SWATT: SoftWare-based ATTestation for Embedded Devices*

Arvind Seshadri
CMU/CyLab

Adrian Perrig
CMU/CyLab

Leendert van Doorn
IBM

Pradeep Khosla
CMU/CyLab

Abstract

We expect a future where we are surrounded by embedded devices, ranging from Java-enabled cell phones to sensor networks and smart appliances. An adversary can compromise our privacy and safety by maliciously modifying the memory contents of these embedded devices. In this paper, we propose a SoftWare-based ATTestation technique (SWATT) to verify the memory contents of embedded devices and establish the absence of malicious changes to the memory contents. SWATT does not need physical access to the device's memory, yet provides memory content attestation similar to TCG or NGSCB without requiring secure hardware. SWATT can detect any change in memory contents with high probability, thus detecting viruses, unexpected configuration settings, and Trojan Horses. To circumvent SWATT, we expect that an attacker needs to change the hardware to hide memory content changes.

We present an implementation of SWATT in off-the-shelf sensor network devices, which enables us to verify the contents of the program memory even while the sensor node is running.

1 Introduction

With the proliferation of embedded devices, we expect to be surrounded by them in the near future. For example, furniture, clothing, and appliances are all expected to contain integrated microcontrollers for advanced functions such as automated checkout, inventory control, fire detection, or climate monitoring. It is clear that such devices pose an inherent risk, as an attacker may only need to compromise one of our devices to compromise our privacy and safety. This risk is compounded by the frequent lack of security in embedded devices. In an environment where we are surrounded by embedded devices, we thus need mechanisms to attest the current memory contents of the devices, to detect when an attacker altered the software or configuration.

In embedded systems, cost is a concern. Even a small increase in per device cost leads to a significant increase in overall production costs in high-volume manufacturing. Clearly, a software-based attestation technique will incur lower cost than an attestation technique that requires additional hardware. More importantly, a technique that works entirely in software can be used on legacy devices. In this paper, we propose such a purely SoftWare-based ATTestation technique called SWATT.

SWATT *externally* attests the code, static data, and configuration settings of an embedded device. By "externally" we mean that the entity performing the attestation (verifier) is physically distinct from the embedded device. Hence, the verifier cannot directly read or write the device's memory. Note that we need an external verifier since, without secure hardware, a (potentially) compromised device cannot be trusted to verify itself correctly.

SWATT uses a challenge-response protocol between the verifier and the embedded device. The verifier sends a challenge to the embedded device. The embedded device computes a response to this challenge, using a verification procedure that is either pre-programmed into the embedded device's memory or downloaded from the verifier prior to verification. The verifier can locally compute the answer to its challenge, and can thus verify the answer returned by the embedded device. The design of SWATT ensures that the embedded device can return the correct answer only if its memory contents are correct. In fact, when SWATT is used the only way an attacker can hide malicious changes to the memory contents of the embedded device is to change the device's hardware. To motivate the use of SWATT, we discuss several applications:

- As a concrete example of the applicability of SWATT, consider network printers. Some network printers are extremely vulnerable today [6], where anybody with network access can easily upload software to a printer, turning it into an eavesdropping or active attack device. A network administrator can use SWATT over the lo-

*This research was supported in part by the Center for Computer and Communications Security at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office, and by gifts from Bosch, Cisco, Intel, and Matsushita Electric Works Ltd. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of ARO, Bosch, Carnegie Mellon University, Cisco, Intel, Matsushita Electric Works Ltd., or the U.S. Government or any of its agencies.

cal network to attest that the printer code and configuration settings are as expected. During the attestation process, the administrator can configure the network such that the printer can only communicate with the verifier. This ensures that answer returned to the attestation request is authentic, by preventing a malicious printer from colluding with other entities to answer the challenge from the verifier.

- Consider a smart cell phone, with an e-mail client as part of its firmware. Suppose that a worm that exploits a vulnerability in the e-mail client is currently active on the Internet. The user of the cell phone will want to know if their e-mail client has been compromised. To determine this, the user plugs the cell phone into its hot-sync cradle that is linked to the PC. The PC then uses SWATT to verify the code running on the cell phone. The point-to-point nature of the communication link between the PC and cell phone ensures the authenticity of the answer returned by the cell phone. If the cell phone was compromised, we can use SWATT to verify that a patch installation or software reinstallation was successful.

- Electronic voting machines represent an important application of our technique. Recently, there have been instances where manufacturers of e-voting machines used uncertified voting software [1]. To prevent such occurrences, voting machine inspectors can use SWATT to ensure that the correct voting software is running on the voting machine.

- Consider smart cards that are used to store user passwords. Before entrusting their passwords to such a smart card, a user can use SWATT to ensure that the smart card has the correct code running on it. Otherwise the smartcard may contain a malicious code segment that could leak all passwords.

- In the future, airlines, hotels or car rental companies may provide Personal Digital Assistants (PDA) for personal use. An attacker could easily reprogram one of these devices such that it captures any username and password information that is input. The vendor could erase and reprogram each device after it is returned by a customer, to ensure that the code running on it is trustworthy. But how can a customer verify that the firmware is trustworthy before using the device? To verify the firmware, the user could plug a USB key into the PDA and use SWATT to verify the code running on the PDA.

As we will further discuss in the related work section, SWATT is quite different from any other techniques that we are aware of. SWATT may appear to provide similar properties to secure boot, but it is distinct. Systems such as TCG (formerly known as TCPA) [16] and NGSCB (formerly known as Palladium) [11] use a secure coprocessor that is used during system initialization to bootstrap trust. SWATT does not need a secure coprocessor, and allows a trusted external verifier to verify the memory contents of an embedded device. Once the code running on the embedded device is verified, the code forms the trusted computing base. Hence we bootstrap trust entirely in software.

Kennell and Jamieson propose techniques to verify the genuinity of computer systems entirely in software [9]. Central to their technique is the premise that by including sufficient amount of architectural meta-information that is generated in a complex CPU into a simple checksum of the memory contents, an attacker with a different CPU, who is trying to simulate the CPU in question, will suffer a severe slowdown in checksum computation. They use the virtual memory subsystem (caches hits and misses, TLB hits and misses etc.) as the source of architectural meta-information. However, the 8 and 16-bit microcontrollers, that are used in small embedded systems, have no virtual memory support. Devices based on small microcontrollers constitute the majority of embedded systems that are manufactured and used today. Kennell and Jamieson's techniques cannot be used to verify these devices. Further, as we describe in detail in the related work section, Kennell and Jamieson's method suffers from a security vulnerability that allows an attacker to hide malicious code on the platform.

Scope of this paper In this paper, we propose an approach to verify the memory contents of an embedded device without having physical access to the device's memory. SWATT is secure as long as the verifier has a correct view of the hardware. In particular, the verifier needs to know the clock speed, the instruction set architecture (ISA), and the memory architecture of the microcontroller on the embedded device, and the size of the device's memories. In this paper, we do not address the case where the attacker changes the hardware (e.g., uses a faster microcontroller).

As a first step, we have implemented SWATT on the Berkeley Mica Mote platform. The motes have an 8-bit microcontroller with no virtual memory support. Hence, they are a perfect example of a small embedded system.

Contributions This paper makes the following contributions:

- We propose, SWATT, a technique to *externally* verify the contents of the memory of an embedded device. We show that an external verifier can detect with high probability if a single byte of the memory deviates from the expected value. SWATT provides a strong guarantee of correctness. We show that if SWATT successfully terminates, we have a high probability that the memory contents are correct.

- We verify the memory contents entirely in software and do not need secure coprocessors or other secure hardware. Thus, SWATT can be used on legacy systems. Also, if cost is a concern, secure hardware may not be available.
- We present an implementation of SWATT on the Berkeley Mica Motes, a sensor network node architecture. Our implementation enables us to directly verify the program memory content of a running sensor node.

Outline In Section 2, we give a problem definition and describe the attacker model. Section 3 presents the SWATT design, implementation, and evaluation. In Section 4 we discuss related work, and we present our conclusions in Section 5.

2 Problem Definition, Assumptions and Threat Model

A naive approach for verifying the embedded device's memory contents is for the verifier to challenge the embedded device to compute and return a message authentication code (MAC) of the embedded device's memory contents. The verifier sends a random MAC key, and the embedded device computes the MAC on the entire memory using the key and returns the resulting MAC value. The random key prevents pre-computation and replay attacks, that would be possible if a simple hash function were used. However, we show that just verifying the response is insufficient—an attacker can easily cheat. The embedded device is likely to have some empty memory, which is filled with zeros. When an attacker alters parts of the memory (e.g., inserting a Trojan horse or virus), the attacker could store the original memory contents in the empty memory region and compute the MAC function on the original memory contents during the verification process. Figure 1 illustrates this attack. It is not necessary for the embedded device to have an empty memory region for this attack to succeed. An attacker could just as easily move the original code to another device that it could access when computing the MAC.

Consider the setting of Figure 2. A *verifier* wants to check whether the memory contents of an *embedded device*, which we refer to hereafter as the *device*, is the same as some expected content. We assume that the verifier knows the expected memory contents; our goal is to design an effective *verification procedure*, which will resist the attack described previously. The verification procedure will be used by the device to compute a checksum over its memory contents. The checksum will be correct only if the memory contents of the device is the same as the value expected by the verifier, and it will fail with high probability if the memory contents of the device differs from the expected content. We say a verification procedure with this property is a *secure verification procedure*.

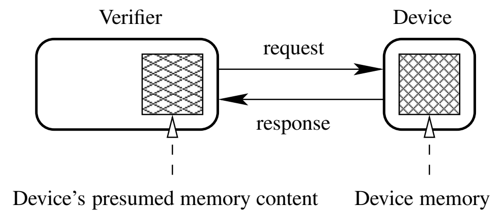

Figure 2. Generic external memory verification. The verifier has a copy of the device's presumed memory, and sends a request to the embedded device. The device can prove its memory contents by returning the correct response.

Assumptions We assume that the device contains a memory-content-verification procedure that the verifier can activate remotely.[1] We also assume that the verifier knows the exact hardware architecture and the expected memory contents of the device. In particular, for the hardware, the verifier knows the clock speed, the memory architecture and the instruction set architecture (ISA) of the microcontroller on the embedded device, and the size of the device's memory.

To verify that the device's memory matches the expected memory contents, the verifier creates a random challenge and sends it to the device. The device then computes the response to the challenge using the verification procedure. Using its local copy of the device's expected memory, the verifier can locally compute the expected response and verify the correctness of the device's response. Note that we do not need to assume that the device contains a trusted version of the verification procedure—for example, we assume that an attacker can take full control of a compromised device and may not run the legitimate verification procedure. However, a secure design of the verification procedure will ensure that the verification will fail if the memory content of the device does not match the expected content no matter what code the device runs for the verification.

Threat Model We assume that an attacker has full control over the memory of the device. However, we assume that the attacker does not modify the hardware of the device, e.g., increase the size of the memory, change the memory

[1] The procedure could also be downloaded any time prior to the verification.

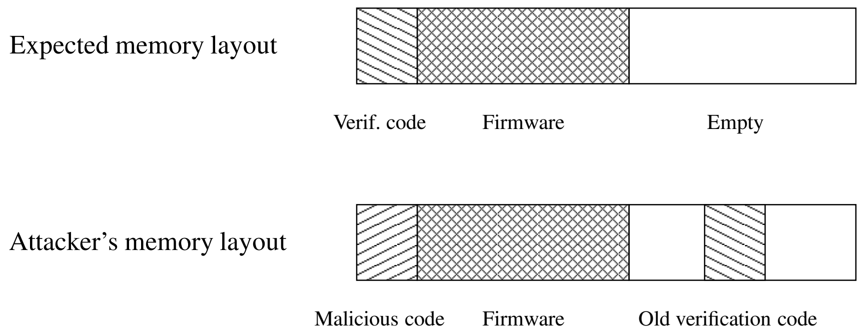

Figure 1. Memory verification attack. The attacker replaces the verification code with malicious verification code and copies the old verification code into empty memory.

access timings or increase the clock speed of the processor either by modifying the hardware or by changing settings in the device's BIOS.[2]

3 SWATT: SoftWare-based ATTestation

We start this section by discussing our general approach. We then discuss the desired properties of the memory-content-verification procedure. Section 3.3 describes our implementation. This is followed by experimental evaluation of SWATT. We then give some guidelines for practical applications of SWATT.

3.1 Approach: Pseudorandom Memory Traversal

As mentioned in Section 2, the embedded device contains a memory-content-verification procedure that the verifier can activate remotely. This verification procedure uses a *pseudorandom memory traversal*. In this approach, the verifier sends the device a randomly-generated challenge. The challenge is used as a seed to the pseudorandom number generator (PRG) which generates the addresses for memory access. The verification procedure then performs a pseudorandom memory traversal, and iteratively updates a checksum of the memory contents. The key insight, which prevents the attack on MACs mentioned in Section 2, is that an attacker cannot predict which memory location is accessed. Thus, if the attacker alters the memory, it has to perform a check whether the current memory access is to one of the altered locations, for each iteration of the verification procedure. If the current memory access indeed touches an altered location in the memory, the attacker's verification procedure needs to divert the `load` operation to the memory location where the correct copy is stored. Even if the attacker alters a single memory location, the increase in running time of the verification procedure due to the added `if` statement becomes noticeable to the verifier, as the verification procedure is very efficient and performs many iterations. So a verifier will detect the altered memory because either the checksum returned by the embedded device is wrong, or the result is delayed a suspiciously long time. We construct the verification procedure in a way that a single additional `if` statement will *detectably slow down* the checksum computation by the embedded device.

[2]In current embedded systems, the BIOS is often minimal and thus difficult to alter. We will discuss this assumption further in Section 3.6.

3.2 Desired Properties of the Verification Procedure

The verification procedure needs the following properties: pseudo-random memory traversal, resistance to pre-computation and replay attacks, high probability of detecting even single-byte changes to memory contents, small code size, efficient implementation, and non-parallelizable. In the following paragraphs we discuss the reasons why we need these properties. Section 3.3 describes our design and implementation to achieve these properties.

Pseudo-random memory traversal The verification procedure makes a pseudo-random traversal of the memory regions it checks, thereby forcing the attacker to check every memory access made by the verification procedure for a match with location(s) that the attacker altered. We use a cryptographic pseudo-random generator (PRG) to produce the pseudo-random sequence of memory locations. The choice of the PRG to be used depends on the CPU architecture of the device. For 8-bit architectures, we could use the the keystream generated by RC4 stream cipher to generate the pseudo-random memory locations. Helix is a fast stream cipher with built-in MAC functionality, optimized for 32-bit architectures [4]. So the keystream of Helix can be used to generate the pseudo-random sequence of memory accesses on 32-bit architectures, and the MAC can be used for the checksum. Another option for address generation on 16 and 32-bit architectures is to use bits from the output of a multiword T-function [10].

Resistance to pre-computation and replay attacks To prevent pre-computation and replay attacks, the checksum returned by the device must depend on a random challenge that is sent by the verifier. We achieve this by having the verifier send the seed for the pseudo-random generator (PRG), that is used by the verification procedure to generate the memory addresses for the memory traversal.

High probability of detecting changes The probability that the verification procedure returns the correct checksum when the attacker modifies some memory contents should be very small. First, we would like the verification procedure to touch every memory location, with high probability, even though it accesses the memory in a pseudo-random sequence. We achieve this by using the result of the Coupon Collector's Problem. The verification procedure does $O(n \ln n)$ accesses to the memory, where n is the memory size. The result of the Coupon Collector's Problem states that if X is the number of memory accesses required to access each memory location at least once then, $Pr[X > cn \ln n] \leq n^{-c+1}$. Second, we would like the checksum function to be sensitive to value changes. The checksum should differ even when the input changes by a single byte and it should be difficult for the attacker to find a different input that gives the same checksum result as the original memory content on a randomly chosen challenge.

Small code size Our pseudo-random memory traversal forces an attacker to insert an `if` statement into the verification procedure, which causes a detectable increase in its running time. An `if` statement translates to a compare instruction followed by a conditional jump in assembly. Hence, it takes about 2-3 CPU cycles to execute an `if` statement on most 8 and 16-bit microcontroller architectures. If insertion of the extra `if` statement is to cause a detectable slowdown of the verification procedure, then the main body (excluding the initialization and epilogue code) of the verification procedure should take a few tens of CPU cycles to execute.

Optimized implementation The attacker may find an optimized implementation of the verification procedure, so that the extra `if` statement does not cause any overhead. Hence, it is important that our implementation of the body of the verification procedure loop not allow any further optimizations. We need to optimize only the verification procedure loop since the initialization and epilogue code consume a negligible fraction of the execution time.

Joshi, Nelson and Randall propose a superoptimizer, Denali, that uses an automatic theorem prover to generate a nearly mathematically optimal straight line machine code sequence, that evaluates a given set of expressions using the minimal possible instruction count on a given architecture [8]. Denali is perfectly suited to optimize the body of the verification procedure loop, which is a straight line code sequence. Further, tools like GNU superopt generate the smallest instruction sequence for a given function on a specified architectural platform, by using an exhaustive generate-and-test approach [15]. Due to the small code size of the loop body of the verification procedure we were able to hand-optimize it to be highly efficient. In Section 3.3 we argue why our code sequence cannot be optimized further.

Non-parallelizable It may be possible for multiple devices to collude to speed up the checksum computation. For example, the checksum computation, which is a sequence of operations, can be split into two halves, each half can be computed independently by a device and the results combined in the end. The second device needs to run the PRG for half the total number of iterations of the verification procedure to bring the PRG into the state required to start computing the second half of the checksum. But this consumes less time than running the verification procedure fully. Hence, the total computation time using two devices will be less than that for a single device. To prevent this, we have to make the verification procedure non-parallelizable. We achieve this by making the address for the memory access and the computation of the checksum depend on the current value of the checksum.

3.3 Design and Implementation of Verification Procedure on Sensor Motes

We have designed and implemented our verification procedure for sensor motes, which use an Atmel ATMEGA163L microcontroller, an 8-bit Harvard Architecture with 16K of program memory and 1K of data memory [7]. The CPU on the microcontroller has a RISC architecture. We first describe our design, give the pseudo-code for the main loop of the verification procedure and then show its realization in assembly language of the ATMEGA163L. Even though we show an 8-bit implementation, the implementation can be extended to 16 or 32-bit architectures.

Pseudorandom memory traversal We use the RC4 Pseudo-Random Generator (PRG) by Rivest to generate the pseudo-random sequence of addresses for memory access. RC4 takes a seed as input and outputs a pseudo-random keystream. Known attacks exist against the RC4 stream cipher [5]. To evade these weaknesses, we discard the first 256 bytes of the RC4 keystream. We use RC4 because of its extremely efficient and compact implementation on a 8-bit architecture: our RC4 implementation only requires 8 machine instructions (in the main loop) and outputs one 8-bit pseudo-random number every 13 cycles on the ATMega163L microcontroller. Since we need 16-bit addresses to access the program memory of the microcontroller, we concatenate the 8-bit RC4 output with a current value of the checksum to generate a 16-bit address.

The checksum function To achieve a low probability of collision for different memory contents, we need a sufficiently long output for the checksum. If our checksum function outputs n bits, 2^{-n} is a lower bound on the collision probability. In this implementation, we use a 64-bit checksum.

We propose to use a simple and efficient checksum function. Efficiency on an 8-bit architecture was our main design goal, so that an additional `if` statement introduces a substantial slowdown. Further research is required to explore the design space for these checksumming functions, to identify the ideal tradeoff between security and efficiency.

Figure 3 shows the pseudocode of the checksum function and Figure 4 shows our implementation in assembly for the ATMega163L processor. We now describe the function in more detail and discuss the design decisions.

To generate the 64-bit checksum we treat the 64-bit checksum as a vector of eight 8-bit values. In each iteration of our function, we update one 8-bit value of the checksum, incorporating one memory value and mixing in the RC4 values as well as previous values of the checksum.

We derive the 16-bit address of the memory location to be accessed as follows. The high byte of the address is the RC4 value generated in that round. The previous value of the checksum vector is the low byte.

One of our design goals was that a changed memory location perturbs all fields of the checksum. To achieve this, each 8-bit value of the checksum affects the following two iterations. First, the 8-bit value of the checksum is used as the low byte of the memory address of the following iteration, and the value is again incorporated in the computation of the subsequent 8-bit value by XORing it with the loaded memory value. Thus, if an altered memory location is accessed, the following memory access will load a different value, and the checksum in the following iteration will also be affected. This design of checksum function also ensures that computation of the current 8-bit value of the checksum depends on previously computed 8-bit values. This makes the loop of the verification procedure non-parallelizable.

We did consider the possibility of using a MAC function for the checksum function. Helix is fast stream cipher with built-in MAC functionality [4]. Hence, the main loop of our verification procedure can be completely replaced by Helix. However, because Helix is optimized for 32-bit architectures, it takes many instructions to compute on 8 and 16-bit microcontrollers. Therefore, an `if` statement would only cause a small increase in running time. The same situation is true of MACs as well. Since they often take many instructions to compute, they may not be short enough to allow us to identify the slowdown caused by `if` statements.

Pseudocode Figure 3 shows the pseudocode of the main loop of the verification procedure. The code is presented in a verbose and unoptimized form to improve readability. The variable m represents the number of iterations of the loop, which is equal to the the total number of memory accesses performed by the checksum procedure. m is sent by the verifier as part of the verification request. The number m depends on the size of the memory being verified due to the result of the Coupon Collector's Problem.

The variables of the verification procedure are initialized as follows. We use the notation $RC4_i$ to denote the ith output byte of the RC4 keystream. Since we discard $RC4_0$ to $RC4_{255}$ for security reasons, the eight 8-bit values in the checksum C_0 to C_7 are initialized with $RC4_{256}$ through $RC4_{263}$. The initial value of $RC4_{i-1}$ is set to $RC4_{264}$.

Assembler code Figure 4 shows the assembly code corresponding to the pseudocode shown in Figure 3, written in the assembly language of the Atmel ATMEGA163L microcontroller. The current assembly code is manually optimized. In our manual optimization, by carefully unrolling the verification procedure loop, we do away with the code that updates the checksum index (the variable j in the pseudocode).

The architecture of the microcontroller has the following characteristics:

- The microcontroller has a Harvard Architecture, with 16Kbytes of program memory and 1 Kbyte of data memory.
- The CPU inside the microcontroller uses a RISC ISA. This means that all instructions except loads and stores have only CPU register and immediate as operands. Only loads and stores use memory addresses.
- The CPU has 32 8-bit general purpose registers, r0 - r31. Registers r26 and r27 together can be treated as a 16-bit register x, used for indirect addressing of data memory. Similarly, r28 and r29 form register y and r30 and r31 form register z. The upper and lower 8-bits of the 16-bit registers are named using the suffix 'h' and 'l' after the name of the register. Thus xh and xl refer to the upper and lower bytes of x and similarly for y and z.
- Data and program memory can be addressed directly or indirectly. To indirectly address data memory, one

algorithm Verify(m)
//Input: m number of iterations of the verification procedure
//Output: Checksum of memory
Let C be the checksum vector and j be the current index into the checksum vector
for $i \leftarrow 1$ **to** m **do**
//Construct address for memory read
$A_i \leftarrow (RC4_i \ll 8) + C_{((j-1) \mod 8)}$
//Update checksum byte
$C_j \leftarrow C_j + (Mem[A_i] \oplus C_{((j-2) \mod 8)} + RC4_{i-1})$
$C_j \leftarrow rotate\ left\ one\ bit(C_j)$
//Update checksum index
$j \leftarrow (j+1) \mod 8$
return C

Figure 3. Verification Procedure (Pseudocode)

of x, y or z registers holds the pointer to the memory location. In case of program memory, only the z register can be used for indirect addressing. Indirect addressing has displacement, pre-decrement and post-increment modes.

The main loop of our verification procedure is just 16 assembly instructions and takes 23 machine cycles. Hence, the addition of a single `if` statement (compare + branch) that takes 3 cycles, to the main loop, adds a 13% overhead, in terms of machine cycles, to each iteration of the loop.

In future work, we plan to use an approach similar to Denali [8] to ensure that our checksum code cannot be optimized further. In our current implementation, we conjecture that only the two `mov` instructions may be optimized away, as all other instructions perform essential operations. However, we need a strict mathematical argument that the code is optimal for high security.

3.4 Experiment and Results

We implemented two versions of the verification procedure—a genuine version and an attacker's version that assumes that the attacker changes a single byte of code and copies it into the data memory. To check for this the attacker then inserts an extra `if` statement into the verification procedure. This is an optimistic scenario for the attacker. If the attacker modifies even a single memory location, it also has to modify the verification procedure. This means that the attacker will end up making multiple changes to the memory contents. So, the attacker will have to insert multiple `if` statements.

For development and testing of our verification procedure, we used the AVR studio version 4.0, which is an integrated development environment for Atmel microcontrollers, developed by Atmel Corp. The AVR studio has a simulator for the ATMega163L. We ran both versions of the verification procedure in the simulator, keeping interrupts and all peripherals disabled to minimize running time variations due to external events. The simulator profiled both versions of the checksum function and returned the running time. We show a plot of the running time versus number of code memory accesses in Figure 5. We vary the number of memory accesses (which is equal to the number of iterations of the main loop) by the verification procedure from 10000 to 320000 ($\sim 2 \cdot n \ln n$, for $n =$ 16Kbytes).

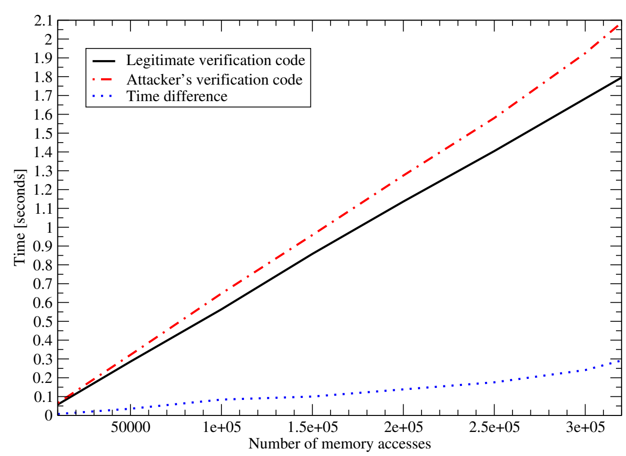

Figure 5. Result from our implementation. Time difference of legitimate code versus attacker code. The more memory locations we include in our checksumming procedure, the larger the time difference between the legitimate and attacker.

Assembly explanation	Pure assembly code
Generate i^{th} member of random sequence using RC4	
zh ← 2	ldi zh, 0x02
r15 ← *(x++)	ld r15, x+
yl ← yl + r15	add yl, r15
zl ← *y	ld zl, y
*y ← r15	st y, r15
*x ← r16	st x, r16
zl ← zl + r15	add zl, r15
zh ← *z	ld zh, z
Generate 16-bit memory address	
zl ← r6	mov zl, r6
Load byte from memory and compute transformation	
r0 ← *z	lpm r0, z
r0 ← r0 ⊕ r13	xor r0, r13
r0 ← r0 + r4	add r0, r4
Incorporate output of transformation into checksum	
r7 ← r7 + r0	add r7, r0
r7 ← r7 << 1	lsl r7
r7 ← r7 + carry_bit	adc r7, r5
r4 ← zh	mov r4, zh

Figure 4. Verification Procedure (Assembly Code)

3.5 Considerations for Practical Use

Selecting number of iterations Due to the pseudo-random nature of the memory access by the verification procedure, there is always a finite probability that a single changed memory location may be undetected. However, the probability that this happens can be made arbitrarily low as by increasing the number of memory accesses performed by the verification procedure, i.e., by increasing the number of iterations of the verification procedure loop. The drawback of doing this is that the running time of the verification procedure increases linearly with the number of loop iterations.

Memory architecture dependencies The 8 and 16-bit microcontrollers used in small embedded systems are designed in either of two memory architectures: the Von Neumann Architecture, in which both program code and data reside in a single physical memory; and the Harvard Architecture, in which the program and data memories are distinct. These microcontrollers lack support for virtual memory. Since the verification procedure can directly access and check the entire physical memory, if the result of a verification request is correct, the verifier is assured that the memory contents are correct. Note that an attacker cannot reconstruct the correct memory image before verification and undo changes afterward. Doing this will require the code that does the undo operation to be resident in the memory of the device. This change in memory contents will be detected by SWATT.

In the Harvard architecture, the program memory is separate from the data memory and typically different instructions are required to access the program and data memory. Since the data memory is not executable, we usually only need to verify the program memory. The program memory typically contains executable code and static data. Hence, its contents will be known to the verifier apriori. This makes it feasible to verify a running device.

Microcontrollers that use a Harvard architecture have a non-volatile storage such as flash memory as program memory and SRAM for data memory. The size of the program memory is typically an order of magnitude larger than the size of the data memory. 8 and 16-bit microcontrollers from several popular vendors like Atmel, National Semiconductor, Microchip, Texas Instruments, and Zilog use this approach.

In the Von Neumann architecture, the program and data share the same memory address space. Since we want to verify all code memory, but cannot externally determine how much of the memory is used for the code, we need to verify the entire memory, including all data. Consequently, the verifier needs to know the exact state of the data memory, which includes the program stack. The set of programs that run on an embedded device and their runtime schedules are fixed at the time the embedded device is designed.

So, it should be possible to have checkpoints in the code at which all dynamic state present in memory, with the possible exception of environmentally-influenced state like sensor readings, is externally predictable. If we carry out our verification at these checkpoints, we can verify a running system. In any case, the verifier can download all dynamic content when performing a verification.

Effect of memory size on running time The verification procedure makes O($n \ln n$) accesses to memory to compute the checksum, where n is the memory size. Each memory word read is operated upon in constant time. Hence the running time of the verification procedure grows as O($n \ln n$) in the size of the memory.

Empty memory regions Empty regions of memory are often filled with zeros. So, if an attacker places malicious code in the empty memory regions, it can suppress the read to these memory locations and substitute it with zero. Also, the attacker need not compute the `exor` operation when computing the checksum of a zero-valued memory location. Together, the time saved by not performing these two operations may offset the time for an extra `if` statement. To prevent this attack, we suggest that empty memory regions be filled with a pseudo-random pattern.

CPU Architecture and ISA dependencies Microcontroller architectures vary considerably between vendors. The techniques discussed this section can always be used with Von Neumann architectures. For Harvard Architectures, our technique works if the architecture satisfies the following conditions:

1. If the code and data memories have different word sizes, the word size of the code memory should be smaller than or equal to the width of the datapath of the CPU of the microcontroller. If not, then the code memory should be readable in sizes equal to the width of the datapath.

2. The instruction set of the microcontroller must have an instruction to read words the from the code memory. This can be the move instruction or any other special purpose instruction designed for that purpose.

3. On some Harvard architectures, the loads from program memory take 2 to 3 times as long as loads from data memory. This could be the source of a potential attack. An attacker can keep the copy of the original memory contents in the data memory. Then by diverting the load operation to the data memory, the attacker might be able to offset the extra time taken by the `if` statement. In view of this attack, we require that the difference between the latencies of loads from program and data memory not be greater than the time taken to execute the `if` statement.

3.6 Discussion

The design for the verification procedure, discussed in this section, is central to SWATT. The design ensures that the checksum of memory contents returned by the embedded device will be correct only if the memory contents of the device is the same as the value expected by the verifier. It will be different with high probability if the memory contents of the device differ from the expected contents. This statement about the checksum holds as long as the verifier has the correct view of the embedded device's hardware configuration. So the only way an attacker can hide changes to memory content is to change the hardware of the device. This is typically much more difficult to do than changing memory contents.

An interesting application of SWATT is in virus checking. If an embedded device is suspected to be infected, an external verifier can ship the verification procedure to the device, download the entire memory image from the device, and use SWATT to ensure that the downloaded memory content is indeed the same as on the device. The verifier can then use the downloaded content to perform virus checking locally. This approach will prevent the virus from interfering with the virus checker, if we were to run the checker on the infected device. Note that the verification procedure can be shipped to the device even after the device is infected. The design of SWATT ensures that the device can return the correct checksum within a specified time frame only if runs the correct verification procedure. If the device is found to be infected, it can be patched to remove the virus. SWATT can be used to verify that the installation of the patch was successful.

One vulnerability of SWATT is that time at which the memory is verified is not the same as the time at which the device is used. So there is a possibility that an attacker changes the memory contents of the device between verification and use. It is an open research problem how to deal with this weakness.

4 Related Work

The IBM 4758 secure cryptographic coprocessor [12, 13, 14] runs a general purpose operating system and allows field upgrades of its software stack. To ensure the integrity of the system it uses a form of secure boot [2, 3] that starts from an initial trusted state and each layer verifies the digital signature of the next layer before executing it. This ensures that the software stack has not been altered.

Systems such as TCG (formerly known as TCPA) [16] and NGSCB (formerly known as Palladium) [11] use essentially the same notion to bootstrap trust but the mechanisms are very different. TCG and NGSCB measure the integrity of the various components using a secure hash function (SHA-1) and the result is stored in a separate secure coprocessor. This coprocessor can attest to these measurements by signing them with the attestation identity key that is stored inside the coprocessor. What is measured differs per system, TCG starts measurement from system boot and NGSCB starts measuring when the Nexus takes control.

SWATT does not need a secure coprocessor and allows a trusted external entity to verify the memory contents of an embedded device using a software-based technique. Once the memory contents are verified, it forms the trusted computing base. Hence we bootstrap trust entirely in software.

Kennell and Jamieson propose techniques to verify the genuinity of computer systems entirely in software [9]. As we discuss in Section 1, their technique cannot be used on embedded systems. They always send their checksum code as part of the challenge, which introduces vulnerabilities due to the threats of mobile code. Their technique is similar in that they also compute a randomized hash of the memory, and also use timing to detect genuinity. However, both of these techniques serve different purposes than in SWATT: the randomized memory access is used to trigger more page faults and cache misses (and not as in our technique to force insertion of an `if` statement to slow down the attacker), and the timing based approach assumes that it would take longer to simulate the hardware on another device. Thus, their mechanisms are different from ours as well as their target platform. They presuppose a virtually paged architecture and the availability of low-level CPU performance counters to measure the effect of instruction and data TLB replacements. These kinds of architectural features are typically only available on high end CPUs and not on small embedded devices.

In addition, Kennell and Jamieson's technique suffers from a security vulnerability that enables an attacker to change an arbitrary number of memory locations and remain undetected with a 50% probability. The attack proceeds as follows. They compute a 32-bit checksum by adding 32-bit memory words read into the current value of the checksum, where the traversal is influenced by cache and TLB misses. Periodically, they also XOR the current values of cache and TLB misses into the checksum. Thus their checksum function can be described as
$checksum = checksum + [MemoryLocation]$
$checksum = checksum \oplus CacheMisses || TLBMisses$
With this method of computing the checksum, an attacker can flip an arbitrary number of most significant bits (MSB) of 32-bit words and the resulting checksum will still be correct with probability 50%. Only the MSB has this property, as the carry bit of the MSB produced by an addition is lost. Thus, if an odd number of changed locations are included in the checksum, the resulting checksum will have the MSB flipped, however, if an even number of changed locations are included, the resulting checksum is correct.

5 Conclusion and Future Work

SWATT is a technique for externally verifying the code, static data and configuration settings of an embedded device. Central to our technique is a carefully constructed verification procedure that computes a checksum over memory in such a way that an attacker cannot alter the content of that memory without changing the externally observed running time of the verification procedure while still producing the correct checksum. In particular, we use a randomized access pattern to force the attacker to insert check statements before every memory access if the memory was altered. We have presented a practical implementation of such a procedure and provided a detailed analysis of it.

Our future work concentrates on how to perform secure device verification remotely, over an untrusted network. We are also working to expand verification to CPUs with sophisticated architectural features like virtual memory and branch predictors. We hope that our work motivates further research on this important problem.

6 Acknowledgments

We would like to thank Dan Boneh, David McGrew, David Maltz, Robert O'Callahan, Mike Reiter, Adi Shamir, Bhaskar Srinivasan, and Brian Weis for stimulating discussions, and suggestions on how to improve this paper. We also thank the anonymous reviewers for their comments and suggestions.

References

[1] Elise Ackerman. Voting machine maker dinged. `http://www.mercurynews.com/mld/mercurynews/business/technology/7511145.%htm`, Dec 2003.

[2] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A reliable bootstrap architecture. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 65–71, Oakland, CA, May 1997. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press.

[3] William A. Arbaugh, Angelos D. Keromytis, David J. Farber, and Jonathan M. Smith. Automated recovery in a secure bootstrap process. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS '98)*, pages 155–167, San Diego, California, March 1998. Internet Society.

[4] N. Ferguson, D. Whiting, B. Schneier, J. Kelsey, S. Lucks, and T. Kohno. Helix: Fast encryption and authentication in a single cryptographic primitive. In *Proceedings of Fast Software Encryption (FSE2003)*, pages 345–362, February 2003.

[5] Scott Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of RC4. In *8th Annual Workshop on Selected Areas in Cryptography*, volume 2259 of *Lecture Notes in Computer Science*, pages 1–24, Toronto, Canado, August 2001. Springer-Verlag, Berlin Germany.

[6] FX and kim0. Attacking networked embedded systems. In *Black Hat Briefings, Las Vegas 2002*, 2002. Presentation available at `http://www.blackhat.com/presentations/bh-asia-02/bh-asia-02-fx.pdf`.

[7] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[8] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: a goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 304–314. ACM Press, 2002.

[9] Rick Kennell and Leah H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of the 11th USENIX Security Symposium*. USENIX, August 2003.

[10] Alexander Klimov and Adi Shamir. New cryptographic primitives based on multiword t-functions. Invited talk at the Fast Software Encryption Workshop 2004. `http://www.wisdom.weizmann.ac.il/~ask/`.

[11] Next-Generation Secure Computing Base (NGSCB). `http://www.microsoft.com/resources/ngscb/default.mspx`, 2003.

[12] S.W. Smith, E. Palmer, and S.H. Weingart. Using a high-performance, programmable secure coprocessor. In *2nd International Conference on Financial Cryptography*, 1998.

[13] S.W. Smith, R. Perez, S.H. Weingart, and V. Austel. Validating a high-performance, programmable secure coprocessor. In *22nd National Information Systems Security Conference*, October 1999.

[14] S.W. Smith and S.H. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks (Special Issue on Computer Network Security)*, 31:831–960, 1999.

[15] Superopt - finds the shortest instruction sequence for a given function. `http://www.gnu.org/directory/devel/compilers/superopt.html`.

[16] Trusted Computing Group (TCG). `https://www.trustedcomputinggroup.org/`, 2003.