On the Difficulty of Software-Based Attestation of Embedded Devices

Claude Castelluccia, Aurélien Francillon, Daniele Perito
INRIA Rhône-Alpes
{ccastel,francill,perito}@inrialpes.fr

Claudio Soriente
University of California, Irvine
csorient@ics.uci.edu

ABSTRACT

Device attestation is an essential feature in many security protocols and applications. The lack of dedicated hardware and the impossibility to physically access devices to be attested, makes attestation of embedded devices, in applications such as Wireless Sensor Networks, a prominent challenge. Several software-based attestation techniques have been proposed that either rely on tight time constraints or on the lack of free space to store malicious code. This paper investigates the shortcomings of existing software-based attestation techniques. We first present two generic attacks, one based on a *return oriented rootkit* and the other on code compression. We further describe specific attacks on two existing proposals, namely SWATT and ICE-based schemes, and argue about the difficulty of fixing them. All attacks presented in this paper were implemented and validated on commodity sensors.

Categories and Subject Descriptors

K.6.5 [**Operating Systems**]: Security and Protection

General Terms

Experimentation,Security

Keywords

Software-Based Attestation, Return Oriented Programming, Code Compression, Embedded Systems, Wireless Sensor Networks, Indisputable Code Execution, SWATT

1. INTRODUCTION

Embedded systems are employed in several critical environments where correct operation is an important requirement. Malicious nodes in a Wireless Sensor Network (WSN) can be used to disrupt the network operation by deviating from the prescribed protocol or to launch internal attacks. Preventing node compromise is difficult; it is therefore desirable to detect compromised nodes to isolate them from the network. This is performed through *code attestation*, i.e., the base station verifies that each of the nodes is still running the initial application and, hence, has not been compromised. Attestation techniques based on tamper-resistant hardware [7], while possible [13] are not generally available, nor are foreseen to be cost effective for lightweight WSNs nodes.

Software-based attestation [22, 25, 27] is a promising solution for verifying the trustworthiness of inexpensive, resource constrained sensors, because it does not require dedicated hardware, nor physical access to the device.

Previously proposed techniques are based on a challenge-response paradigm. In this paradigm, the verifier (usually the base station) challenges a prover (a target device) to compute a checksum of its memory. The prover either computes the checksum using a fixed integrity verification routine or downloads it from the verifier right before running the protocol. In practice, memory words are read and incrementally loaded to the checksum computation routine. To prevent replay or pre-computation attacks, the verifier challenges the prover with a nonce to be included in the checksum computation. Since the verifier is assumed to know the exact memory contents and hardware configuration of the prover, it can compute the expected response and compare it with the received one. If values match, the node is genuine, otherwise, it has most likely been compromised.

Contributions.

This paper highlights shortcomings of several attestation techniques for embedded devices and shows practical attacks against them. First, we present a *Rootkit* for embedded systems – a malicious program that allows a permanent and undetectable presence on a system [12] – that circumvents attestation by hiding itself in non-executable memories. The implementation of this attack uses a technique called *Return Oriented Programming* (ROP) [26, 4], a generalization of return-into-libc [29, 19, 17]. ROP can be used by the adversary to compromise the node and perform arbitrary computations without injecting code. Node compromise is achieved by reusing and controlling pieces of code already present in the device's memory. Second, we present an attack that uses code compression to free memory space which can be used to hide malicious code.

We then describe some specific attacks against previously proposed attestation protocols, ultimately showing the difficulty of software-based attestation design.

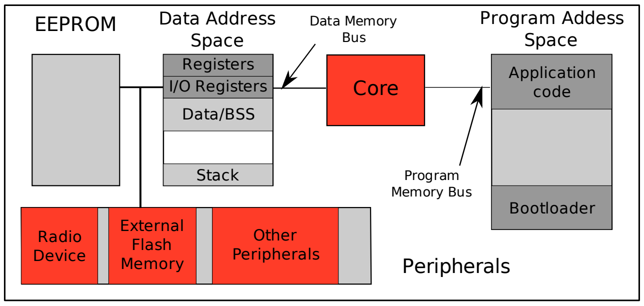

Figure 1: Overview of memories on a MicaZ node; the EEPROM and external memories are accessed from the I/O Registers.

Organization.

Section 2 introduces assumptions and surveys relevant work in the area of attestation for embedded devices. Section 3 presents two generic attacks that highlight flaws in several existing protocols, while Section 4 introduces details of attacks implemented against SWATT [25] and ICE [21]. The paper concludes in Section 5.

2. ASSUMPTIONS AND OVERVIEW OF PREVIOUS WORK

2.1 Assumptions

Hardware platform description.

Throughout the paper we use the MicaZ, an off-the-shelf wireless sensor node. The MicaZ is an Atmel AVR based device with a Harvard memory architecture. Its memory layout is depicted in Figure 1, which includes *program memory*, *data memory* and *external memory*. Program memory is a flash memory that contains the application running on the sensor as well as the *bootloader*. The latter is a minimal program that is usually present on most devices to allow remote code update. Code updates are often required when, for example, a vulnerability is found and physically maintenance is not an option. Actually, most embedded devices are equipped with a bootloader [9], since devices without self-reprogramming capability would have limited value.

The data memory contains the stack and statically allocated variables (Data sections) as well as CPU and I/O registers. The external memory is used to store data collected from the environment.

While the presented attacks are validated on an experimental platform composed of wireless sensor nodes, they are not specific to WSNs. They exploit the characteristics of the micro-controller and device hardware. Proposed attacks are applicable to any embedded device that uses a similar micro-controller and communicates via an open channel. For example, they could be applied to constrained systems embedded in cars [24], home automation and Advanced Metering Infrastructure (AMI) devices.

Adversary model.

As in other proposals [5, 20, 21, 22, 23, 25, 27, 32], the envisioned adversary has the objective of installing its malicious code in an executable memory of the target device and passing the attestation protocol without being detected.

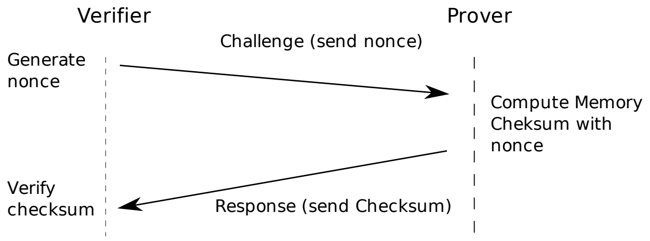

Figure 2: Basic attestation challenge response protocol

Before attestation, the attacker has full control over all device memories. It is therefore able to modify program and data memory or any other memories on the platform. However, we assume that at attestation time, while the malicious code is still running, the attacker has no direct control on the device anymore. The attack succeeds if the device passes the attestation protocol despite the presence of the malicious code.

How the attacker installs its code on the device is beyond the scope of this paper and is not discussed in detail. Malicious code installation could be performed via remote exploitation of a software vulnerability [9, 10, 11], a non invasive hardware attack [2] or simply using an off-the-shelf JTAG programming adapter, if the feature is activated[1]. Yet another possibility would be to use a non authenticated or vulnerable code update mechanism.

Like in other proposals, we assume that the attested device cannot collude with malicious peers. This could be enforced, for example, by restricting network access and discarding the result of the attestation if suspicious network activity is detected. Finally, we assume that the attacker does not modify the device hardware. It is also assumed that the verifier knows the hardware and memory configuration of the prover.

2.2 Software-based code attestation

All of the existing software-based attestation techniques are based on a challenge-response paradigm where the verifier (usually the base station) challenges a prover (a target device) to compute a checksum of its memory.

This section describes the basic challenge-response protocol and then presents how it is used by the existing software-based attestation schemes.

2.2.1 Challenge-response protocol

A challenge-response attestation routine uses a suitable checksum function $\mathcal{H}(\cdot)$ to compute the checksum of the attested memory. A nonce provided by the verifier (Figure 2) is used as the first input to $\mathcal{H}(\cdot)$; then memory words are sequentially read (from the first to the last) and incrementally input to the function. The output of the last iteration of the function is the result of the attestation. The nonce provided by the verifier prevents pre-computation or replay attacks. Alternatively, the sequence of input memory words can be determined by a pseudo-random number generator, initialized with a seed provided by the verifier. In this case, to make sure that all memory words are used in the computation of the checksum with high probability, the number

[1]JTAG access can be deactivated before deployment, yet it is often left active.

of memory accesses increases from n to $n \ln(n)$, where n is the total number of memory words[2]. Pre-computation or replay attacks are prevented because it is not feasible for the attacker to guess the seed ahead of time and learn the sequence in which memory words are going to be input to $\mathcal{H}(\cdot)$.

2.2.2 Existing Proposals

SWATT.

SoftWare-based ATTestation (SWATT) by Seshadri et al. [25] relies on timing of sensor responses to identify compromised nodes. In SWATT, the program memory is attested by reading memory words in a pseudo-random fashion, using a nonce provided by the verifier. If a compromised prover runs a modified version of the original code, some (or all) memory accesses must be redirected to memory locations where the original code words are, in order to compute a valid response. The authors claim that the overhead caused by redirection would be easily detected by the verifier. They claim to have implemented the fastest checksum function and to have considered the fastest redirection routine and show that it would still introduce a considerable overhead to checksum computation. Section 4.1.1 presents an implementation of redirection that is faster than the one presented in [25], showing how difficult it is to design an attestation protocol based on tight timing constraints. Moreover, as SWATT does not attest data memory nor external storage, the prover could store malicious code in one of those memories and restore it after attestation using ROP (Section 3.1).

Filling empty program memory.

The authors of [32] introduce a protocol where sensors collaborate to attest the code authenticity of their peers. In their proposal, the free program memory space of each sensor is filled with randomness before deployment. The authors claim that if the whole program memory is verified, the adversary would have no empty space to store its malware, unless it deletes parts of the original memory contents (code or random data). Section 3.2 shows that an attacker can compress the original code in program memory and gain enough free space to store and run its malicious program. As in SWATT, this protocol considers only program memory.

Choi et al. [5] take a similar approach to make sure that the prover is left with no space where to store the malicious code at attestation time. In their protocol, the prover uses a random seed provided by the verifier to produce a pseudo-random bitstream and uses it to fill the empty memory locations. Hence, security is based on the prover's compliance to the protocol. A malicious node would rather deviate from the original protocol, still trying to produce a valid response. This could be achieved, for example, by generating random bytes on the fly (e.g. using time-memory trade-offs), instead of storing them in the program memory. Finally, as in previous protocols, the authors consider only program memory.

Self-modifying code based attestation.

Shaneck et al. [27] perform attestation transferring the attestation code from the base station to the sensor at attestation time. The authors assume that the adversary is not aware of the attestation code and that the latter uses obfuscated predicates to prevent static code analysis. The protocol relies on the use of self modifying code to prevent analysis and modifications before the attestation code is run. Self-modifying code is notoriously difficult to implement and is therefore a questionable design choice for an attestation protocol. Moreover, most embedded systems have their program memory on a flash memory which is usually programmable by pages, after a page erase. It will therefore be slow and complex, if not impossible, to implement self modifying code on a flash based device[3].

ICE.

Indisputable Code Execution (ICE) based schemes [22, 23, 21] rely on an attestation procedure being performed on the attestation routine itself, including the program counter in the computation. The idea behind it, is to prevent the adversary from mounting an attack where a modified attestation routine located at a different place in memory is run. Unfortunately, not all platforms make the program counter available to software. This is the case, for example, of the AVR family of micro-controllers [4] used on MicaZ devices. Porting ICE on this family of processors would require complex changes or would just not be feasible. Additionally, Section 4.2 shows that weaknesses in the checksum function can be abused to mount a practical attack.

3. TWO GENERIC ATTACKS ON CODE ATTESTATION PROTOCOLS

This Section introduces two attacks that are applicable to several software-based code attestation protocols.

The first attack circumvents malware detection by moving malicious code between program memory and non-executable memory, during the code attestation procedure. This is achieved using a technique called *Return Oriented Programming.* The second attack uses code compression to free space in the program memory in order to hide the malicious code.

3.1 A Rootkit-based attack

Recent work [26, 4] showed that *Return Oriented Programming* can be used to maliciously execute *legitimate* pieces of code on a system, even within the constraints imposed by embedded systems [9]. These pieces of code are called *gadgets* and are sequences of instructions terminated by a `return` instruction. By crafting a stack and carefully controlling its return addresses an adversary can perform arbitrary computations[5]. As a consequence, in order to determine the correct behavior of a device, it is not sufficient to verify the correctness of its code.

While ROP has been initially introduced to perform arbitrary computations without injecting code and hence *gain* control over a system, we demonstrate that it can also be

[2]Using the Coupon Collector's Problem.

[3]The MSP430 based Telosb mote with a Von Neumann memory architecture can execute code present in data memory. This makes self-modifying code easier to implement. However, the AVR based Mica family of motes can only execute instructions from the flash memory.

[4]MIPS and 8051 suffer from the same limitation.

[5]As the malicious code has complete control over the data memory, techniques such as memory safety [6] and stack canaries cannot prevent the usage of ROP. However, we notice it could be prevented by *Control Flow Integrity* [1, 8, 31].

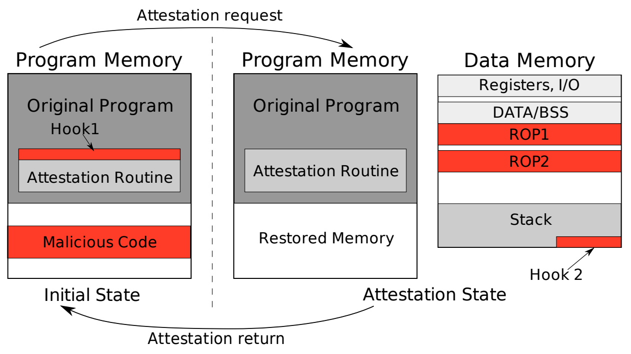

Figure 3: Return Oriented Programming attack.

used to implement a rootkit. We show that ROP can be used to hide malware on an embedded system, and prevent its detection during the attestation procedure. We also show that ROP can be used to restore the malware after the attestation procedure to re-gain control of the compromised device.

The rootkit hiding code has been implemented on a MicaZ sensor and only uses the instructions present in the device bootloader. It works by inserting a hook (a jump instruction) into the attestation routine. Upon attestation, the hook triggers the rootkit hiding functionality that deletes the rootkit code from the program memory. In practice, the rootkit deletes its code from program memory executing instructions (using ROP) stored in the bootloader. ROP is also used, once attestation is completed, to re-install the rootkit and re-gain control over the device.

Figure 4 presents a generic attestation function. In our prototype, we insert a hook to the rootkit bootstrap code, by replacing the first instruction of the attestation function with a `jump`. When the latter is invoked the hook transfers execution to the rootkit bootstrap code which deletes malicious content (including itself) from the program memory. It then returns to the attestation code that runs on a clean program memory. Once attestation is over, the rootkit restores itself into program memory using ROP.

3.1.1 Rootkit description

Our rootkit requires two hooks: one in the program memory at the beginning of the attestation routine and one in the data memory after the attestation function returns (Figure 3). It is composed of different parts:

Rootkit bootstrap code: the code used to hide and restore the malicious payload and itself from program memory.

Rootkit payload: the malicious code, i.e. the malware.

Program memory hook: the hook installed in the function receiving the attestation request message. Hooking is performed by replacing the first instruction of the `receive_checksum_request` function with a jump to the rootkit, so that the latter is called at each attestation request.

Data memory hook: the second hook bootstraps the ROP that restores the rootkit in program memory. This hook can not be included in program memory (e.g. at the end of the `receive_checksum_request` function) without being detected by the verifier. Therefore, it is added in the stack, replacing the stored return address of the `receive_checksum_request` function.

```
void receive_checksum_request(uint8_t nonce){
    uint8_t checksum[8];
    prepare_checksum(nonce);
    do_checksum(checksum);
    send(checksum);
    return;
}
```

Figure 4: Example of attestation function.

Return oriented programs: the ROP used to move the rootkit hiding code is composed of two gadgets chains (or return oriented programs) in the data memory: the first (ROP1) is used before attestation to erase the rootkit and restore the original program memory contents, while the second one (ROP2) is used after attestation to restore the rootkit.

3.1.2 Attack description

When an attestation request is received (*Initial State* in Figure 3), the *rootkit bootstrap code* is called by the program memory hook and performs the following operations:

1. Copies ROP1 to data memory. The *rootkit bootstrap code* is 620 bytes, hence it requires the gadget chain to use 3 reprogramming gadgets [9], each of them containing 256 bytes of the code to be restored. A fourth reprogramming gadget is used to restore the program memory page where the program memory hook is present.

2. Writes ROP2 to data memory.

3. Copies the malicious payload to EEPROM (about 1 KByte) where it will be temporarily stored until attestation ends.

4. Places the data memory hook by changing the return address (saved on stack) of the `receive_checksum_request` function.

5. Removes the malicious payload from program memory.

6. Executes the first gadget chain (*ROP1*) that restores the program memory in its original state. Upon completion ROP1 returns to the address of the `receive_checksum_request` function.

The attestation function is therefore executed over a clean program memory (*Attestation State* in Figure 3). Its output is sent to the verifier and the `receive_checksum_request` function returns. When the final `ret` instruction of this function is executed the following actions are triggered:

1. The data memory hook launches the restoration gadget chain (*ROP2*).

2. The restoration gadget chain writes the *rootkit bootstrap code* to program memory and executes it.

3. The *rootkit bootstrap code* copies the malicious payload code from EEPROM to program memory.

4. The malicious payload re-installs the program memory hook in the `receive_checksum_request` function; the device is therefore ready to receive a new attestation request (*Initial State*).

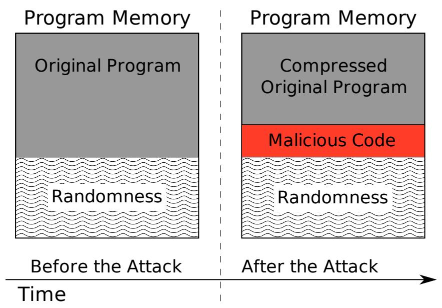

Figure 5: Compression Attack.

Application	Size	Compression Gain (Bytes)		
	(Bytes)	Huffman	Gzip	PPM
6LowPan Cli	23982	2669	8667	10180
Base Station	15778	1858	5400	7029
Oscilloscope	13276	1679	4740	6091
” Multi-hop	31836	4208	14241	16948
” Multi-hopLqi	23848	2952	9311	11611
Sense	2950	252	484	1124
Avg Gain (B)	-	2269	7186	8830
Avg Gain (%)	-	12.19	38.61	47.45

Table 1: Compression results for Micaz applications (similar results where found for TelosB applications).

3.1.3 Experimental results

The rootkit detailed above was implemented on a MicaZ device and used to attack the attestation protocol presented in [25]. The entire rootkit fits in 1.6 KBytes, of which, only 620 bytes need to be stored in data memory to be re-installed with ROP. The remaining code is restored from EEPROM (or external flash memory). In our experimental setup, the time required to the rootkit to hide itself is less than 50 milliseconds (Figure 7). Compared to the running time of SWATT, introduced overhead is about 0.3%.

3.1.4 Discussion

In our implementation the attack relies on a single reprogramming meta-gadget that is composed of more than a 100 instructions. Therefore, it does not require a Turing complete gadget set. It uses instructions that manipulate the code memory and that are very likely to be found in devices that are equipped with a bootloader. Additionally, as this reprogramming meta-gadget is a part of the default TinyOS bootloader, it is independent of the application executed on the device. The presence of this reprogramming meta-gadget in the bootloader is sufficient to mount the attack. Nevertheless, the availability of a Turing complete gadget set would probably make the attack easier to implement.

	Sequential Access		Random Access	
Compression Algorithm	Time (Sec)	Freed Space (Bytes)	Time (Sec)	Freed Space (Bytes)
Huffman	6	2220	269	1252
None	1	-	145	-

Table 2: Compression Attack, using Canonical Huffman encoding.

3.2 Compression attack

Common sensor applications are appreciably smaller than the available program memory[6]. Empty memory locations contain a fixed value, i.e. `0xFF`, which is the default state of non-programmed flash memory. Even if those locations are considered for attestation, an adversary could just write them with arbitrary data and “remember” the original value when it is requested by the attestation routine.

Previously proposed schemes [32, 7] tried to prevent malicious empty memory usage, filling it with pseudo-random values at deployment time. Those values are generated, for example, using a stream cipher with a key only known to the verifier. The advantage of this approach is clear: random values do not hinder attestation, since the verifier knows them, and the attacker cannot simply overwrite those values because they are used in the computation of the checksum.

The following attack is effective against any attestation scheme that uses random data to fill empty memory space before deployment.

The idea is to compress the original code in program memory in order to free enough space to store malicious data (Figure 5). At attestation time, the malicious code can decompress the original program on-the-fly, retrieve the original program words and succeed in the attestation. As our tests show on demo TinyOS applications, code size can be significantly compressed, reducing it by 11.6%, on average (Table 1). That translates to around 2.3 KBytes of free space for the considered applications.

For the implementation of the compression attack, we used Canonical Huffman encoding [14] because of its simplicity and its ability to start decompression from arbitrary positions of the compressed stream. Which is important if the attestation routine requires pseudo-random memory access.

Our decompression routine uses a list of checkpoints in the compressed stream as a trade-off between space (to keep the list in memory) and average speed to decompress an arbitrary memory word. The decompression routine of the Canonical Huffman encoding was implemented on the Atmel AVR platform. It uses only 1707 bytes of program memory and 2565 bytes of data memory. Using Canonical Huffman encoding, we were able to compress the code of Multi-hop Oscilloscope for Micaz (31836 bytes) to 27368 bytes. Using 512 bytes for the Canonical Huffman tree and 995 bytes for the checkpoints, we were left with 2961 bytes of free program memory to install arbitrary code. Although this seems a small gain for the attacker, it is sufficient to implement the attack we presented in Section 3.1.

Table 2 compares the time to access Multi-hop Oscilloscope code with and without compression for sequential and pseudo-random access, respectively. For the latter, if compression is

[6]For example, MicaZ motes have 128 KBytes of program memory while a typical application size is between 10 to 60 KBytes.

original instructions	added instructions	comment
...		previous instr
	`sbrs r31,7`	skip next instruction if bit 7 is set in $r31$, i.e. if address $> 0x8000$
	`cbr r31, 6`	clear bit 6 of address
`lpm Z`		read program memory at address (r31,r30)
...		

(a) Additional instructions of the memory shadowing attack; r31 holds high byte of random address, (Z is a 16 bit register and an alias to the 8 bit registers r30 and r31).

Addr. MSB	expected address range	changed MSB	resulting address range
0 0	0x0000-0x3FFF	0 0	0x0000-0x3FFF
0 1	0x4000-0x7FFF	0 1	0x4000-0x7FFF
1 0	0x8000-0xBFFF	1 0	0x8000-0xBFFF
1 1	0xC000-0xFFFF	1 *0*	*0x8000-0xBFFF*

(b) Address translation performed with the memory shadowing attack in Figure 6(a); as the address range (0xC000,0xFFFF) is not included in the checksum, the attacker could store the modified attestation code there.

Figure 6: Outline of the memory shadowing attack, with inserted instructions and translated addresses.

used, total time could be reduced incrementing the number of checkpoints. While incurred delay could be detected by a verifier, previously proposed protocols that fills program memory with randomness [32] do not rely on strict time bounding.

4. ON THE DIFFICULTY OF DESIGNING SECURE TIME-BASED ATTESTATION PROTOCOLS

This section presents attacks on some specific code attestation schemes. Our goal is to show that secure time-based attestation schemes are hard to design. We first focus on SWATT [25] and describe an attack that questions its main design assumption; we then show that SWATT can not be easily ported to devices others than the ones used in the original implementation. Finally, we investigate how to extend SWATT to prevent those attacks.

The second part of this section considers the ICE protocol [22] and presents an attack that violates one of its security features.

4.1 SWATT

The security of SWATT [25] relies on the time it takes for the attested device to compute its memory checksum. Memory words, to be input to the checksum function, are chosen in a pseudo-random fashion using a seed provided by the verifier. To succeed in the checksum computation, a malicious device would need to test each memory access and, if it points to a location where the malware is stored, redirect it to the location where the original memory word was copied, or alternatively "guess" its value. The authors claim that redirection incurs in a delay that can be easily detected by the verifier. If response time is below a given threshold, the verifier can be confident that the prover is not running malicious code. Precise estimation of the threshold is therefore crucial to the security of SWATT, in order to differentiate between modifications to the original checksum function and transmission delays.

4.1.1 A memory shadowing attack

The authors of [25], assumed that the best attack against SWATT is to check each address generated in the pseudo-random sequence. This test would be implemented by performing a test and redirection using *branch and compare* instructions. They expect at least 3 machine code cycles for test and redirection.

Since their implementation of the attestation function main loop is 23 cycles long, adding 3 cycles for test and redirection would lead to a malicious attestation function that is 13% slower than the original one. It is clear that this argument holds assuming that (1) they have the fastest implementation of the checksum function [7] and (2) they considered the fastest implementation of the attack. Such assumptions might lead to unexpected security flaws. Indeed, test and redirection can be implemented with less than 3 machine cycles.

Our attack leverages the fact that most of the program memory available on a sensor is usually empty (i.e., set to `0xFF`). If the second half of the program memory (addresses `1xxx...x`) is empty [8], all accesses to the last quarter of the memory (addresses `11xx...x`) could be redirected to the third quarter (addresses `10xx...x`) just flipping one bit in the address. The last quarter of the program memory would then be shadowed and could be used to hide malicious code.

At each memory access by the attestation routine, the adversary checks the highest bit of the address to load and, if it is set, sets the second highest bit to zero (Figure 6(b)). This can be implemented in an AVR micro-controller with two instructions (Figure 6(a)). We used the instruction `sbrs R, k` (skip next instruction if bit k is set in register R) followed by a `cbr R, k` instruction (clear bit k in register R).

Those two instructions take exactly 2 cycles to execute (irrespective of the result of the bit test). In our tests, we also needed to reorder some instructions to avoid corruption of the address pointer which is later included in the checksum computation. Instruction reordering does not change the global timing of the checksum algorithm.

The resulting modified checksum computation routine is presented in Appendix A. We tested the attack on our complete TinyOS implementation of SWATT and found that the new routine is only 7.4% slower than the original one (Figure 7). This result, on a practical implementation, leads to an attack that is 43% faster than the best attack expected by the designers of SWATT (13% of overhead). While this

[7] Speed requirements force implementation in assembly language, which is a very error-prone process. For example, we found one bug in the original implementation of SWATT provided in [25]: the assembly code is not performing the RC4 table swap properly. Although this is just a simple coding error, it has a dramatic effect on the quality of the generated random numbers. In fact, this error decreases the entropy of the internal state of the stream cipher. At each RC4 round, one position of the 256 bytes RC4 internal state is overwritten with the value of a register that is not initialized.

[8] This attack would therefore not be possible if the free program memory is used or filled with randomness (as in [5, 32]), but this is not the case with SWATT.

overhead could still be detected by the verifier, it shows that it is extremely difficult for protocol designers to assess what is the best attack against their protocols. Indeed, the best possible attack highly depends on the functionalities present in the instruction set of the micro-controller and on the set of available peripherals[9]. We therefore cannot exclude the existence of other implementations of a malicious checksum computation function that would compute a valid checksum without any noticeable delay.

4.1.2 Porting SWATT on MicaZ

SWATT was implemented for an early mica Berkeley mote, based on an ATMega163L micro-controller which has 16 KBytes of program memory. The devices considered in this paper are the most recent Mica2 and MicaZ sensors that use an ATMega128L micro-controller [3] which has 128 KBytes of program memory. Although the two micro-controllers are very similar, using the original SWATT code on the ATMega128L micro-controller would fail to check half of the program memory. Running SWATT once for each half of program memory would be fatal for the security of the protocol as the attacker could copy its malicious code from one half of the program memory to the other in a constant time between the two checks.

Surprisingly, porting SWATT to the new device was not straightforward and required a heavy redesign of the protocol. On the Atmega163L micro-controller the whole program memory can be addressed with a 16 bit pointer (the Z pointer) and a specific instruction "LPM" (Load from Program Memory). In SWATT this address is computed with one byte generated from $RC4$ pseudo-random stream and an extra byte specific to the SWATT algorithm. The 16 bit address is sufficient to address 64 KBytes of program memory.

In order to check the whole program memory of an ATMega128L micro-controller, we need to use another instruction, "ELPM" (Extended Load from Program Memory), that can access the whole memory byte-wise. This instruction uses the Z pointer plus another bit in a configuration register (RAMPZ) in order to build the 17 bit address needed to access the whole program memory. We implemented this solution by using, at each step of the partially unrolled loop, an extra random bit. As the unrolled loop contains 8 memory accesses, the extra random bit is provided by a spare register loaded with one $RC4$ random byte. For each of the 8 memory accesses, our modified implementation uses one bit of the spare register to compute the 17-th bit of the address.

Changes to the original SWATT protocol have a non-negligible side effect. The main loop of the SWATT attestation routine is extended by 4.8 cycles on average, while the original attack [25] as well as the memory shadowing one (Section 4.1.1) are possible in the same time. Therefore, the overhead of the original attack is reduced from 13% to 10.7% and the memory shadowing attack overhead is reduced from 7.4% to 6.5% (Figure 7).

We conclude that the security of SWATT relies on some unique characteristic of the devices considered by the authors to run their experiments. Porting SWATT on a new device with a new instruction set or a different memory size, dramatically changes the rules for both the attacker and the verifier, which can undermine the security of the scheme.

[9]For example, AVR micro-controllers have powerful bit manipulation instructions and a DMA engine is present on the MSP430 micro-controller used in Telosb motes.

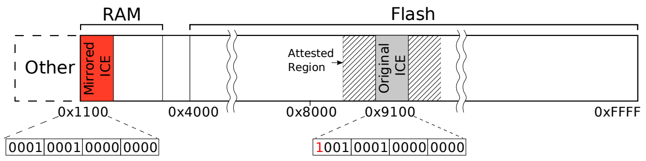

Figure 8: While the legitimate ICE routine is stored at address 0x9100, a malicious copy of the routine is stored at address 0x1100. These two addresses differ only in their most significant bit allowing the attacker to run the malicious copy of ICE and still pass attestation.

4.1.3 Preventing the rootkit attack

In [25] the authors do not consider attestation of data memory as the AVR architecture does not allow to execute code stored there. As seen in Section 3.1, an attacker could use ROP to transfer malicious code between executable memory and non-executable ones. To prevent such attacks there are two possible approaches: attesting data memory, or having SWATT clean data memory at the end of the attestation protocol.

Data memory attestation.

Modifying SWATT to check data memory as well is non-trivial and requires a deep redesign of the SWATT main loop. One of the challenges is that program and data memory are not accessed with the same instructions and are located in different address spaces. A possible solution would be to check the program memory and the data memory in two consecutive steps. This would be risky as the attacker could move malicious data/instructions right between the two steps and avoid detection. Alternatively, SWATT could be designed such that, at each iteration of the checksum function one of the two memories is chosen at random and then a random word is accessed within the selected memory. However, accessing one out of two memories per iteration would let the attacker insert its malicious instructions in a branch executed every two memory loads, on average. As a result, the overhead of an attack such as the memory shadowing one (Section 4.1.1), would be divided by two, i.e., the malicious instructions would be executed half of the time. Therefore, both memories must be attested at the same time to guarantee the trustworthiness of the device.

Lastly, it is important to consider that the data address space contains different regions (registers, I/O space and Data sections) that might not be included in the checksum computation because their values are unpredictable to the verifier.

Enforcing memory cleanup.

SWATT can enforce memory cleanup at the end of the attestation protocol, by erasing the whole data memory and rebooting the device without performing any function return.

The verifier has a copy of the original code on the device, so it can check if checksum computation has been performed without returning. Not executing a return instruction would prevent the attack presented in Section 3.1, but not the shadowing attack showed in Section 4.1.1.

Method	Time of Execution (ms)	Attack Overhead (ms)	Attack Overhead (%)
Original SWATT	11061	-	-
Original SWATT Attack	-	-	13 %
Our Shadow Attack	11879	818	7,4 %
SWATT 128 KBytes	13103	-	-
Shadow Attack 128 KBytes	13956	852	6,51 %
Attack ROP (Hiding time)	-	42,3	0,32 %

Figure 7: Timing of different attacks. The timings collected on SWATT with 128 KBytes were performed with the same number of cycles that the original SWATT. On 128 KBytes the number of SWATT cycles should be increased, according to the *Coupon's Collector Problem*; we have not done it in order to have easily comparable values.

4.2 ICE-based attestation schemes

Indisputable Code Execution (ICE) based protocols (such as, SCUBA [22], SAKE [21] and Message-in-a-bottle [18]) are a class of protocols that use the ICE routine to perform attestation. The ICE routine is a self-checksumming routine used to bootstrap trust on a remote device. The self checksumming code is based on a class of functions, called T-functions [16], used to generate a random permutation of memory locations. For each memory location traversed, a 160 bit checksum value C composed of ten 16 bit registers C_j ($C = [C_0, ..., C_9]$) is updated as follows:

$$\begin{aligned} C_j &= C_{j-1} + PC \oplus mem[current\_address] \\ &\quad + j \oplus C_{j-1} + x \oplus current\_address + C_{j-2} \oplus SR \end{aligned}$$

where PC is the program counter, x is the last value returned by the T-function, j is a loop counter, SR is the status register, $+$ denotes the addition of two 16 bit words without carry and $\oplus$ is the 16 bit exclusive or operation. The program counter and the status register are included to prevent a wide range of attacks detailed in the original paper. To optimize the computations, these values are mixed together only using bit-wise exclusive or operation and addition, two functions that provide poor diffusion of the input bits.

As explained earlier, some micro-controllers do not make the current program counter directly accessible to software. Unlike other protocols reviewed in this paper, ICE has been originally proposed for TelosB devices based on an MSP430 [30] micro-controller with a Von Neumann memory architecture. On the MSP430 the program counter is directly accessible as a special register.

Our attack aims at altering two input values, such that these two alterations would cancel out and therefore lead to a correct checksum. This could be accomplished flipping the most significant bit (MSB) of, for example, the PC and of the status register. Altering the MSB is the best choice because, since additions discard the carry, a change of this bit does not propagate to other bits. Another possibility to obtain the same result is to flip the MSB of the PC register (i.e. running a copy of ICE at a different address) and the MSB of every memory value accessed by ICE (i.e. $mem[current\_address]$).

Alteration to the PC leads to the attack depicted in Figure 8. It allows to store a copy of the ICE routine at a different position than it was intended to, violating one of the main security property that ICE is expected to guarantee. This specific property is crucial for several protocols that rely on ICE, as they assume that after its execution, ICE will hand execution to an attested part of the code. Because the displaced copy of the ICE routine is not modified, it runs in exactly the same time as the original one and computes the correct checksum. Therefore, it passes the attestation and it is able to hand over execution to any code of its choice.

5. CONCLUSION

This paper investigated the security of existing software-based device attestation protocols.

Software based attestation on general purpose operating systems [15] has been previously shown to have serious weaknesses [28]. To our knowledge this paper presents the first security analysis of software based attestation schemes specifically designed for low en embedded systems.

We presented two generic attacks on software code attestation. We also designed and implemented new specific attacks (and discussed possible fixes) against existing software attestation techniques, namely SWATT and ICE.

From our experience, we can conclude that secure time-based attestation schemes are very difficult, if not impossible, to design correctly. Time-based attestation schemes must rely on very tight timing bounds. Their implementation must therefore be small, simple and time-optimized. Those properties rule out cryptographic functions as they are complex and time consuming. Design choices are then restricted to ad-hoc functions (usually based on permutations or bit-wise exclusive or operations) which very often provide only weak security. In fact, one of our attacks partially leverages on a weakness of the functions used for checksum computation. Moreover, speed requirements force implementation in assembly language, which is a very error-prone process. We also stress that attesting only the code memory, as performed by existing schemes, is not sufficient. As shown by our rootkit attack, an attacker can still hide malicious code using Return-Oriented Programming. We argue that all memories (RAM, ROM, EEPROM) have to be attested. Designing an attestation scheme that involves all the memories of the end device is quite challenging. Our future work will be to investigate the feasibility of a software-based attestation protocol that can guarantee security while being efficient and portable across different architectures.

ACKNOWLEDGMENTS

The authors would like to thank Karim El Defrawy for his helpful feedback and editorial suggestions as well as our shepherd Rob Johnson and the anonymous reviewers for their insightful comments and help to improve this article.

The work presented in this paper was supported in part by the European Commission within the STREP WSAN4CIP project. The 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.

6. REFERENCES

[1] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity. In *CCS '05: Proceedings of the 12th ACM conference on Computer and Communications Security* (2005), ACM.

[2] Anderson, R., and Kuhn, M. Tamper resistance - a cautionary note. In *In Proceedings of the Second Usenix Workshop on Electronic Commerce* (1996).

[3] Atmel Corporation. Atmega128 datasheet. `http://www.atmel.com/atmel/acrobat/doc2467.pdf`.

[4] Buchanan, E., Roemer, R., Shacham, H., and Savage, S. When good instructions go bad: generalizing return-oriented programming to RISC. In *Proceedings of CCS '08* (2008), ACM.

[5] Choi, Y.-G., Kang, J., and Nyang, D. Proactive code verification protocol in wireless sensor network. In *ICCSA* (2007), O. Gervasi and M. L. Gavrilova, Eds., vol. 4706 of *Lecture Notes in Computer Science*, Springer.

[6] Cooprider, N., Archer, W., Eide, E., Gay, D., and Regehr, J. Efficient memory safety for TinyOS. In *SenSys '07* (2007), ACM.

[7] England, P., Lampson, B., Manferdelli, J., Peinado, M., and Willman, B. A trusted open platform. *Computer 36*, 7 (2003).

[8] Ferguson, C., Gu, Q., and Shi, H. Self-healing control flow protection in sensor applications. In *WiSec '09* (2009), ACM.

[9] Francillon, A., and Castelluccia, C. Code injection attacks on Harvard-architecture devices. In *ACM Conference on Computer and Communications Security* (2008), P. Ning, P. F. Syverson, and S. Jha, Eds., ACM.

[10] Goodspeed, T. Exploiting wireless sensor networks over 802.15.4. In *Texas Instruments Developper Conference* (2008).

[11] Gu, Q., and Noorani, R. Towards self-propagate mal-packets in sensor networks. In *WiSec* (2008), ACM.

[12] Hoglund, G., and Butler, J. *Rootkits : Subverting the Windows Kernel*. Addison-Wesley, 2005.

[13] 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.

[14] Huffman, D.A. A method for the constructionof minimum redundancy codes. *Proceedings of the IRE 40* (1962).

[15] 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.

[16] Klimov, A., and Shamir, A. New cryptographic primitives based on multiword t-functions. In *Fast Software Encryption, 11th International Workshop, FSE 2004* (2004).

[17] Krahmer, S. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. Tech. rep., suse, September 2005. available at http://www.suse.de/ krahmer/no-nx.pdf.

[18] Kuo, C., Luk, M., Negi, R., and Perrig, A. Message-in-a-bottle: user-friendly and secure key deployment for sensor nodes. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems* (2007), ACM.

[19] Nergal. The advanced return-into-lib(c) exploits (pax case study). *Phrack Magazine 58*, 4 (2001). http://www.phrack.org/issues.html?issue=58&id=4.

[20] Park, T., and Shin, K. G. Soft tamper-proofing via program integrity verification in wireless sensor networks. *IEEE Trans. Mob. Comput. 4*, 3 (2005).

[21] 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).

[22] 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.

[23] 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.

[24] Seshadri, A., Perrig, A., van Doorn, L., and Khosla, P. Using SWATT for verifying embedded systems in cars. In *Proceedings of Embedded Security in Cars Workshop (ESCAR 2004)* (Nov. 2004).

[25] 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.

[26] Shacham, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of CCS 2007* (2007), ACM.

[27] Shaneck, M., Mahadevan, K., Kher, V., and Kim, Y. Remote software-based attestation for wireless sensors. In *ESAS* (2005).

[28] 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).

[29] Solar Designer. *return-to-libc attack*. Bugtraq mailing list, August 1997.

[30] Texas Instruments. Msp430 f1611 datasheet.

[31] Yang, X., Cooprider, N., and Regehr, J. Eliminating the call stack to save ram. In *To appear in LCTES 2009* (June 2009), ACM.

[32] 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.

APPENDIX

A. MODIFIED SWATT IMPLEMENTATION AND ATTACK

Generate i^{th} member of random sequence using RC4				cycles
initialize high byte of array address		zh ← 2	ldi r31, 0x02	1
$i++$	and $R15 <= S[i]$	r15 ← *(x++)	ld r15, x+	2
$j = j + S[i]$		yl ← yl + r15	add r28, r15	1
	$(R30 <= S[j])$	zl ← *y	ld r30, y	2
$swap(S[i], S[j])$		*y ← r15	st y, r15	2
		*x ← zl	st x, r30	2
$tmp = S[i] + S[j]$	index to read from	zl ← zl + r15	add r30, r15	1
$RC4_i = S[tmp]$	RC4 value, saved to zh	zh ← *z	ld r31, z	2
Generate 16-bit memory address				
$Z = Zh \vert Zl = RC4_i \vert C_{k-1}$	$A_i <=> Z$	zl ← r6	mov r30, r6	1
Load byte from memory and compute transformation				
$R0 = Mem[Ai]$		r0 ← *z	lpm r0, z	3
$R0 = R0 \oplus C_{k-2}$	$C_{k-2} <=> R13$	r0 ← r0 ⊕ r13	xor r0, r13	1
$R0 = R0 + RC4_{i-1}$	$RC4_{i-1} <=> R4$	r0 ← r0 + r4	add r0, r4	1
Incorporate output of transformation into checksum				
$C_k = C_k + R0$		r7 ← r7 + r0	add r7, r0	1
$C_k = rot(C_k)$		r7 ← r7 ≪ 1	lsl r7	1
		r7 ← r7 + carry bit	adc r7, r5	1
		r4 ← zh	mov r4, r31	1
total cycles				23

Figure 9: Original SWATT implementation on AVR micro-controller. In the original paper, at the 6^{th} line the instruction is *st x, r16*. $r16$ is never affected and $r30$ holds the value to swap.

Generate i^{th} member of random sequence using RC4				cycles
initialize high byte of array address		zh ← 2	ldi r31, 0x02	1
$i++$	and $R15 <= S[i]$	r15 ← *(x++)	ld r15, x+	2
$j = j + S[i]$		yl ← yl + r15	add yl, r15	1
	$(R30 <= S[j])$	zl ← *y	ld r30, y	2
$swap(S[i], S[j])$		*y ← r15	st y, r15	2
		*x ← zl	st x,r30	2
$tmp = S[i] + S[j]$	index to read from	zl ← zl + r15	add r30, r15	1
$RC4_i = S[tmp]$	RC4 value, saved to zh	zh ← *z	ld r31, z	2
Generate 16-bit memory address				
$Z = Zh \vert Zl = RC4_i \vert C_{k-1}$	$A_i <=> Z$	zl ← r6	mov r30, r6	1
add r4 now (previous memory address)				
$C_k = C_k + RC4_{i-1}$		r7 ← r7 + r4	add r7, r4	1
backup the r31 to r4 before modifying it				
		r4 ← zh	mov r4, r31	1
mangle two high bits of memory address				
	skip next instr. if address starts with 0		sbci r31,7	$\}2$
	clear bit 6 of Zh		cbr r31, 64	
Load byte from memory and compute transformation				
$R0 = Mem[Ai]$		r0 ← *z	lpm r0, z	3
$R0 = R0 \oplus C_{k-2}$	$C_{k-2} <=> R13$	r0 ← r0 ⊕ r13	xor r0, r13	1
Incorporate output of transformation into checksum				
$C_k = C_k + R0$		r7 ← r7 + r0	add r7, r0	1
$C_k = rot(C_k)$		r7 ← r7 ≪ 1	lsl r7	1
		r7 ← r7 + carry bit	adc r7, r5	1
total cycles				25

Figure 10: Malicious implementation of SWATT on a AVR micro-controller; main loop is 2 cycles longer. This is possible because commutative operators are used in the checksum computation (operator `and` and `exclusive or`).