Using Software-based Attestation for Verifying Embedded Systems in Cars

Arvind Seshadri
Carnegie Mellon University
arvinds@cs.cmu.edu

Adrian Perrig
Carnegie Mellon University
perrig@cmu.edu

Leendert van Doorn
IBM & Carnegie Mellon University
leendert@ece.cmu.edu

ABSTRACT

With advances in automobile electronics, we find a rapid proliferation of embedded systems in cars, both in safety-critical applications and for passenger comfort. These embedded systems are increasingly networked for their operation and enhanced functionality. However, the increased connectivity of embedded systems also greatly complicates design, increases the number of failure modes, and introduces the risk of remote malicious attacks, such as worms and viruses. Moreover, car owners may alter the code on their car to access features they did not pay for or achieve higher motor performance. Such owner-initiated changes are likely to deteriorate the car's safety.

We propose SWATT, a SoftWare-based ATTestation mechanism to detect and defend against these threats. SWATT enables an external verifier to verify the code of a running system to detect maliciously inserted or altered code. So far, *code attestation* has been proposed as a mechanism to verify the code running on a system, and special hardware mechanisms have been designed to achieve this property, e.g., TCG (formerly known as TCPA) [15] and NGSCB (formerly known as Palladium) [6]. However, special hardware to provide attestation may not be available in legacy systems or due to cost reasons. Therefore, we design SWATT to be software-based. Code attestation is instrumental to many applications, such as remote detection of malicious code (such as Trojan horses and viruses) in embedded systems and gives an assurance that critical embedded systems are running the correct code. If we use SWATT to verify code running on embedded systems in a car, an attacker is forced to perform a hardware change to hide the presence of altered code; greatly increasing the effort required by an attacker and preventing entire classes of remote attacks.

1 INTRODUCTION

The number of embedded systems in cars is expanding. These embedded systems are increasingly networked for enhanced functionality and firmware upgrades. Networking these devices has both positive and negative aspects: on the positive side networking enhances capabilities, but on the negative side networking greatly complicates design, increases the number of failure modes, and introduces the risk of remote malicious attacks, like worms and viruses. For example, recently, we have witnessed the "cabir" worm that infects cell phones using their bluetooth interface [3], and we could easily imagine worms that use similar mechanisms to propagate from car to car on a highway. Another risk are car owners that alter the firmware of embedded car systems to gain access to additional features, for example use the GPS technology of the General Motors OnStar system without paying for the service [8]. It is clear that unauthorized changes to firmware may interfere with other car systems and thus reduce the car safety. Finally, car owners that want to get more power out of their motor may alter the firmware of the engine controller to change the CAM and valve timings [11]. If done incorrectly, such timing changes can cause catastrophic engine damage, and even result in engine explosion.

Software updates are often used in embedded systems to patch bugs in existing code or to add greater functionality. The update mechanism also introduces a security vulnerability. An attacker may exploit the update mechanisms to inject malicious code into the device. This attack becomes increasingly serious, as we continue to network embedded systems and connect them to the Internet.

So far, *code attestation* has been proposed as a mechanism to verify the code running on a system, and special hardware mechanisms have been designed to achieve this property, e.g., TCG (formerly known as TCPA) [15] and NGSCB (formerly known as Palladium) [6]. However, special hardware to provide attestation may not be available in legacy systems or due to cost reasons.

We designed and implemented SWATT, a software-based approach to the problem of verifying the code running on an embedded system [9]. SWATT enables an external verifier to verify the code running on a system, without direct physical access to the embedded system'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, instruction set architecture (ISA) and the memory architecture of the microcontroller on the embedded device, and the size of the device's memories. To attack SWATT, an attacker would need to change the hardware of the device. This significantly increases attacker effort, previously an attacker simply needed to upload malicious firmware, but with SWATT the attacker also has to modify the hardware.

As a first step, we implemented SWATT on the Mica Mote sensor network devices [4]. The motes have an 8-bit microcontroller with no virtual memory support. Many of the embedded systems used in cars, such as the engine timing controller, are based on these kind of small microcontrollers. Hence, our approach is directly applicable to car-based embedded systems.

Outline In Section 2, we give a problem definition and describe the attacker model. Section 3 presents our general approach and describes the implementation of SWATT on the mica mote sensor network devices. In Section 4 we discuss related work, and we present our conclusions in Section 5.

2 PROBLEM DEFINITION, ASSUMPTIONS, & THREAT MODEL

Software attestation is a method to externally verify the code running on an embedded device, without physical access to the memory. Consider the setting that Figure 1 shows. We assume that a *verification device*, which we call the *verifier*, wants to check whether the code memory contents of an *embedded device*, which we refer to as the *device*, is the same as some expected value. We assume that the verifier knows the expected code memory contents. For embedded systems used in cars, the verifier could be the manufacturer or another authorized entity. So the expected value of the device's code memory will be known to the verifier. The goal is to design an effective *verification procedure* such that it will succeed if the code memory contents of the device is the same as the expected value, and it will fail with extremely high probability if the code memory contents of the device differs from the expected value even by a single byte. We say a verification procedure with this property is a secure verification procedure.

We assume that the device contains a memory content verification procedure that the verifier can activate remotely. (Alternatively, this procedure could also be downloaded any time prior to the verification.) 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.

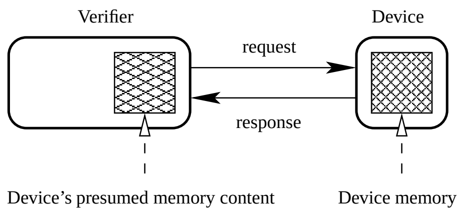

Figure 1: Generic external memory verification. The verifier has a copy of the device's presumed memory, and sends a request to the embedded device. The device can prove its memory contents by returning the correct response.

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, or increase the clock speed of the processor. That is, we assume that the verifier knows the exact hardware configuration of embedded device. Also, it is assumed that impersonation and proxy attacks, where the response to the challenge from the verifier is computed by another entity on the device's behalf, are not possible.

2.1 Naive Approaches and Attacks

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 2 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.

3 SWATT: Software-based Attestation

In this section, we first discuss our general approach. We then briefly describe the implementation of the SWATT memory content verification procedure on the mica mote sensor network devices, giving the assembler code. Our publication describes SWATT in greater detail and also shows the results of our experiments [9].

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, in our implementation on the mica mote sensor nodes, the slowdown was 13%.

3.2 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 16 Kbytes of program memory and 1 Kbyte of data memory [4]. The CPU on the microcontroller has a RISC architecture. We first describe our design and then show its realization in assembly language of the ATMEGA163L.

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.

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. Figure 3 shows the pseudo code of our implementation.

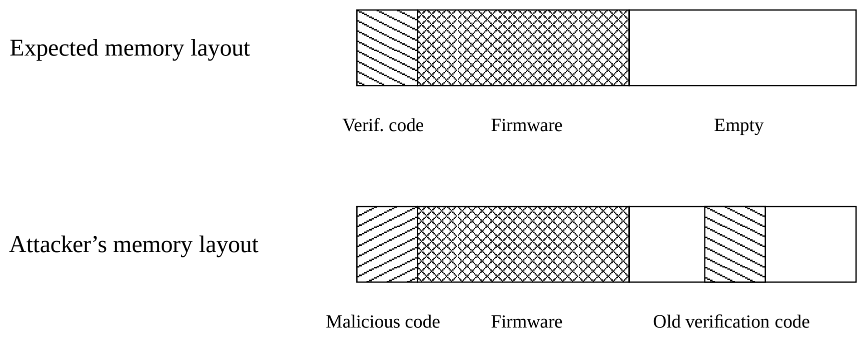

Figure 2: Memory verification attack. The attacker replaces the verification code with malicious verification code and copies the old verification code into empty memory.

```
**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)

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 $\oplus$ 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)

Assembly code Figure 4 shows the assembly code, written in the assembly language of the Atmel ATMEGA163L microcontroller.

The architecture of the microcontroller has the following characteristics:

- The microcontroller has a Harvard Architecture, with 16 Kbytes 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 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, predecrement 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. We show through experiments that this overhead is externally detectable [9].

3.3 Discussion

SWATT provides code attestation without requiring any hardware modifications. This is an important advantage, as owners, car manufacturers, or service stations could verify the firmware on embedded systems in the car without having direct access to the memories. SWATT could be used today on current cars, for example to detect firmware alterations on the engine controller. For future networked cars that may be connected to the Internet, we could use SWATT to detect maliciously injected code, and to ensure that the correct code is running on the system. As an example, the car key could perform code attestation every time we start the engine, by storing challenge-response pairs in the key. Since SWATT is purely software based, it benefits from a lower deployment cost than hardware based approaches.

4 RELATED WORK

Paar and Wollinger discuss security risks in cars and show how security and cryptography can be used to achieve access control, theft, anonymity, reliable communication, content protection, and legal controls [7]. In addition to cryptographic processes, we propose SWATT in this article to further control these risks and protect against changes of the car's firmware.

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 [1, 2] 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) [15] and NGSCB (formerly known as Palladium) [6] 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 code on a device using software techniques. Further, this verification need not be done at device boot but can be carried out whenever the external verifier wishes to do so. Once the code is verified, it forms the trusted computing base. Hence we bootstrap trust entirely in software and provide gurantees similar to TCG or NGSCB, without requiring secure hardware. Therefore, SWATT can be used with legacy systems that do not have secure hardware and with systems that lack secure hardware due to cost concerns.

Kennell and Jamieson propose techniques to verify the genuinity of computer systems [5]. 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. Unfortunately, their technique suffers from security vulnerabilities [9, 10].

5 CONCLUSION

SWATT is a software-based technique for externally verifying the code running on 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.

SWATT is targetted towards embedded systems, without secure hardware. It does code attestation and provides properties similar to TCG and NGSCB without requiring secure hardware. Thus, SWATT can be used to detect malicious code. Hence, it can be used as a primitive for building high confidence embedded systems in a networked environment, where risk of attacks due to malicious code is significant.

If we use SWATT to verify code running on embedded systems in a car, an attacker is forced to perform a hardware change to hide the presence of altered code; greatly increasing the effort required by an attacker and preventing entire classes of remote attacks.

REFERENCES

[1] 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, May 1997.

[2] 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, March 1998.

[3] Celeste Biever. First cell phone worm emerges. `http://www.newscientist.com/news/news.jsp?id=ns99995111`, June 2004.

[4] 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.

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

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

[7] Christof Paar and Thomas Wollinger. Eingebettete sicherheit und kryptographie im automobil: Eine einführung. In *Workshop Automotive SW Engineering and Concepts*, October 2003.

[8] John Schwartz. This car can talk. what it says may cause concern. `http://www.nytimes.com/2003/12/29/technology/29car.html`, December 2003.

[9] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Swatt: Software-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

[10] Umesh Shankar, Monica Chew, and J. D. Tygar. Side effects are not sufficient to authenticate software. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[11] Peter Shearman. The black mystic art of cam timing! `http://mail.symuli.com/vw/camp1.html`, `http://mail.symuli.com/vw/camp2.html`, `http://mail.symuli.com/vw/camp3.html`, 1997.

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