ and sends it to the prover, for $0 \leq \ i < n$. (In practice, read and

write operations involve *words* and not individual bits. However, this makes no difference in our description.)

If we assume that the $\mathcal{V} \leftrightarrow \mathcal{P}$ communication channel is lossless and error-free, it suffices for round four to be a simple acknowledgement. Otherwise, round four must be a checksum of the code downloaded in round three. In this case, the checksum routine must reside in ROM; denoted by $H()$ in round four of Figure 3. In the event of an error, the entire procedure is repeated.

5.1 Efficient Proof of Secure Erasure

As shown in Figure 3, secure erasure is achieved by filling prover's memory with verifier-selected randomness, followed by the prover returning the very same randomness to the verifier. On the prover, these two tasks are executed by the ROM-resident read-and-send and receive-and-write functions, respectively.

It is easy to see that, given our assumptions of: i) adversary's software only attacks, ii) prover's fixed-size memory M, iii) no hardware modification of compromised provers, and iv) source of true randomness on the verifier, the proof of secure erasure holds. In fact, the security of erasure is *unconditional*, due to lack of any computational assumptions.

Unfortunately, this simple approach is woefully inefficient as it requires a resource-challenged $\mathcal{P}$ to send and receive n bits. This prompts us to consider whether secure erasure can be achieved by either (1) sending fewer than n bits to $\mathcal{P}$ in round one, or (2) having $\mathcal{P}$ respond with fewer than n bits in round two. We defer (1) to future work. However, if we sacrifice unconditional security, bandwidth in round two can be reduced significantly.

One way to reduce bandwidth is by having $\mathcal{P}$ return a fixed-sized function of entire randomness received in round one. However, choosing this function is not entirely obvious: for example, simply using a cryptographically suitable hash function yields an insecure protocol. Suppose we replace round two with $CHK = H(R_1, ..., R_n)$ where $H()$ is a hash function, e.g., SHA. Then, a malicious $\mathcal{P}$ can start computing CHK in real time, while receiving $R_1, ..., R_n$ during round one, without storing these random values.

An alternative is for $\mathcal{P}$ to compute a MAC (Message Authentication Code) using the last k bits of randomness – received from $\mathcal{V}$ in round one – as the key. (Where k is sufficiently large, i.e., at least 128 bits.) A MAC function can be instantiated using constructs, such as AES CBC-based MAC [7], AES CMAC or HMAC [6] However, minimum code size varies, as discussed in Section 6. In this version of the protocol, the MAC function must be stored in ROM. Clearly, a function with the lowest memory utilization is preferable in order to minimize the amount of working memory that $\mathcal{P}$ needs to reserve for computing MAC-s.
Claim: Assuming a cryptographically strong source of randomness on $\mathcal{V}$ and a cryptographically strong MAC function, the following 2-round protocol achieves secure erasure of all writable memory M on $\mathcal{P}$:

[1]	$\mathcal{P}$	$\longleftarrow$ $\mathcal{V}$	:	$R_1, ..., R_n$ where $K = R_{n-k+1}...R_n$
[2]	$\mathcal{P}$	$\longrightarrow$ $\mathcal{V}$	:	$MAC_K(R_1, ..., R_{n-k})$

where k is the security parameter (bit-size of the MAC key) and K is the k-bit string $R_{n-k+1}, ..., R_n$.

Proof (Sketch): Suppose that malicious code MC occupies $b > 0$ bits and persists in M after completion of the secure code update protocol. Then, during round one, either: (1) some MAC pre-computation was performed and certain bits (at least b) of $R_1, ..., R_{n-k}$ were not stored in M, or (2) the bit-string $R_1, ..., R_{n-k}$ was compressed into a smaller x-bit string ($x < n - k - b$). However, (1) is infeasible since the key K is only communicated to $\mathcal{P}$ at the very end of round one, which precludes any MAC pre-computation. Also, (2) is infeasible since $R_1, ..., R_{n-k}$ is originates from a cryptographically strong source of randomness and its entropy rules out any compression. □

Despite its security and greatly reduced bandwidth overhead, this approach is still computationally costly considering that it requires a MAC to be computed over entire n-bit memory M. One way to alleviate its computational cost is by borrowing a technique from [4] that is designed to obtain a probabilistic proof in a Provable Data Possession (PDP) setting discussed in Section 2.3. The PDP scheme in [4] assumes that data outsourced by $\mathcal{V}$ (client) to $\mathcal{P}$ (server) is partitioned into fixed-size m-bit blocks. $\mathcal{V}$ generates a sequence of t block indices and a one-time key K which are sent to $\mathcal{P}$. The latter is then asked to compute and return a MAC (using K) of the t index blocks. In fact, these t indices are not explicitly transferred to $\mathcal{P}$; instead, $\mathcal{V}$ supplies a random seed from which $\mathcal{P}$ (e.g., using a hash function or a PRF) generates a sequence of indices.

As shown in [4], this technique achieves detection probability of: $\mathcal{P} = 1-(1-\frac{m}{d})^t$ where m is the number of blocks that $\mathcal{V}$ **did not** store (i.e., blocks where malicious code resides), d is the total number of blocks and t is the number of blocks being checked.

Consider a concrete example of a Mica Mote with 128 Kbytes of processor RAM and further 512 Kbytes of data memory, totaling 640 Kbytes. Suppose that block size is 128 bytes and there are thus $5,120$ blocks. If $\frac{m}{d} = 1\%$, i.e., $m = 51$ blocks, with $t = 512$, detection probability amounts to about 99.94%. This represents an acceptable trade-off for applications where the advantage of MAC-ing $\frac{1}{10}$-th of verifier memory outweighs the 0.06% chance of residual malicious code and/or data. Figure 4 plots the probability t for different values of m.

5.2 Optimizing Code Update

Recall that, in the base case of Figure 3, round three corresponds to code update. Although, in practice, code size is likely to be less than n, receiving and storing entire code is a costly step. This motivates the need for shortcuts. Fortunately,

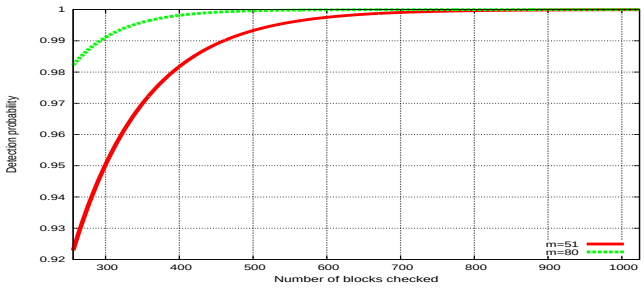

Fig. 4: Probability of detecting memory modifications for # of checked blocks varying between 256 (5%) and 1024 (20%)

there is one effective and obvious shortcut. The main idea is to replace a random $(n-k)$-bit string with the same-length encryption of new code under some key K'. This way, after round two (whether as in the base case or optimized as in the previous section), $\mathcal{V}$ sends K' to $\mathcal{P}$ which uses K' to decrypt the code. The resulting protocol is shown in Figure 5.

[1]	$\mathcal{P}$	$\longleftarrow$	$\mathcal{V}$	: $R_1, ..., R_n$
[2]	$\mathcal{P}$	$\longrightarrow$	$\mathcal{V}$	: $MAC_K(R_1, ..., R_{n-k})$
[3.1]	$\mathcal{P}$	$\longleftarrow$	$\mathcal{V}$	: K'
[3.2]	$\mathcal{P}$: $C_1, ..., C_{n-k} = D_{K'}(R_1, ..., R_{n-k})$, where $D()$ is decryption and $C_1, ..., C_{n-k}$ is new code			
[4]	$\mathcal{P}$	$\longrightarrow$	$\mathcal{V}$	: ACK

Fig. 5: Optimized Protocol

Note again that, since we assume no communication interference and no packet loss or communication errors, the last round is just an acknowledgement, i.e., not a function of decrypted code or K'. This optimization does not affect the security of our scheme if a secure block cipher is used, since encryption of code $[C_1, ..., C_{n-k}]$ with key K' is random and unpredictable to the prover before key K' is disclosed. Hence, the proof in Section 5.1 also holds for this optimized version of the protocol.

6 Implementation and Performance Considerations

In order to estimate its performance and power requirements, we implemented PoSE on the ATMEGA128 micro-controller mounted on a MicaZ sensor. Characteristics of this sensor [11] platform relevant to our scheme are: 648KB total

programmable memory; 250kbps data rate for the wireless communication channel. The total memory is divided into: 128KB of internal flash; 4KB of internal SRAM; 4KB of configuration EEPROM; 512KB of external flash memory. The application was implemented on TinyOS.

6.1 Performance Evaluation

Three main metrics affect the performance of our scheme and for this reason will be evaluated separately: communication speed; read/write memory access time; computation speed of the message authentication code.

Communication channel throughput The maximum claimed throughput of TI-CC2420 radio chip, as reported in the specifications, is 250kbps, which translates to $31,250$ bytes/sec. This upper-bound is unfortunately quite unattainable and our tests show that, in a realistic scenario, throughput hovers around $11,000$ bytes/sec. The total memory available on a MicaZ is 644KB, including external and internal flash and EEPROM. Our efficient proof of erasure only requires randomness to be sent once, from the verifier to the prover. Then a realistic estimate for the transmission time of the randomness amounts to approximately 59 seconds, as was indeed witnessed in our experimental setup.

Memory Access Another important factor in the performance of PoSE is memory access and write time. Write speed on the internal and external flash memory is $60KB/sec$ according to specifications. This estimate has also been confirmed by our experiments. Therefore, memory access accounts for only a small fraction of the total run-time.

MAC Computation We evaluated the performance of three different MAC constructs: HMAC-MD5, HMAC-SHA1 and SkipJack in CBC-MAC. Note that, even though there are well-known attacks on MD5 that find chosen-prefix collisions [33], the short-lived nature of the integrity check needed in our protocol rules out attacks that require 2^{50} calls to the underlying compression function. Table 2(a) shows the results: in each case we timed MAC computation over 644KB of memory on MicaZ.

The fact that MD5 is the fastest is not surprising, given that, in our implementation, the code is heavily in-lined, which reduces the number of context switches for function calls while also resulting in increased code size.

6.2 Memory Usage

We now attempt to estimate the amounts of code and volatile memory needed to run PoSe. An estimate of code memory needed to run it is necessary to understand ROM size requirements. Furthermore, estimating required volatile

memory is critical for the security of the protocol. In fact, in order to correctly follow the protocol, the prover needs a minimal amount of working memory. This memory can not be filled with randomness and hence $\mathcal{P}$ could use it to store arbitrary values. However, by keeping the amount of volatile memory to a *minimum* we can guarantee that $\mathcal{P}$ can not store both arbitrary values and carry on the necessary computation to complete the protocol.

Since assuring that the amount of volatile memory used in a specific implementation is difficult, one way to minimize effects of volatile memory is to include it in the computation of the keyed MAC (or send it back to $\mathcal{V}$ in the base case). Even though the contents of volatile memory are dynamic, they are entirely depended on the inputs from $\mathcal{V}$. Therefore, they are essentially deterministic. In this case, the verifier would have to either simulate or re-run the attestation routine to compute the correct (expected) volatile memory contents.

Table 2: MAC constructions on MicaZ.

(a) Energy consumption and time

MAC	Time (sec)	Energy ($\mu J/byte$)
HMAC-MD5	28.3	1
HMAC-SHA1	95	3.5
Skipjack CBC-MAC	88	3.1

(b) Code and working memory required

MAC	ROM (bytes)	RAM (bytes)
HMAC-MD5	9,728	110
HMAC-SHA1	4,646	124
Skipjack CBC-MAC	2,590	106

Code Size To estimate code size, we implemented the base case PoSE protocol in TinyOS. It transmits and receives over the wireless channel using Active Messages. The entire application takes $11,314$ bytes of code memory and 200 bytes of RAM. RAM is needed to hold the necessary data structures along with the stack. Our implementation used regular TinyOS libraries and compiler. Careful optimization would most likely reduce memory consumption.

In the optimized version of PoSE, we also need a MAC housed in ROM. Table 2(b) shows the amount of additional memory necessary to store code and data for various MAC constructions. Finally, Table 3 shows the size of both code and working memory for all presented above.

The reason for MD5 having a larger memory footprint is because, as discussed above, the implementation we used is highly inlined. While this leads to better performance (faster code) it also results in a bigger code size.

Table 3: Code and volatile memory size.

Protocol	ROM (bytes)	RAM (bytes)
PoSE(Base Case)	11,314	200
PoSE-MD5	21,042	264
PoSE-SHA1	15,960	274
PoSE-SkipJack	13,904	260

Memory Mapping In the previous discussion, we have abstracted away from specific architectures by considering a system with uniformly addressable memory space M. However, in formulating this generalization extra care must be taken: in real systems, memory is not uniform, since there can be regions assigned to specific functions, such as memory-mapped registers or I/O buffers. In the former case, changing these memory locations can result in modified registers which, in turn, might cause unintended side effects. In the latter, memory content of I/O buffers might change due to asynchronous and non-deterministic events, such as reception of a packet from a wireless link. When we refer to prover memory M, we always exclude these special regions of memory. Hence both the verifier and the prover have to know a mapping from the virtual memory M to the real memory. However, this mapping can be very simple, thus not requiring a memory management unit. For example on the Atmel ATMEGA128, as used in the MicaZ, the first 96 bytes of internal SRAM are reserved for memory-mapped register and I/O memory.

6.3 Read-Only Memory

PoSE needs a sufficient amount of read-only memory (ROM) to store the routines (read-and-send, receive-and-write and, in its optimized version, MAC) needed to run the protocol. While the use of mask ROM has always been prominent in embedded devices, recently, due to easier configuration, flash memory has supplanted cheaper mask ROM.

However, there are other means to obtain read-only memory using different and widely available technologies. For example, ATMEGA128 [5] allows a portion of its flash memory to be *locked* in order to prevent overwriting. Even though the size of this lockable portion of memory is limited to 4KB, this feature shows the feasibility of such an approach on current embedded devices. Note that, once locked, the memory portion cannot be unlocked unless an external JTAG debugger is attached to unset the lock bit.

Moreover, ATMEGA128 has so-called *fuse bits* that, once set, cannot be restored without unpacking the MCU and restoring the fuse. This clearly illustrates that the functionalities needed to have secure read-only memory are already present in commodity hardware.

Another way to achieve the same goal would be to use one-time programmable (OTP) memory. Although this memory is less expensive than flash, it still offers some flexibility over conventional ROM.

7 Limitations and Challenges

In this paper, our design was guided mainly by the need to obtain clear security guarantees and not to maximize efficiency and performance. Specifically, we aimed to explore whether remote attestation without secure hardware is possible at all. Hence, PoSE-based protocols (even the optimized ones) have certain performance drawbacks. In particular, the first protocol round is the most resource-consuming part of all proposed protocols. The need to transmit, receive and write n bits is quite expensive. It remains to be investigated whether it is possible to achieve same security guarantees with a more efficient design.

In terms of provable security, our discussion of Proofs-of-Secure-Erasure (PoSE-s) has been rather light-weight. A more formal treatment of the PoSE primitive needs to be undertaken. (The same holds for the multi-prover extension described in Appendix A).

Furthermore, we have side-stepped the issue of verifier authentication. However, in practice, $\mathcal{V}$ must certainly authenticate itself to $\mathcal{P}$ before engaging in any PoSE-like protocol. This would entail additional requirements (e.g., storage of $\mathcal{V}$'s public key in $\mathcal{P}$'s ROM) and raise new issues, such as exactly how (possibly compromised) $\mathcal{P}$ can authenticate $\mathcal{V}$?

Another future direction for improving our present work is by giving the adversary the capability of attacking our protocol with another device (not just the actual prover). This device would try to aid the prover in computing the correct responses in the protocol and pass the PoSE. Assuming wireless communication, one way for verifier to prevent the prover from communicating with another malicious device is is by actively jamming the prover.

Jamming can be used to selectively allow the prover to complete the protocol, while preventing it from communicating with any other party. Any attempt to circumvent jamming by increasing transmission power can be limited by using readily available hardware. For example, the CC2420 radio, present on the MicaZ, supports transmission power control. Thresholds can be set for the Received Signal Strength (RSS), RSS_{min} and RSS_{max}, such that only frames with $RSS \in [RSS_{min}, RSS_{max}]$ are accepted and processed. This is enforced in hardware by the radio chip. Hence, if the verifier wants to make sure that the prover does not communicate, it can simply emit a signal with $RSS > RSS_{max}$. This approach is similar to the one employed in [22], albeit, in a different setting.

8 Conclusions

This paper considered secure erasure, secure code update and remote attestation in the context of embedded devices. Having examined prior attestation ap-

proaches (both hardware- and software-based), we concluded that the former is too expensive, while the latter – too uncertain. We then explored an alternative approach that generalized the attestation problem to remote code update and secure erasure. Our approach, based on Proofs-of-Secure-Erasure relies neither on secure hardware nor on tight timing constraints. Moreover, although not particularly efficient, it is viable, secure and offers some promise for the future. We also assess the feasibility of the proposed method in the context of commodity sensors.

Acknowledgments

We thank ESORICS'10 anonymous reviewers for their comments. We are also grateful to Ivan Martinovic, Claude Castelluccia, Aurelien Francillon and Brian Parno for their comments on early drafts of this paper. Research presented in this paper was supported, in part, by the European Commission-funded STREP WSAN4CIP project, under grant agreement ICT-225186. All views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsement of the WSAN4CIP project or the European Commission.

References

1. Anderson, R., and Kuhn, M. Tamper resistance - a cautionary note. In *In Proceedings of the Second Usenix Workshop on Electronic Commerce* (1996).
2. Arbaugh, W. A., Farber, D. J., and Smith, J. M. A secure and reliable bootstrap architecture. In *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 1997), IEEE Computer Society, p. 65.
3. Ateniese, G., Burns, R., Curtmola, R., Herring, J., Kissner, L., Peterson, Z., and Song, D. Provable data possession at untrusted stores. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, 2007), ACM, pp. 598–609.
4. Ateniese, G., Di Pietro, R., Mancini, L. V., and Tsudik, G. Scalable and efficient provable data possession. In *SecureComm '08: Proceedings of the 4th international conference on Security and privacy in communication networks* (New York, NY, USA, 2008), ACM, pp. 1–10.
5. Atmel Corporation. Atmega128 datasheet. `http://www.atmel.com/atmel/acrobat/doc2467.pdf`.
6. Bellare, M., Canetti, R., and Krawczyk, H. Keying hash functions for message authentication. In *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology* (London, UK, 1996), Springer-Verlag, pp. 1–15.
7. Bellare, M., Kilian, J., and Rogaway, P. The security of cipher block chaining. In *CRYPTO '94: Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology* (London, UK, 1994), Springer-Verlag, pp. 341–358.

8. Cachin, C., and Maurer, U. Unconditional security against memory-bounded adversaries. In *In Advances in Cryptology CRYPTO 97, Lecture Notes in Computer Science* (1997), Springer-Verlag, pp. 292–306.
9. Castelluccia, C., Francillon, A., Perito, D., and Soriente, C. On the difficulty of software-based attestation of embedded devices. In *CCS 09: Proceedings of 16th ACM Conference on Computer and Communications Security* (November 2009).
10. Choi, Y.-G., Kang, J., and Nyang, D. Proactive code verification protocol in wireless sensor network. In *ICCSA 07: Proceedings of the International Conference on Computational Science and Its Applications* (2007), O. Gervasi and M. L. Gavrilova, Eds., vol. 4706 of *Lecture Notes in Computer Science*, Springer.
11. Crossbow Technology Inc. Micaz datasheet. `http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf`.
12. England, P., Lampson, B., Manferdelli, J., Peinado, M., and Willman, B. A trusted open platform. *IEEE Computer 36*, 7 (2003).
13. Flammini, F., Gaglione, A., Mazzocca, N., Moscato, V., and Pragliola, C. Wireless sensor data fusion for critical infrastructure security. In *CISIS 08: Proceedings of the International Workshop on Computational Intelligence in Security for Information Systems* (October 2008).
14. Francillon, A., and Castelluccia, C. Code injection attacks on Harvard-architecture devices. In *CCS 08: Proceedings of the 15th ACM Conference on Computer and Communications Security* (2008), P. Ning, P. F. Syverson, and S. Jha, Eds., ACM.
15. Goodspeed, T. Exploiting wireless sensor networks over 802.15.4. In *Texas Instruments Developper Conference* (2008).
16. Gratzer, V., and Naccache, D. Alien vs. quine. *IEEE Security and Privacy 5* (2007), 26–31.
17. Hu, W., Corke, P., Shih, W. C., and Overs, L. secfleck: A public key technology platform for wireless sensor networks. In *EWSN* (2009), vol. 5432 of *Lecture Notes in Computer Science*, Springer.
18. Jakobsson, M., and Johansson, K.-A. Assured detection of malware with applications to mobile platforms. Tech. rep., DIMACS, February 2010. available at http://dimacs.rutgers.edu/TechnicalReports/TechReports/2010/2010-03.pdf.
19. Juels, A., and Kaliski, Jr., B. S. Pors: proofs of retrievability for large files. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, 2007), ACM, pp. 584–597.
20. Kennell, R., and Jamieson, L. H. Establishing the genuinity of remote computer systems. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2003), USENIX Association, pp. 21–21.
21. Kil, C., Sezer, E. C., Azab, A. M., Ning, P., and Zhang, X. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In *DSN 09: Proceedings of the 39th IEEE/IFIP Conference on Dependable Systems and Networks* (June 2009).
22. Martinovic, I., Pichota, P., and Schmitt, J. B. Jamming for good: a fresh approach to authentic communication in wsns. In *WiSec '09: Proceedings of the second ACM conference on Wireless network security* (New York, NY, USA, 2009), ACM, pp. 161–168.
23. Park, T., and Shin, K. G. Soft tamper-proofing via program integrity verification in wireless sensor networks. *IEEE Trans. Mob. Comput. 4*, 3 (2005).

24. Roman, R., Alcaraz, C., and Lopez, J. The role of wireless sensor networks in the area of critical information infrastructure protection. *Inf. Secur. Tech. Rep. 12*, 1 (2007), 24–31.
25. Sailer, R., Zhang, X., Jaeger, T., and van Doorn, L. Design and implementation of a tcg-based integrity measurement architecture. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2004), USENIX Association, pp. 16–16.
26. Seshadri, A., Luk, M., and Perrig, A. SAKE: Software attestation for key establishment in sensor networks. In *DCOSS '08: Proceedings of the 4th IEEE international conference on Distributed Computing in Sensor Systems* (2008).
27. Seshadri, A., Luk, M., Perrig, A., van Doorn, L., and Khosla, P. SCUBA: Secure code update by attestation in sensor networks. In *WiSe '06: Proceedings of the 5th ACM workshop on Wireless security* (2006), ACM.
28. Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., and Khosla, P. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles* (2005), ACM.
29. Seshadri, A., Perrig, A., van Doorn, L., and Khosla, P. K. SWATT: SoftWare-based ATTestation for embedded devices. In *IEEE Symposium on Security and Privacy* (2004), IEEE Computer Society.
30. Shacham, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS '07: Proceedings of the 14th ACM conference on Computer and Communications Security* (2007), ACM.
31. Shaneck, M., Mahadevan, K., Kher, V., and Kim, Y. Remote software-based attestation for wireless sensors. In *ESAS* (2005).
32. Shankar, U., Chew, M., and Tygar, J. D. Side effects are not sufficient to authenticate software. In *Proceedings of the 13th USENIX Security Symposium* (August 2004).
33. Stevens, M., Lenstra, A., and Weger, B. Chosen-prefix collisions for md5 and colliding x.509 certificates for different identities. In *EUROCRYPT '07: Proceedings of the 26th annual international conference on Advances in Cryptology* (Berlin, Heidelberg, 2007), Springer-Verlag, pp. 1–22.
34. Trusted Computing Group. Specifications.
35. Yang, Y., Wang, X., Zhu, S., and Cao, G. Distributed software-based attestation for node compromise detection in sensor networks. In *SRDS* (2007), IEEE Computer Society.

A: Dealing with Multiple Devices

Thus far, in this paper we assumed one-on-one operation: one prover and one verifier. However, in practice, embedded devices are often deployed in groups and in relative proximity (and density) among them, e.g., Wireless Sensor Networks (WSNs). If the task at hand is to perform code attestation or update of multiple proximate devices, our approach can be easily extended to support this setting and, at the same time, obtain a significant efficiency gain. The main observation is that, if the verifier can communicate with $t > 1$ devices at the same time (i.e., via broadcast), download of randomness in the first round of our protocol

– which represents the most time-consuming part of the protocol – can be done in parallel for all devices within the verifier's communication range. Of course, in order to receive replies the verifier has to be within communication range of all t provers.

At the same time, parallel code update of multiple devices prompts us to re-examine the adversarial model. In the one-on-one setting, it makes sense to assume radio silence, i.e., the fact that, during the protocol, the prover device is not communicating with any party other than the verifier, and no other (third) device is transmitting any information that can be received by either the prover or the verifier. Note that the term *adversary* refers collectively to any compromised devices running malicious code as well as any extraneous devices physically controlled by the adversary. However, the one-on-one setting does not preclude the adversary from *over-hearing* communication between the prover and the verifier, i.e., eavesdropping on protocol messages. We claim that this has no bearing on security, since each protocol involves a distinct stream of randomness.

In contrast, when multiple parallel (simultaneous) provers are involved, the situation changes. In particular, we need to take into account that possibility that one or more of the t provers is running malicious code. Suppose that a malicious code-running prover $\mathcal{P}_x$. Then, if we naïvely modify our protocol from Figure 5 as shown in Figure 6, the resulting protocol is insecure. The reason for the lack of security is simple: suppose that $\mathcal{P}_x$ ignores the message in round 1.0 and does not store verifier-supplied randomness. Then, in round 2.0, $\mathcal{P}_x$ over-hears and records a reply – $MAC_K(R_1, ..., R_{n-k})$ – from an honest prover $\mathcal{P}_1$. Clearly, $\mathcal{P}_x$ can just replay this MAC and thus convince the verifier of having received and stored the randomness from message 1.0.

Assume reachable provers $\mathcal{P}_1, ..., \mathcal{P}_t$ and $1 < j \leq t$

[1.0] $\mathcal{P}_j \Longleftarrow \mathcal{V}$: $R_1, ..., R_n$ where $K = R_{n-k}, ..., R_n$
[2.0] $\mathcal{P}_j \longrightarrow \mathcal{V}$: $MAC_K(R_1, ..., R_{n-k})$
[3.1] $\mathcal{P}_j \Longleftarrow \mathcal{V}$: k'
[3.2] $\mathcal{P}_j$: $C_1, ..., C_{n-k} = D_{K'}(R_1, ..., R_{n-k})$
[4.0] $\mathcal{P}_j \longrightarrow \mathcal{V}$: ACK

Fig. 6: Insecure Multi-Prover Protocol

The above discussion leads us to amend the adversarial model as follows: the adversary is allowed to record any portion of the protocol. However, for fear of being detected, it is not allowed to transmit anything that is not part of the protocol. In particular, during the protocol, none of the (potentially compromised) t provers can transmit anything that is not part of the protocol. And, no extraneous entity can transmit anything to any of the t provers.

Assume reachable provers $\mathcal{P}_1, ..., \mathcal{P}_t$ and $1 < j \leq t$

[1.1] $\mathcal{P}_j \Longleftarrow \mathcal{V} : R_1, ..., R_n - k$
[1.2] $\mathcal{P}_j \longleftarrow \mathcal{V} : K_j$ where $k = |K_j|$
[2.0] $\mathcal{P}_j \longrightarrow \mathcal{V} : MAC_{K_j}(R_1, ..., R_{n-k})$
[3.1] $\mathcal{P}_j \Longleftarrow \mathcal{V} : K'$
[3.2] $\mathcal{P}_j : C_1, ..., C_{n-k} = D_{K'}(R_1, ..., R_{n-k})$
[4.0] $\mathcal{P}_j \longrightarrow \mathcal{V} : \mathrm{A}CK$

Fig. 7: Multi-Prover Protocol

The modified (and secure) protocol that supports $t > 1$ provers is shown in Figure 7. The main difference from the insecure version in Figure 6 is the fact that random and distinct keys K_j are generated and sent to each prover $\mathcal{P}_j$.

This protocol guarantees that, in the context of the modified adversarial model, each prover has to independently store the randomness sent by the verifier. Since, the key sent by the verifier is unique to each prover and so is the MAC computation. This assertion clearly needs to be substantiated via a proof of security. This issue will be addressed as part of our future work.

Caveat: We acknowledge that, while the multi-prover protocol achieves better performance through parallelization, it does not improve energy consumption on each prover. We plan to explore this issue as part of our future work.