Exécution sécurisée de code sur systèmes embarqués

Daniele Perito

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7/08/2006

Présentée par

Daniele Perito

Thèse dirigée par **Claude Castelluccia**

préparée au sein **INRIA**
et de **Université de Grenoble**

Garanties d'exécution de code sur systèmes embarqué - Trustworthy Code Execution on Embedded Devices

Thèse soutenue publiquement le **13/10/2011**,
devant le jury composé de :

Pr. Andrzej Duda
Grenoble INP-Ensimag, Président
Pr. Peter Langendoerfer
IHP, Rapporteur
Pr. Refik Molva
Eurecom, Rapporteur
Pr. Srdjan Capkun
ETH, Examinateur
Pr. Ahmad-Reza Sadeghi
TU-Darmstadt, Examinateur
Dr. Claude Castelluccia
INRIA, Directeur de thèse

Abstract

Embedded devices are currently used in many critical systems, ranging from automotive to medical devices and industrial control systems. Most of the research on such devices has focused on improving their reliability against unintentional failures, while fewer efforts have been spent to prevent intentional and malicious attacks. These devices are increasingly being connected via wireless and connected to the Internet for remote administration, this increases the risk of remote exploits and malicious code injected in such devices. Failures in such devices might cause physical damage and health and safety risks. Therefore, protecting embedded devices from attacks is of the utmost importance.

In this thesis we present novel attacks and defenses against low-end embedded devices. We present several attacks against software-based attestation techniques proposed for embedded devices. Furthermore we design and implement a novel software-based attestation technique that is immune to the aforementioned attacks. Finally, we design a hardware solution to attest and establish a dynamic root of trust on embedded devices, this solution is proven secure and does not rely on the strong assumptions used for software attestation.

Résumé

Les systèmes embarqués sont utilisés dans de nombreux systèmes critiques, des les automobiles jusqu'aux les systèmes de contrôle industriels. La plupart des recherches sur ces systèmes embarqués se sont concentrés sur l'amélioration de leur fiabilité face à des fautes ou erreurs de fonctionnement non intentionnelles, moins de travaux on été réalisés considérant les attaques intentionnelles. Ces systèmes embarqués sont de plus en plus connectés, souvent à Internet, via des réseaux sans fils, par exemple pour leur administration à distance. Cela augmente les risques d'attaques à distance ou d'injection de code malicieux. Les fautes de fonctionnement de ces équipements peuvent causer des dommages physiques comme par example rendre des appareils médicaux defectueux. Par conséquent, il est primordial de protéger ces systèmes embarqués contre les attaques.

Dans cette thèse nous présentons des attaques et défenses contre les systèmes embarqués contraints. Nous présentons plusieurs attaques contre des techniques d'attestation logicielle utilisées dans les systèmes embarqués. Puis nous présentons

la conception et l'implémentation d'une technique d'attestation logicielle qui est résistante aux attaque présentées précédemment. Finalement, nous présentons la conception d'une solution permettant de réaliser l'attestation de code ainsi que la création d'une racine de confiance dynamique (dynamic root of trust) pour les systèmes embarqués. La sécurité de cette solution est prouvée, et ne repose pas sur des hypothèses fortes, comme dans le cas des solutions d'attestation logicielle.

Acknowledgments

First and foremost I would like to thank my family for their immense and unconditional support. It was a long journey that took me far from home, but they always supported me.

Second, I would like to thank my advisor Claude Castelluccia as I could not have hoped for a better mentor. Claude helped me focus my research efforts, but also left me a great deal of freedom that allowed me to pursue my ideas and travel and collaborate with other teams. Also, I would like to thank Professor Gene Tsudik for his tremendous support. I would like to thank all the members of the Planete team, and especially Aurélien Francillon, for the great discussions we had and the work we have done together. I would also like to thank Helen Pouchot that helped me so much in the organization of my travels and journeys.

Finally, I would like to thank the reviewers of the thesis for their helpful comments and suggestions.

Published work during the PhD

International Conferences

[DFPT12] Karim El Defrawy, Aurélien Francillon, Daniele Perito and Gene Tsudik. *SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust*. Network & Distributed System Security Symposium, NDSS 2012.

[CDP12] Claude Castelluccia, Markus Duermuth, Daniele Perito. *Adaptive Password-Strength Meters from Markov Models*. Network & Distributed System Security Symposium, NDSS 2012.

[PCKM11] Daniele Perito, Claude Castelluccia, Mohamed Ali Kaafar, Pere Manils. *How Unique and Traceable are Usernames?*. Proceedings of the Privacy Enhancing Technology Symposium, PETS 2011.

[BBP+11] Elie Burzstein, Romain Beauxis, Hristo Paskov, Daniele Perito, Celine Fabry, John Mitchell. *The Failure of Noise-Based Non-Continuous Audio Captchas*. Proceedings of the 32nd IEEE Symposium on Security and Privacy, S&P 2011.

[PT10] Daniele Perito and Gene Tsudik. *Secure Code Update for Embedded Devices via Proofs of Secure Erasure*. Proceedings of the European Symposium on Research in Computer Security, ESORICS 2010.

[CCP10] Claude Castelluccia, Emiliano De Cristofaro, Daniele Perito. *Private Information Disclosure from Web Searches* Proceedings of the Privacy Enhancing Technology Symposium, PETS 2010.

[CFPS09b] Claude Castelluccia, Aurélien Francillon, Daniele Perito and Claudio Soriente. *On the Difficulty of Software-Based Attestation of Embedded Devices*. Proceedings of the 16th ACM conference on Computer and Communications Security, CCS 2009.

[CKMP09] Claude Castelluccia, Mohamed Ali Kaafar, Pere Manils, Daniele Perito. *Geolocalization of Proxied Services and its Application to Fast-Flux Hidden Servers*. Proceedings of the ACM Internet Measurement Conference , IMC 2009.

[CFPS09a] Aurélien Francillon, Daniele Perito, Claude Castelluccia. *Defending Embedded Systems Against Control Flow Attacks*. In ACM Conference on Computer and Communications Security, Workshop on Virtual Machine Security, 2009.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

Embedded systems are encountered in many settings, ranging from mundane to critical. In particular, sensor and actuator networks are used to control industrial systems as well as various utility distribution networks, such as electric power, water and fuel. They are also widely utilized in automotive, railroad and other transportation systems. Furthermore they are widely used to control implanted medical devices, such as pacemakers. In such environments, it is often imperative to verify the internal state of an embedded device to assure lack of spurious, malicious or simply residual code and/or data. It is also imperative to prevent remote exploitation when security vulnerabilities (e.g., due to coding mistakes) are present in the code.

In the past, embedded devices used in critical applications were rarely inter connected to the outside world or to other devices, hence they benefited from a blanket protection against remote intrusion. However, in the last few years the increased adoption of open wireless communications for these devices has augmented the risk of remote exploits for such devices.

When embedded devices are used in safety critical applications, software faults caused by malicious activity can have dire real-world consequences. Notably, the recent Stuxnet worm [FMC11] demonstrated the magnitude of damage that could stem from attacks to embedded devices. Stuxnet infected Programmable Logic Controllers (PLC) used in industrial control systems for nuclear reactors. By doing so, it altered the operational parameters of the uranium enrichment centrifuges of one nuclear plant located in Iran, causing permanent damage to the centrifuges. Embedded devices are also used in automotive control systems. Recently, such systems have been shown vulnerable to a variety of attacks [KK10], including malicious re-flashing of in-car electronic safety systems. Protecting such devices from exploitation is still an open research challenge. Another possible application domain is direct recording electronic (DRE) voting machines, that have been shown vulnerable to return-oriented programming attacks in [CFK+09] and other forms of manipulation of the ballot reporting [WWH+10]. These machines are equipped with embedded devices similar to those targeted by our work.

These facts make embedded devices security an important challenge. We believe that, while examples of real life attacks are still limited, it is vital to design and deploy the correct security solutions in a proactive rather than reactive way.

1.1 Context of This Work

The focus of this work is the protection of the execution environment in low-end embedded devices from malicious attacks. In particular, we focus on embedded devices used in cyber-physical systems and used to operate the critical infrastructure.

1.1.1 Low-end Embedded Devices

Embedded devices are computer systems designed to perform one or a limited set of specific functions. They are embedded in the sense that all their parts (e.g. memory and CPU) are contained in one single encasing or chip. Embedded devices usually lack of any user interface and are programmed and debugged using a single debug interface.

In this work we focus on low-end embedded devices, that have hard constraints on their computational capabilities, memory, energy and cost. In particular, all the protocols and techniques presented in this work have been implemented and tested on two devices, the Texas Instruments MSP-430 and the Atmel AVR family of micro-controller units (MCU). These embedded devices rely, respectively, on a 16-bit and an 8-bit instruction set. The MSP-430 is based on a Von Neumann architecture, which means that both instructions and data are stored in a single address space. The AVR instead relies on a Harvard architecture, where instructions and data are stored in physically separate memories.

These micro-controllers are both embedded on one silicon die where the CPU as well as the memories reside (both RAM and flash storage). The same chip also holds external peripherals such as signal converters (digital to analog and analog to digital), bus interfaces (e.g., UART, etc.), with the possibility of attaching network interface (e.g., Zigbee). Encasing all the components in one single die allows to keep productions costs and energy consumption low. Also, since embedded devices are designed to perform specific tasks, their design can be optimized further reducing their final cost.

Embedded devices are used in a variety of applications ranging from safety critical (e.g., energy plants or automotive systems) to mundane (MP3 players and digital watches). In this work we concentrate on the challenges that arise in securing the operation of the former class of applications, where embedded devices are used as part of cyber physical systems and perform as both actuators and sensors.

1.1.2 Cyber-Physical Systems

A cyber-physical system (CPS) is a system where there is tight coordination of the system's computational and physical elements [Lee08]. Cyber-physical systems are composed of three main components: the processing unit, one or more physical sensor (temperature, humidity, etc.) and actuators. Today, CPS are used in a wide variety of applications ranging from automotive, industrial control systems, energy production, health care, aerospace, etc. The definition of cyber-physical systems and embedded devices often overlap, however when referring to cyber-physical systems the stress is on the tight integration between the computing part and the sensors and actuators, rather then the embedded nature of the computation.

There are a number of applications already deployed of cyber-physical systems and more are envisioned to come in the future. Embedded devices are used to control medical devices, such as internal defibrillators and pacemakers or insulin pumps. Most modern cars are largely controlled by, so called, *electronic control devices* that supervision many aspects of the driving ranging from steering to breaking and more. Energy plants of all types (including nuclear plants) rely on embedded devices to monitor and adjust the operational parameters of the systems. Factories rely on embedded devices to control various aspects the production and distribution, called industrial control systems. Cyber-physical systems present a number of interesting challenges that sets them apart from traditional computers and, in 2007, they were identified as as the most important research priority by the NITRD in an address to the US President [NN07].

Cyber-physical systems are required to operate unattended for extended periods of time, possibly in harsh environments. They are often subject to strict reliability and timing constraints. Faults might results in real-world safety risks. The devices can be physically tampered with for malicious purposes. In this work we are interested in exploring the security challenges that arise when embedded devices

are used in cyber-physical systems, especially when these devices are used to operate and control the critical infrastructure.

1.1.3 Critical Infrastructure

The term critical infrastructure is used by governments to identify and describe assets and systems that are essential to the proper functioning of a society and economy. According to a 2006 report to the European Union [fCIP07] member states are advised to identify their critical infrastructure based on criteria that include:

- "Scope - The disruption or destruction of a particular critical infrastructure will be rated by the extent of the geographic area which could be affected by its loss or unavailability.
- Severity - The consequences of the disruption or destruction of a particular infrastructure will be assessed on the basis of several parameters listed below."

In turn severity is rated in terms of: public effect (population affected); economic effect (significance of economic loss and/or degradation of products or services); environmental effect; political effects; psychological effects; public health consequences.

While there is no general consensus on what exactly constitutes the critical infrastructure, based on this definition the following national assets associated with: transport and distribution; telecommunication; electricity generation and distribution; water supply; public health; financial services (banks); security services (police and military) and more.

1.1.4 Embedded Systems Security

In the context of cyber physical systems and the critical infrastructure, embedded devices security poses a number of unique challenges that are different from traditional computing devices.

Energy Constraints Some embedded devices are battery powered and need to operate for extended periods of time. Implantable medical devices, for example, are fully encased in the patient's body and cannot rely on any external energy source, but still need to operate for years without any intrusive replacement. Any security protocol designed for such devices needs to use computation and communication in a very efficient manner to preserve battery life. Furthermore, the security protocols themselves might introduce threats to the availability of the device, if the attacker is specifically interested in exhausting the device's battery[1].

[1]This class of attacks is sometimes called sleep deprivation attacks.

Low Cost Some embedded devices are deployed in large numbers and are therefore very cost sensitive: even a small addition in the design might result in high final costs. An increase in few cents in the production cost might render a system economically infeasible.

Real-time Constraints Most embedded devices operate under tight timing constraints. Security related operation must be designed not to interrupt normal operation. This presents a very hard challenge since many cryptographic primitives (e.g., cryptographic hashes) are extremely difficult to be made efficient.

Physical Attacks Embedded devices might be left unattended in unprotected areas for extended periods of time. This leaves them vulnerable to physical tampering to completely subvert the platform, gain cryptographic keys and more. Embedded devices must be designed to withstand such attacks, or at least make these attacks detectable remotely.

1.2 Problem Statement

As explained, embedded devices are used in a large number of critical applications. It is therefore essential to protect them against malicious attackers. However, given the vast number of possible attacks against low-end embedded devices one is confronted with the problem of designing a set of counter-measures that covers as many attack vectors as possible (ideally all) while remaining efficient and, possibly, low-cost. Several trade-offs, corner cases and implicit assumptions need to be analyzed and made explicit. For example, are the embedded devices tamper resistant? If yes, to which degree and at what cost? If no, are hardware attacks realistic?

The list of such questions is very big and very soon one realizes that the design space is extremely vast. First there is a very large combination of possible adversarial assumptions. Second and perhaps more important, there is a large number of solutions to choose from. These solutions can be implemented in hardware or software; can be implemented on the device or at the network level; they can rely on tamper resistant hardware or not; etc. Luckily, this vast design space has already been charted on commodity devices, where the problem of securing execution has been studied for more than a decade. This means that designing security solutions for embedded devices has the advantage of being able to look at the big picture and choosing the solutions that provide the best trade-offs, for example, in terms of security and efficiency. Also, this approach guarantees that the solutions implemented do not overlap or collide.

One of the great advantages of working with embedded devices is the general lack of legacy constraints. In fact, it is not uncommon for a micro-controller unit to be heavily customized to fit the needs of a specific application. This is in stark

contrast with commodity devices, such as server and desktop computers, where legacy is one of the biggest, if not the biggest, hurdle in designing solutions to the problem of securing execution.

To guide in the design of our solutions we follow a map of possible attacks and, therefore, explore adversarial assumptions.

1.2.1 Code Injection Attacks

Here we list the attacks that we consider and address in this work. They encompass the ways in which malicious code can be injected in an embedded device.

Remote Exploits This form of attacks is perhaps the most well-known and widely studied. In this case, the attacker and the victim device communicate over a communication channel, for example a TCP/IP connection or the system call interface of an operating system. The attacker tries to abuse and subvert the communication protocol and format to expose an implementation bug on the victim device. Once discovered, the bug can be used to subvert the flow of control on the victim and inject malicious code that performs actions chosen by the attacker.

Attacks Against Software Updates Most software systems – and the ones on embedded devices are not an exception – share the need of being amendable to updates. Updates can occur for a number of reason, but most commonly it is due to a implementation bug that needs to be fixed or a feature that needs to be added. This factor has important consequences on security, since, often, there is no immutable trusted code base that can be trusted implicitly. Instead the system needs to be able to accept new code, provided that it comes from a trusted source. This process can be subverted by an attacker, if the right countermeasures are not put in place, and lead to malicious code being injected on the victim device. This type of attack could be possible even if remote exploits are completely mitigated. More subtly, even if the software update protocol is secure, malicious code injection is still possible if the computer that issues the update has been compromised. This type of attacks have been recently discovered in the context of external defibrillators [HRMM+11] and it was the case in the StuxNet outbreak [FMC11].

Attack Against Management Interfaces Furthermore, embedded devices often have debugging and management interfaces, e.g., JTAG interfaces. These interface are frequently used during development and prior to deployment and give highly privileged access to the device and allow to upload new software or change the settings. As explained, embedded devices are often left unattended for long periods of time and therefore these debugging interface might be accessed

by an attacker. Security of the debugging interfaces is enforced in two main fashions, either by preventing un-authorized access by authenticating the legitimate administrators (for example via passwords) or by blocking completely the debug interface when the device goes to a production environment (for example by blowing specific fuses on the device). Neither of these approaches are completely secure though (passwords can be guessed and fuses can be tampered with) allowing malicious code injection.

Injection at Production Time Consider the following situation[2], Alice sells an empty SIM card to Bob. Bob goes ahead and uploads its code on the SIM card. Bob does not see anything wrong, however the card originally contained malicious code and the upload procedure was rigged to allow a malicious piece of code by Alice to be present on Bob's SIM card. This malicious piece of code could remain silent unless a specific event is triggered by Alice.

1.2.2 Attacks Against the Integrity and Confidentiality of the Data

There are other attacks that, albeit not addressed in our work need to be considered in the design of secure embedded devices. Even though these attacks range from logical to physical, we group them because the goal they achieve, which is violating the integrity or confidentiality of the data stored on the embedded devices.

Non Control Data Attacks Non control data attacks have been explored in [CXS+05a] and involve an attacker remotely exploiting an implementation bug, not unlikely the code injection remote exploitation mentioned above. However, the goal of the attacker is not to inject code but to simply change the value of variables used for security purposes. Consider, for example, an attacker that could change the value of a key to a known value, therefore exposing all subsequent communications. Also consider the case of an attacker resetting the value of a flag used for access control purposes. In both these examples, no code is injected on the victim device but a comparable effect is produced for the attacker.

Side Channel Attacks Simple and Differential Power Analysis attacks trace the power consumption of an embedded device and measure the tiny fluctuations that occur when the MCU is computing different instructions or handling different data [MOP07]. By doing so, these attacks are able to recover bits of the secret key. Several related attacks are possible that fall in the broad definition of *side channel attacks*, these include electro-magnetic analysis and timing analysis. Side channel attacks have been originally developed to extract secret from smart cards and therefore, are extremely relevant to embedded devices.

[2]This example is taken from [GN07]

Physical Attacks Physical attacks encompass a wide variety of attack techniques that range from fault injection to the depackaging of the embedded chip to steal its secrets. Fault injection attacks, for example, aim at putting the embedded device in state that can be exploited by the attacker via the manipulation of its surrounding conditions, e.g., manipulating the voltage issued to the CPU. More invasive attacks can modify paths on the circuits of an embedded device to cause faults and recover secrets stored on the device. A complete survey of such techniques is beyond the scope of this work and can be found at [Sko05]. Such attacks can be countered by encasing the embedded device in tamper resistant or tamper evident casing and by including appropriate control logic to detect the unusual conditions exploited in fault injection attacks. Those attacks operate at a different layer than the one analyzed in this work. However, when designing our solutions we considered the amenability to such protective techniques.

1.3 Contributions

Given the landscape defined above we can now proceed to explain where our contributions stand. First, we are interested in investigating techniques to **prevent** code injection via control flow exploitation of low-end embedded devices. Our solution can only help prevent remote exploit attacks (commonly referred as buffer overflow attacks), but it cannot prevent other forms of injection, such as malware introduced at production time. For this reason we also focus our attention to techniques that can **detect** malicious code if present. Hence, we look at techniques to verify the current state of a remote embedded device, namely via *device attestation*. We also concentrate on techniques to establish a *dynamic root of trust*, i.e., executing a small trusted piece of code untampered even when the embedded device has been compromised. This small trusted piece of code can, if needed, perform device attestation.

In doing so we focus on techniques that minimize hardware modifications to existing embedded devices in order to reduce cost. Furthermore, we strive to design solutions that have a small footprint and overhead on the normal operations, realizing how security protocols might interfere with the normal operations of the cyber-physical systems.

The contributions of this work are manifold:

- First, we present a lightweight hardware modification to the AVR MCU family, called IBMAC, that helps preventing remote exploitation of buffer overflows on embedded devices (Chapter 2).
- Then we thoroughly review several protocols proposed in the literature to perform software attestation of embedded devices. We find that most of the proposed solutions are vulnerable to several attacks and do not provide the claimed guarantees (Section 3.2).

- We therefore design PoSE, a software attestation solution that overcomes the limitation of the previously proposed protocols (Section 3.3).
- Finally, we present SMART that uses a hardware/software co-design to guarantee the execution of a piece of code on a remote embedded device (Section 4). This small piece of code can perform device attestation, if needed.

Chapter 2

Protecting Embedded Devices against Remote Attacks

Contents

This chapter presents a control flow enforcement technique based on an Instruction Based Memory Access Control (IBMAC) implemented in hardware. It is specifically designed to protect low-cost embedded systems against malicious manipulation of their control flow as well as preventing accidental stack overflows. This is achieved by using a simple hardware modification to divide the stack in a data and a control flow stack (or return stack). Moreover access to the control flow stack is restricted only to return and call instructions, which prevents control flow manipulation. This *Return Stack* is stored in data memory at a different location than the normal stack and is protected in hardware against accidental or malicious modification. Our approach is binary compatible with legacy applications, only

requires minimal changes to the tool-chain and does not the increase memory requirements. Additionally, it makes an optimal usage of stack memory and prevents accidental stack corruption at run-time. The solution is implemented on the AVR micro-controller using both a simulator and a full implementation on an FPGA. The implementation on reconfigurable hardware showed a small resulting overhead in terms of number of gates, and therefore a low overhead of expected production costs.

This demonstrates the possibility to implement this feature with a modest overhead in terms of logical elements units, with no run-time impact, and backward compatibility on all major software functionality. In order to support this feature the device needs application specific configuration to be performed at boot time. This configuration is performed during the boot of the software on the device. This configuration is performed during the very first step of software initialisation and therefore can be performed by the C library after basic initialisation of memory. Apart from this change the compiler libraries and programs do not need modifications.

2.1 State of The Art

By control flow exploitation we mean all the techniques and attacks that try to subvert the normal flow of control of a program and inject malicious code. These attacks are aimed at gaining remote and privileged access to a computer systems by subverting the data structures that regulate the control flow of a program (e.g., the return addresses stored on the stack). Programs that do not implement appropriate checks allow attackers to remotely write data beyond the normal boundaries and thus overwrite control structures in the program. This enables the attacker to execute code with the same privileges as the exploited code.

In this section we will survey the current state of the art of both prevention as well as exploitation techniques. We begin by describing older attack techniques first and introduce the general problem. We then move to more recent attack techniques and finally describe the prevention mechanisms that have been proposed.

2.1.1 Control Flow Exploitation

2.1.1.1 Buffer Overflows

Buffer overflows were first described by Spafford in [Spa89]. Spafford reports his efforts in reverse engineering the remote infection of a BSD derived versions of UNIX that occurred in November 1988. Spafford finds that a *worm* propagated through the network by exploiting a bug in the implementation of the *fingerd* demon that allowed remote code injection and the execution of a UNIX shell. These types of attacks, later called buffer overflows (or buffer overruns), did not become popular until later in the nineties following the publication of [One96, Mud95].

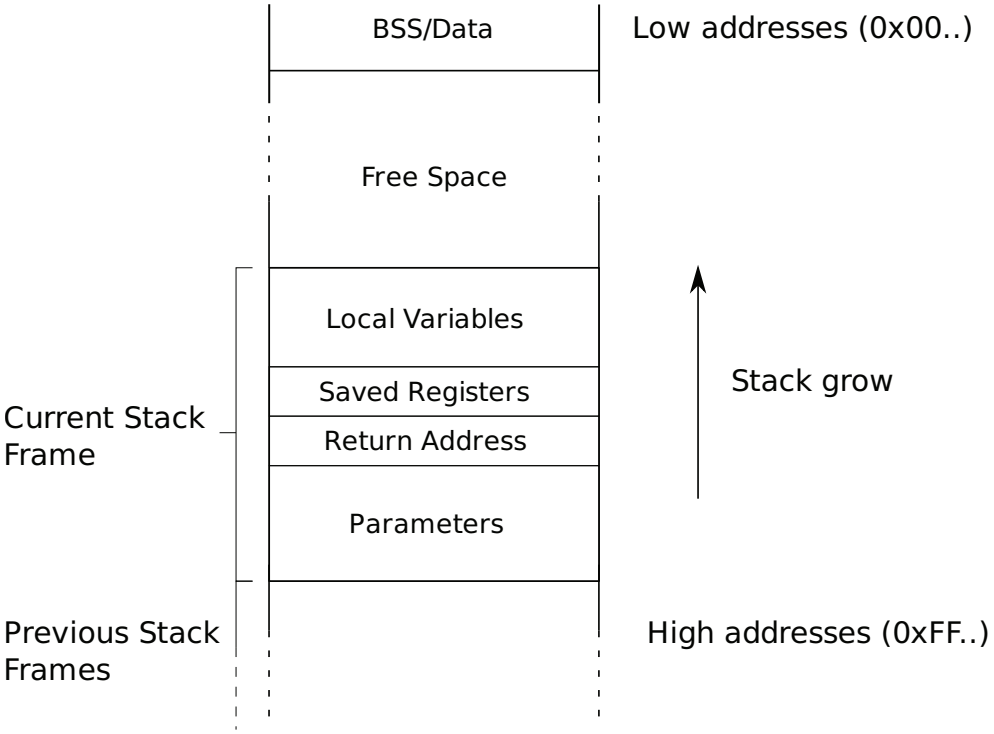


Figure 2.1: Common stack layout

In order to understand how buffer overflows work one needs first to understand how functions and procedures are managed in most systems and programming languages [Ale05]. When a piece of code calls a function the following steps take place:

1. the calling procedure pushes arguments on the stack in reverse order;
2. the calling procedure executes a `call` instruction that stores the address of the calling procedure on the stack and then jumps to the called function;
3. the called function saves the old frame pointer on the stack and then the stack pointer is decremented to make room for the local variables.

In figure 2.1 there is a depiction of the resulting typical stack layout when a function call occurs. Now consider the following code:

```
#include <stdio.h>

int main (int argc, char *argv[]) {
    char buf[16];
    strcpy(buf, argv[1]);
    return 0;
}
```

This code is trivially vulnerable to a buffer overflow attack. In fact, according to the ANSI C specifications the `strcpy` function copies characters in the destination buffer `buf` until a `'\0'` character is encountered. However, if an attacker provides a string that is longer than the allocated length, `buf` is overflown and data adjacent to `buf` in memory can be overwritten. Specifically, the attacker can overwrite the saved return address on the stack. By carefully crafting the content of the input buffer the attacker can divert the return address, inject its own code and make the process execute arbitrary code. This arbitrary code is usually referred to as *payload* or *shell code*.

However, the attack is slightly more complicated than this simple explanation. In fact the attacker has to know the current address of the stack frame to overwrite the return address in memory with the correct value. A common technique is to include a *NOP sledge* in the payload: a NOP sledge is a series of NOP instructions that have the sole purpose of increasing the attacker's chances of correctly guessing the address of the payload. In fact, if the guessed address points to any location within the series of NOPs, execution will continue through the NOPs and eventually reach the payload. This way the attacker does not need to know the exact location of the payload but only its approximate location to fall within the sequence of NOPs.

2.1.1.2 Return to libc

The buffer overflow technique described above assumes the data on the stack to be executable, this way the attacker can inject arbitrary code on the stack and execute it. However, this assumption is not always true. Harvard architectures (like the AVR used in this work) explicitly distinguish code and data. This makes the data non executable. Furthermore, one of the first techniques proposed to prevent buffer overflows (described later in Section 2.1.2.1) is designed to prevent the execution of the data on the stack.

In order to circumvent this limitation, soon a new class of attacks was discovered [Sol97b]. In this class of attacks the attacker does not inject any payload code on the stack, since its execution would be impossible. Instead, the attacker overwrites the return address on the stack with the address of a function already present in the address space of the vulnerable program. The attacker can manipulate the parameters stored on the stack to pass parameters to the target function appropriately. The target function can be any function that has been loaded inside the address space of the attacked process.

Since most C programs load the C library by default (`libc`) this class of attacks often used functions in the C library and was called *return to libc* or *return-into-libc*.

2.1.1.3 Return-Oriented Programming

The return to libc attack is somewhat limited to the functions already present in the program's memory and, in general, does not allow the attacker to execute arbitrary code. To counter the effectiveness of this attack, for example, certain security critical functions (like the UNIX `system` call) could be removed entirely from the program's code making the exploit more difficult to accomplish.

In order to circumvent these limitations Shacham introduced a technique called return-oriented programming (ROP) [Sha07] to exploit vulnerable programs. Instead of executing a complete function present in the address space of a vulnerable program, ROP aims at discovering very short sequences of assembly instructions (two or three instructions long) that terminate with a return. These short sequences, called *gadgets*, provide the basis for a return oriented programming attack. In fact, by carefully crafting the data injected on the stack, a chained sequence of gadgets can be executed within the attacked process to perform arbitrary computation. Later the technique was expanded and augmented to perform attacks without relying on the return instruction [CDD+10].

2.1.2 Control Flow Protection

Control flow attacks have been predominant in many forms since their first appearance. Many different techniques have been proposed to prevent the remote exploitation of vulnerable code, mostly on commodity computers like servers and desktop class machines.

2.1.2.1 Non-Executable Data

One of the first methods to defeat stack based buffer overflows was to make the stack non-executable [Sol97a]. This way even if the attacker is able to inject code on the stack, she cannot execute it. This eventually evolved into what is now referred to as DEP – Data Execution Prevention – (also referred to as X ^W) that is now included in most modern operation systems. In DEP the data memory pages are marked as non-executable either using hardware or software support. When using hardware support, a Non Execute (NX) flag is set for the memory pages of the stack. The processor in turn refuses to execute such memory locations preventing simple stack based overflow. It must be noted, however, that this technique does not prevent other types of exploits like, for example, return-to-libc or ROP attacks.

2.1.2.2 Control Flow Integrity

In Control Flow Integrity, Abadi et al. [ABEL05] propose to embed additional code and labels in the code, such that at each function call, or return, a program is able to check whether it is following a *legitimate* path in a precomputed control

flow graph. If the corruption of a return address occurs, that would make the program follow a non legitimate path, then the execution is aborted as malicious action or malfunction is probably ongoing. The main drawback of the approach is the need for instrumentation of the code, although this could be automated by the compiler tool-chain, it has both a memory and computational overhead and thus might be infeasible on resource constrained devices.

2.1.2.3 Stack Canaries

Canaries were proposed as a solution to buffer overflows in [BST00]. The authors propose to place a *canary* value between the return pointer and local function variables. The value of the canary value is set in the prologue of each function and is checked for validity in the epilogue. In cases when the canary value is altered, there is a clear indication that a memory corruption has occurred and appropriate actions can be taken by the epilogue function, like terminating the process and logging the event.

Canaries introduce a measurable overhead in the normal operation of a program, as the prologue and epilogue are instrumented inside the code at compilation time. Furthermore, canaries have been shown to have a number of vulnerabilities [Ale05]. For example, if the attacker is able to find a double corruption then it can first corrupt a pointer to point past the canary and then modify the return pointer on the stack without modifying the canary.

2.1.2.4 Address Space Layout Randomization

Address space layout randomization [The] can hinder control flow attacks. It is a technique where the base addresses of various sections (`.text`,`.data`,`.bss`, etc.) of a program memory are randomized before each program execution. The memory corruption is not prevented with this technique but exploitation of the vulnerabilities is made considerably more difficult. In fact, the attacker has to correctly guess the randomized address of the target code to execute to complete the attack.

Although, in [SPP+04] show that the effectiveness of address-space randomization is limited on 32-bit architectures by the number of bits available for address randomization. This problem would be even more severe on embedded systems that typically have a 8-bit or 16-bit address space.

2.1.2.5 Protecting the Return Stack

In [Sta] the authors present StackShield that uses a compiler supported return stack. Where the compiler inserts a header and a trailer to each function in order to copy to/from a separate stack the return address from/to the normal stack. As this is implemented at the compiler level there is no backward compatibility, the programs need to be re-compiled with this modified compiler. Moreover, as

additional instructions are introduced there is non negligible a computation and memory overhead.

Furthermore in [XKPI02] the authors propose a return stack mechanism where dedicated `call` and `ret` instructions store and read control flow information from a dedicated stack. However the only guarantee for this return stack integrity is that is located far away the normal stack, which does not prevent modification of the return stack, it just makes it more difficult. Double corruption attacks [Ale05] would allow an attacker to corrupt a data pointer first and then modify an arbitrary memory location on the return stack.

A number of systems already use a separate control flow stack like the PIC micro-controller (for example the pic16[bbM]) or some AVR chips (AVR AT90S1200 [08302]). However those solutions are not designed to improve security. They either allow direct modification of the hardware stack (vulnerable to double corruption) or have a limited stack stored inside the MCU (very limited call depth). For example the AVR AT90S1200 has a return stack supporting only 3 re-entrant routines, if more than 3 re-entrant interrupts or functions calls are performed the hardware return stack is corrupted.

2.2 Instruction Based Memory Access Control for Control Flow Integrity

2.2.1 Overview of our solution

The main idea behind IBMAC is to protect return addresses on the stack from being overwritten with arbitrary data. By doing so, as we will show later, IBMAC also protects embedded systems from memory corruption caused by stack overflows.

The intuition is that control flow data should be only read and written by the `call` and `ret` family of instructions and modifications by other instructions should be prevented. Hence, restricting access to return addresses to `call` and `ret` instructions in hardware seems only logical. However in a normal stack layout, return addresses are interleaved to other types of data, making access controls difficult. In fact, such a fine grained access control would be slow and would lead to a considerable memory overhead, since all the words in memory that have to be protected would need to have an additional flag bit.

That is the main reason why we decided to modify the stack layout adding an additional *Return Stack*, specifically designed to store only return addresses. However, changing the memory layout could have lead to major compatibility issues. That is why our principal design goal was to have a very simple hardware implementation, without extra memory requirement and focused on compatibility. The result is that IBMAC does not require modifications to the tool-chain and most binary libraries could be used without being rebuilt. IBMAC also improves software reliability as stack memory over-consumption [RRW05] can be detected

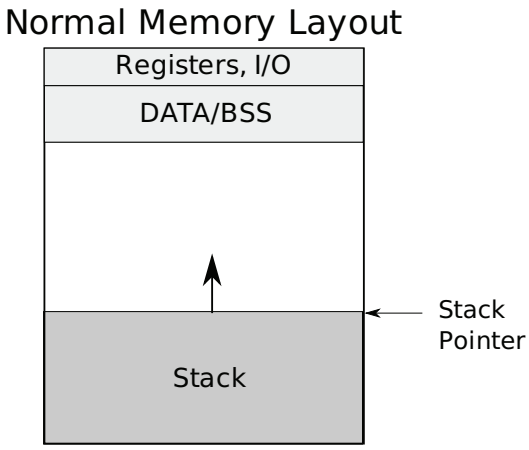


Figure 2.2: Traditional stack layout

at run-time so that a reboot or other actions can be performed (e.g. dedicated interrupt).

Finally we implemented IBMAC as an optional feature that can be activated for example with a write-once configuration register at boot[1]. With those constraints fulfilled and a proven implementation, we believe that this is a very realistic scheme with limited production costs and significantly increased security.

2.2.2 A separate return stack

In Figure 2.2 an architecture with a single stack is shown. While it is convenient to have a single stack, it makes it very difficult to protect the stored return addresses. We therefore implemented a modification to the instruction set architecture in order to support the use of two separate stacks: a *Return Stack* and a *Data Stack*. The return stack is used to store control flow information (return addresses) and the data stack is used to store regular data.

There are several different possible layouts in which those two stacks could be arranged in memory. The arrangement chosen in our implementation is depicted in figure 2.3. The first thing to note comparing figure 2.2 and 2.3 is that the data stack lies where the original single stack was. This was the best solution to maximize backward compatibility, as with this layout the allocation of the stack works in exactly the same way as before and no modifications to the compiler are necessary.

The second thing to note is that the return stack and the data stack grow in opposite directions. This was done in order to optimize memory consumption, as with this layout no memory is wasted in comparison with the original stack layout. The fact that the return stack grows in the opposite direction does not

[1]This could be a *fuse* register on the AVR for example, as fuses cannot be modified without physical tampering.

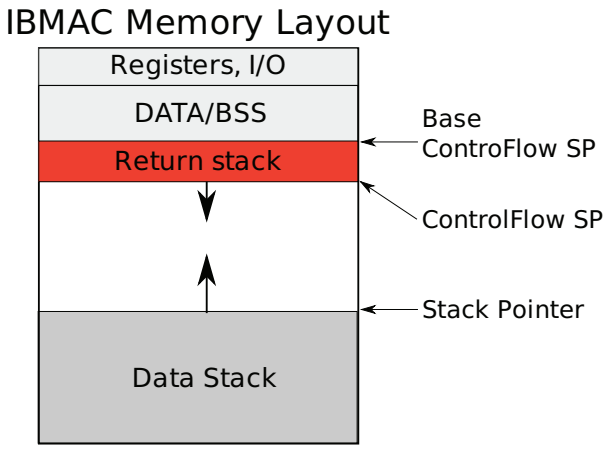


Figure 2.3: IBMAC stack layout. The Base control flow stack pointer is the only register that needs to be initialized in order to support IBMAC.

hinder backward compatibility, as this stack is exclusively handled in hardware by the modified `call` and `ret` instructions.

The third thing to note is that the return stack does not have any static limitation, but instead is only limited by the data stack. However this can also be a drawback as it those not leave room for an unbounded *heap*. In section 2.2.4 we discuss this problem in more detail.

2.2.3 Instruction Based Memory Access Control

The separate return stack layout presented in the previous section provides an easy way to separate control flow information from regular data allocated on the stack. However, it does not prevent modification and corruption of control flow information, but only makes it a bit more difficult as control flow data is not close to stack allocated buffers. Complex attacks would still be able to exploit the return stack. For example if an attacker is able to corrupt the pointer to an array and to further write data to this array she would be able to write data at an arbitrary memory location.

This is the reason why an extra protection layer for the return stack is required. On a general purpose operating system this could be provided by an MMU. However, those are generally not available on low-end embedded devices. The reasons for that are multiple: first, those MCUs are designed for to be at a very low price range, each additional feature comes at an increase of the silicon size and consequently increase the final manufacture price. Second, they are usually designed to execute monolithic applications, therefore they do not require memory protection between different applications or the application and a kernel. The challenge is therefore to protect only the return stack at a very small cost, which is not the case with a complete Memory Management Unit.

Our hardware modification has been designed around the following considerations:

- only control flow related instructions will need to modify the control flow stack
- the data manipulation instructions do not need to access control flow information

Given this observations it is possible to control memory accesses and decide whether to grant or refuse access to the return stack based on which instruction is performing the memory access. On the AVR we used, we identified only two instructions that needed to be able to access the return stack, namely the `call` and the `ret` instructions and their derivatives. The hardware implementation of these two instructions has been modified in such a way to set an internal flag to 1 whenever they are executed. When this signal is high memory access is granted to the control flow stack. If not, the system is rebooted (or could alternatively trow a dedicated interrupt).

2.2.4 Other design considerations

Dynamic memory allocation is one of the basic building blocks of modern operating systems and programing languages. However, it is often avoided on low cost embedded systems for the following reasons: first it is usually difficult to predict the worst case memory usage, which can quickly lead to memory exhaustion on these systems; second, memory fragmentation is a serious problem for architectures without a memory management unit. In fact, on architectures with a memory management unit even if memory fragmentation happens in the virtual address space, it is always possible to defragment the physical memory, freeing large blocks of contiguous memory, in a transparent way for the application. This is not possible in the case of processors lacking a MMU because it would be necessary to keep track of all pointers and update them when the defragmentation process moves a contiguous memory block [2].

Usually on the AVR family of processors memory allocation is either performed statically i.e. global variables or when with dynamic allocation on the stack [3].

Nevertheless, if a heap is needed it is usually allocated within a fixed range of memory addresses for allocation. In such a case, the return stack can be made to start after the end of the heap, with risking overflows or memory waste.

[2] It is possible to use double pointers, as done in the Contiki operating system. However, all access must be preformed with double de-reference, if an intermediate pointer is kept by the application and defragmentation occurs the memory might be corrupted by accessing an invalid address

[3] Variable memory allocation on the stack is possible using as GNU gcc's non standard *alloca* function

Register Name	Description	Atmega103 Address	Atmega128 Address
SP_CF_L	Control Flow stack pointer Low	$00 ($20)	$46 ($66)
SP_CF_H	Control Flow stack pointer High	$01 ($21)	$47 ($67)
SSCR	Split Stack Control Register (sec 2.3.1.4)	$10 ($30)	$49 ($69)
CF_SS_L	Control Flow Stack Start Low	$02 ($22)	$55 ($75)
CF_SS_H	Control Flow Stack Start High	$03 ($23)	$56 ($76)

(a) New register allocation for the additional registers.

Register name Register name	Needs Locking	Locking condition	Unlocking condition	Authorized modifications
SP	No	N/A	N/A	Any
CF_SP	Partial	After First Write	Reboot	Internal to CF instructions
CF_SP_Start	Yes	After First Write	Reboot	None
SSCR	Yes	After First Write	Reboot	None

(b) New registers locking logic

Figure 2.4: Stack configurations and Control flow stack pointer description and additional locking logic

2.3 Implementation

2.3.1 Implementation

In order to validate our approach we implemented the changes to both a simulator and a soft core in a FPGA.

2.3.1.1 Implementation on a simulator

We modified the AVRORA [TLP05] simulator in order to simulate the modified core, this made possible simulate the complete platform composed of an Atmega128[ATM] MCU and a IEEE 802.15.4[Soc] radio device. The simulation involved running *unmodified* TinyOS applications, for wireless sensor networks. The changes to AVRORA simulator were implemented in about 200 lines of code out of the 50,000 lines of code that compose the AVRORA simulator (0.4% change).

2.3.1.2 Implementation on an FPGA

We implemented the modifications in a VHDL implementation of the Atmega103 core available at opencores.org. Although this micro-controller version is discontinued, it is very similar to the Atmega128 and the modifications implemented are probably very similar to those required for an Atmega128. The modifications were made with changes of 8% of the VHDL source code (500 lines out of 6000). The

resulting core was implemented on an Altera Cyclone II FPGA the overhead in number of logical elements used (LUT) is of 9% (2323 LUT for the original MCU and 2538 LUT for the modified MCU). Although, this overhead might appear significant it is a non optimized implementation and as there is no extra memory requirements for its implementation its overhead when implemented in an ASIC would probably be much lower.

2.3.1.3 Control flow modification operations

In the Atmel AVR core the program counter (PC) is not accessible as a general purpose register, instructions such as load and store cannot modify it. Therefore, there are only few instructions that can change the control flow, i.e. modifying the program counter or its saved value [4]. On the AVR the following instructions can modify the control flow :

- *Branch* and *jump* (JMP) instructions update the control flow. However, as the destination address is provided as an immediate constant value, they are not vulnerable to manipulation and no return address is stored on the stack.
- *Call* and *return* instructions use the control flow stack pointer to access the control flow stack. Those instructions will store or fetch the control flow instructions on the control flow stack.
- *Load* and *Store* instructions are prevented to alter the return stack, only access to data stack or other regions is allowed. The control flow stack and the data stack are checked to be consistent when a store is performed.
- *Calli* instruction takes a function pointer as parameter (from a register). This instruction is used for example in schedulers or object oriented code, in such a case an indirect call instruction is performed. If the attacker is able to modify the pointer (or register) before it is used by an indirect call instruction, she would be able to control one control flow change but not the following ones.
- Interrupts transfer the control flow to a fixed interrupt handler and the instruction that was executed while the interruption occurred is saved on the control flow stack, in our modified architecture the return address is therefore protected as well.

One difficulty with the implementation of IBMAC is that the stack pointer as well as the control flow stack pointer are 16 bit values and are modified with

[4]This is not the case in all embedded cores, for example ARM cores have the PC as a regular register, therefore many instructions are able to alter the control flow.

two instructions. Therefore, the update of the stack pointer is non atomic and its value can be temporally invalid. As a consequence it is not possible to enforce the constraints on stack pointers constantly. The solution we used is to enforce this constraint only when memory writes or reads are performed, with this approach the stack pointer can have a temporary invalid value when it is updated, without triggering an error.

2.3.1.4 Control flow stack configuration

The control flow stack needs to be configured before any control flow operation is used. It is activated from the "Split Stack configuration Register" (SSCR). In order to prevent the attacker from maliciously change this register configuration, it is made "writable once per boot": this configuration register is locked in hardware after the first write. The software (e.g. the boot loader) is therefore responsible for setting this register during the boot process. We use for this purpose the *init* sections provided in the default linker scripts, so that the configuration is made as early as possible.

2.3.1.5 Memory layout stack memory areas configuration

Compared to a traditional memory layout some configuration must be performed in order to enable the control flow stack and the memory access enforcement. For this purpose we implemented new configuration registers:

- *SSTACKEN* Split stack enable is a configuration bit which, when set, enables the split stack feature. It is part of the *SSCR* register.
- *CF_START* is a configuration register used to fix the start of the control flow stack. It is automatically initialized from the libc to the end of the statically allocated memory (data/bss) therefore requires no user configuration.
- *CF_SP* is the control flow stack pointer it is initialized with the same value as *CF_START* at boot and cannot be directly modified after initialisation.
- *CF_STACK_configured* is an internal flag in our modified core and it is automatically set after control flow registers have been set up. It cannot be modified by software and is reset when a reboot occurs. When this value is set any direct update of the *CF_START* and *CF_SP* registers are detected as possibly malicious modifications and therefore triggering a reboot.

These additional registers are described in Figure 2.4. In order to avoid conflict with existing peripherals devices or internal logic of the AVR cores the addresses of those configuration registers were chosen in the unused I/O registers addresses. The previously mentioned locking mechanisms that we implemented to prevent malicious manipulation of those registers are presented in Figure 2.4(b).

```
volatile uint16_t abssvar;
volatile uint32_t adatavar=10;

uint16_t factorial(uint16_t val){
    volatile local[10];
    if (val==1) return 1;
    else return val*myfact(val-1);
}

void factorial_with_smallalloc(){
    volatile uint8_t large[20];
    factorial(8);
}

void factorial_with_bigalloc(){
    volatile uint8_t large[200];
    factorial(8);
}

int main(){
    abssvar=10;
    factorial_with_smallalloc();
    factorial_with_bigalloc();
    return 0;
}
```

Figure 2.5: Example of a program that causes the stack to overflow

2.3.2 Evaluation

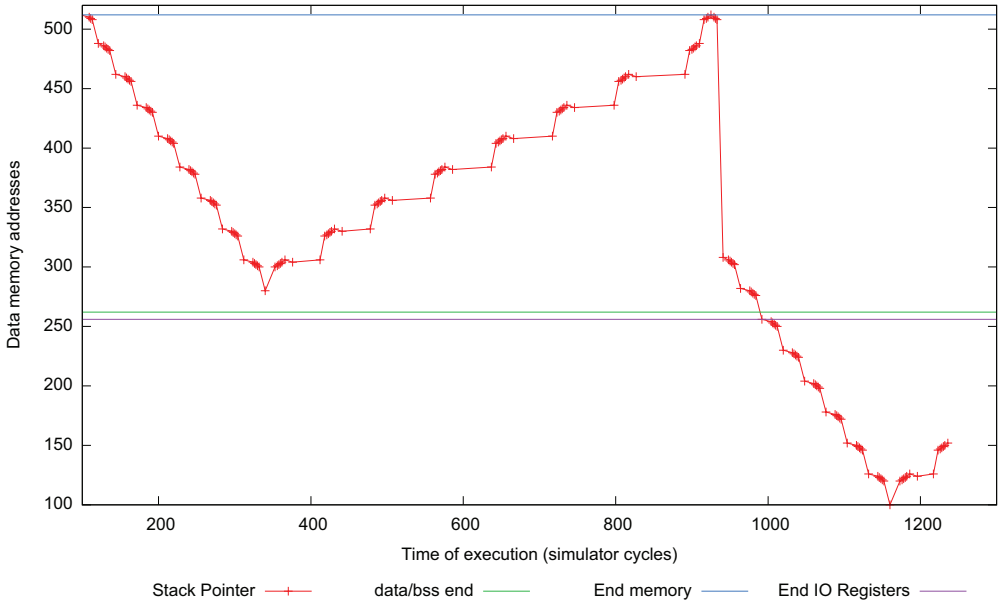
We evaluated the approach with different programs. Figure 2.5 shows an example program that has large stack memory usage. Two functions are present and are computing the factorial of a number with recursive calls. When the function with a larger array allocated on stack (*factorial_with_bigalloc*) is called a stack overflow occurs. Figure 2.6(a) shows the memory usage on an unmodified core, when the stack memory usage is too high the memory is corrupted and eventually unexpected behaviour occurs. On the other hand Figure 2.6(b) shows the resulting memory usage on an AVR core with split stacks and IBMAC. When the memory usage becomes too high the two stacks collide and the processor is rebooted by IBMAC. Similar results would be achieved if a malicious attempt to modify the control flow stack occurred.

2.4 Considerations

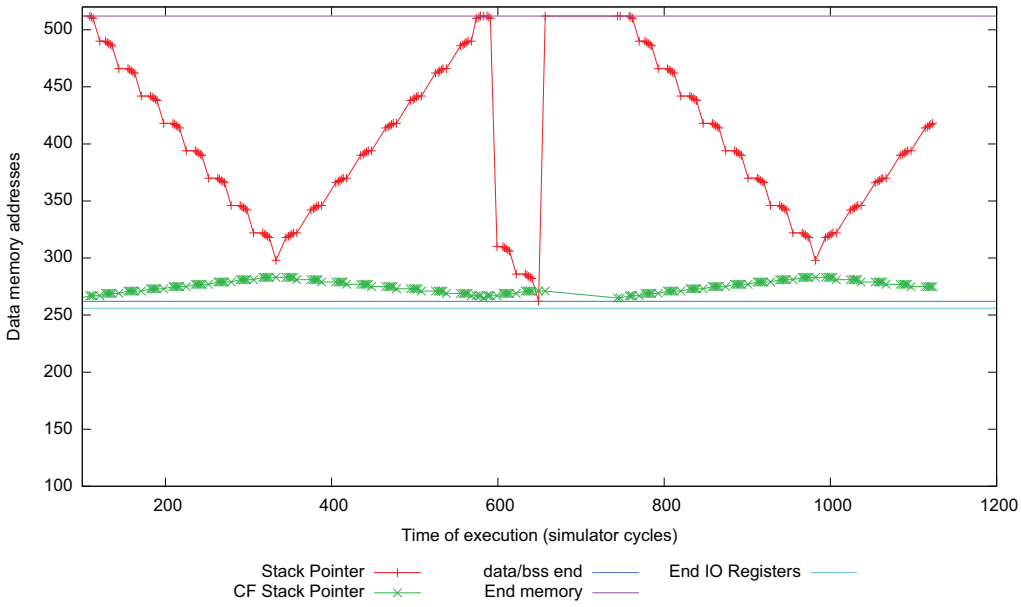
In addition to preventing stack based buffer overflows and stack overflows, IBMAC also prevents malicious software present in the MCU to use Return Oriented Programming. In a MCU without IBMAC an attacker can use *Return Oriented Programming* for malicious purposes, such as maliciously hiding code memory [HHF09a, CFPS09b]. In order to use Return Oriented Programming a malicious program needs to write a *stack* containing both data and return addresses. While an attacker can craft such a stack on normal MCU, IBMAC prevents this as the malicious code is not able to freely modify the return stack. Therefore, it is not possible to maliciously manipulate the control flow with return oriented programming, even tough arbitrary code can be run on the device. In order to prevent this behaviour, on a MCU where the attacker had full control, IBMAC needs to be permanently enabled. This can be performed using an irreversible configuration fuse. Without this the attacker would be able to restart the MCU on a modified program and deactivate the SSTACKEN configuration register.

Although our stack protection technique prevents control flow attacks as we described, it does not prevent all types of software or logical attacks. Mainly, non control data attacks [CXS+05b] are not addressed because they do not rely on a change of the control flow but on overwriting adjacent variables. For example, a buffer overflow could be used to change the value of a variable used as a flag in an *if* statement. This in turn could be used for example to bypass specific controls in the program code.

Regarding the backward compatibility, while most software can run without modifications, the split stack scheme can make the implementation of features such as tasks with context switching and longjump / setjump difficult. Those features requires the software to be able to modify the stack and its control flow.



(a) Execution *without* IBMAC. At point 1000 the stack is overflowing in the data/BSS section and later on the I/O register memory area.



(b) Execution *with* IBMAC enabled. When the return stack and the data stack collide (right after cycle 600), the execution of the program is aborted and restarted. This avoids memory corruption.

Figure 2.6: Comparison of the data memory layout during the execution of the program of figure 2.5. In order to keep the example simple we ran the simulation with only 512 bytes of data memory address space.

If a kernel execution (or execution rings) mode were available those features could be implemented safely. However, they cannot be implemented without major changes to the AVR core without the presence of such a privileged mode.

Chapter 3

Detecting Compromised Embedded Devices via Software Attestation

Contents

Preventing remote exploits does not provide a complete guarantee that an embedded device cannot be compromised. As explained, there are a number of possible vectors that can be used to inject malicious code like, for example, a faulty software update process or attaching the device to a debug interface. In these scenarios, one might still be interested in detecting the presence of malware on a remote embedded device, without physically accessing it. Detection could be possibly followed by the eradication of the infection.

The process of verifying the internal state of a remote device is called *attestation*. In this chapter, we first study the security of several previously proposed attestation schemes that rely purely on software support, therefore they are called software attestation schemes. We discover many of the previously proposed protocols to be vulnerable to a number of novel attacks we devise. Second, we devise an improved software-attestation scheme we call PoSE (Proof of Secure Erasure). PoSE bases its foundation on the lessons learned by attacking the previous attestation protocols and achieves provable security guarantees.

3.1 State of the Art

In this section we will introduce in more depth the concept of remote attestation and explain the current state of the art. Remote attestation is the process of secure verification of the internal state (code, data and configurations) of a remote hardware platform. Generally, it can be achieved either statically (at boot time) or dynamically, during normal operation in order to establish a dynamic root of trust. Attestation takes place between a trusted the *Verifier* and an untrusted *Prover* (the device we wish to ascertain the status of).

A naive approach for verifying the prover's memory content is for the verifier to challenge the prover to compute a message authentication code (MAC) of its memory contents. The verifier can generate a fresh and random key before each attestation and send it to the prover. The prover, in turn, can use the key to compute a MAC of the relevant portions of its memory and send the result back to the verifier. However, this approach is clearly not secure as the prover can easily cheat in the protocol. The prover in fact is not forced in any way to compute the MAC correctly over the requested portions of code. Even if the original code was altered, say by a remote exploit, the attacker can still save a copy of the original memory content. A malicious prover would rather deviate from this behavior and compute the MAC over the copy of the original saved copy of memory and pass attestation. In such a simple protocol the verifier would have no way to distinguish between a legitimate and a malicious prover.

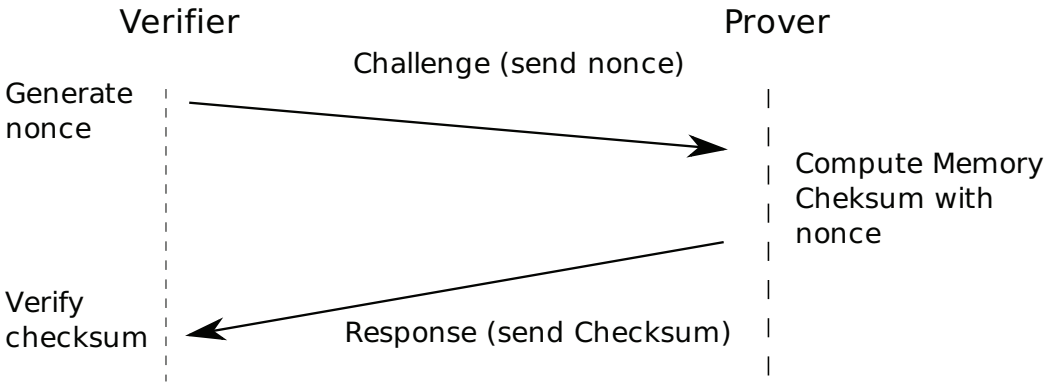


Figure 3.1: Basic attestation challenge response protocol

3.1.1 Software Attestation

Software attestation techniques try to solve the challenge explained in the past section using only software support, i.e., no special hardware support to perform attestation. Most software-based techniques rely on a challenge-response protocol that verifies the integrity of the code memory of a remote device: an attestation routine on the prover computes a checksum of its memory along with a challenge supplied by the verifier. In practice, memory words are read sequentially and fed into the attestation function. The challenge lies in making sure that a malicious prover cannot deviate from the protocol and pass attestation even in the presence of some malicious memory content.

Software attestation protocols can be divided in two broad categories: time based and memory based. In time based attestation protocols, the finite computational resources of the prover are used to estimate the expected time needed to correctly compute attestation of its memory. If the attestation routine is carefully designed, every deviation from the standard behavior is assumed to produce a measurable time delay on the prover, therefore allowing the verifier to detect malicious behavior. Memory based attestation protocols on the other hand utilize the fact that the prover only possesses a limited and known amount of accessible memory. During attestation, the free memory on the prover can be filled with random or pseudo-random values, so that the prover does not have any storage left to store malicious code.

One important and noteworthy assumption in almost all software attestation scheme is that the compromised prover device does not have any *real time* help. In particular, most (perhaps all) software techniques assume “adversarial silence”, meaning that, during every attestation process, only the intended prover (device being attested) is communicating with the verifier (entity that performs attestation). In other words, even though the prover might have malware installed, it is not aided – or impersonated – by any external party during attestation. The same assumption is sometimes referred to as “non collusion”. Any attestation technique

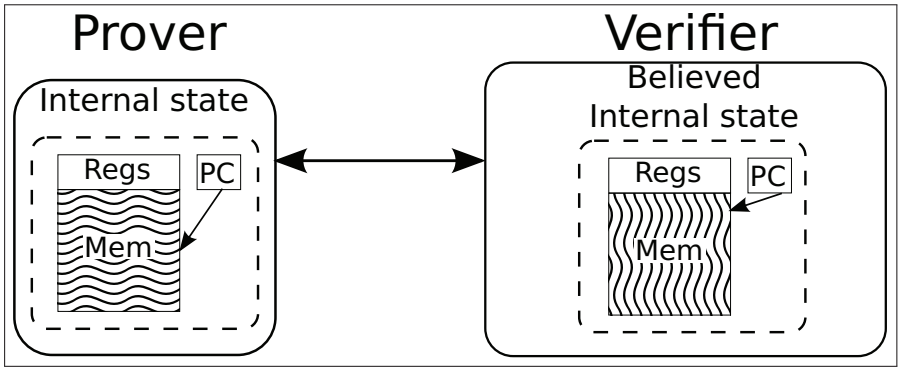


Figure 3.2: Generic remote attestation.

that makes this assumption is limited to close-range (one-hop) communication between the prover and the verifier and its security often relies on strict round-trip time measurements.

Also, software-based attestation assumes that, the adversary impersonating (or colluding with) the prover must use the same hardware as the genuine prover. While this assumption might hold in a few specific settings, it is unrealistic for many applications.

3.1.1.1 Time-based attestation

SWATT [SPvDK04] is an software attestation technique for embedded devices that relies on response timing to identify compromised code. The prover computes a checksum of its memory by traversing it using a pseudo-random sequence of indexes generated from a unique seed sent by the verifier. If a compromised prover wants to pass attestation, it has to redirect some memory accesses to compute a correct checksum. These redirections are assumed to induce a remotely measurable delay in the attestation that can be used by the verifier to decide whether to trust the prover's response. The same concept is used in [SLP+06b] where, the checksum calculation is extended to include also dynamic properties, e.g., the program counter or the status register. Furthermore the computation is optimised by having the checksum computed only on the attestation function itself.

Jakobsson, et al. [JJ10] proposed an attestation scheme to detect malware on mobile phones. This attestation scheme relies on both careful response timing and memory filling. Timing is used to measure attestation computation as well as external memory access and wireless links. Security of this approach depends on a number of hardware-specific details (e.g., flash memory access time). Hence, formal guarantees and portability to different platforms appear difficult to achieve.

3.1.1.2 Memory-based attestation

In [YWZC07] wireless sensors nodes collaborate to attest the integrity of their peers. At deployment time, each empty node's memory is filled with randomness, that is supposed to prevent malicious software from being stored, without deleting some parts of the original memory. A similar approach is taken in [CKN07], but, instead of relying on pre-deployed randomness, random values are generated using a PRF seeded by a challenge sent by the verifier and are used to fill the prover's memory. However, this does not assure compliance to the protocol of a malicious node that could trade computation for memory and still produce a valid checksum.

Gatzer et al. [GN07] suggest a method where random values are sent to a low-end embedded device (e.g., a SIM card) and then read back by the verifier, together with the attestation routine itself (called *Quine* in the paper). This construction, while quite valid, was only shown to be effective on an 8-bit Motorola MCU with an extremely simple instruction set.

3.1.1.3 Hybrid Remote Attestation Approaches

[SMKK05] proposed to use a distinct attestation routine for each attestation instance. The routine is transferred right before the protocol is run and uses self-modifying code and obfuscation techniques to prevent static analysis. Combined with timing of responses, this makes it difficult for the adversary to reverse-engineer the attestation routine fast enough to cheat the protocol and produce a valid (but forged) result. However, this approach relies on obfuscation techniques that are difficult to prove secure. Furthermore, some such techniques are difficult to implement on embedded systems that rely on the Harvard architecture, where code is stored in flash memory programmable only by pages.

In [SSW11] the authors propose a remote attestation scheme that makes use of Physically Unclonable Functions (PUFs). Physically unclonable functions are a class of noisy functions implemented in hardware and embedded in integrated circuits. The output of a PUF depends both on the input and on the intrinsic physical properties of the PUF itself, PUFs have the desirable property of being unclonable, hence they are well suited when authentication of the PUF-carrying device is needed. The proposal in [SSW11] makes use of a combination of the output of a PUF on the prover and an attestation routine derived from SWATT to achieve attestation with prover's authentication.

3.2 Attacks on Existing Software-based Attestation Schemes

Virtually all of the existing software-based attestation techniques previously proposed are based on a challenge-response paradigm where the verifier challenges a prover (a target device) to compute a checksum of its memory.

This section describes this basic challenge-response protocol and then presents how it is used by the existing software-based attestation schemes. We will present several attacks against various existing attestation protocols. But first we will start by introducing the adversarial model assumed by the attacked proposals.

3.2.1 Generic 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 3.1) 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 of memory accesses increases from n to $n\ln(n)$, where n is the total number of memory words (using the Coupon Collector Problem). 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 $\mathcal{H}(\cdot)$.

3.2.2 Adversary model

In most attestation proposals [CKN07, PS05, SLP08, SLP+06b, SLS+05, SPvDK04, SMKK05, YWZC07], 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. 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, it is assumed that at attestation time, while the malicious code is still running, the attacker has no direct control on the device anymore (what we called *adversarial silence assumption*). Finally, it is assumed 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. 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 not discussed in detail. Malicious code installation could be performed via remote exploitation of a software vulnerability as we explained above, a non invasive hardware attack [AK96] 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.

3.2.3 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.

3.2.3.1 A Rootkit-based attack

As discussed, recent work [Sha07, BRSS08, HHF09b] 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 [FC08]. 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[2]

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 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 regain control over the device.

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

[2] If the malicious code has complete control over the data memory, techniques such as memory safety [CAE+07] and stack canaries cannot prevent the usage of ROP.

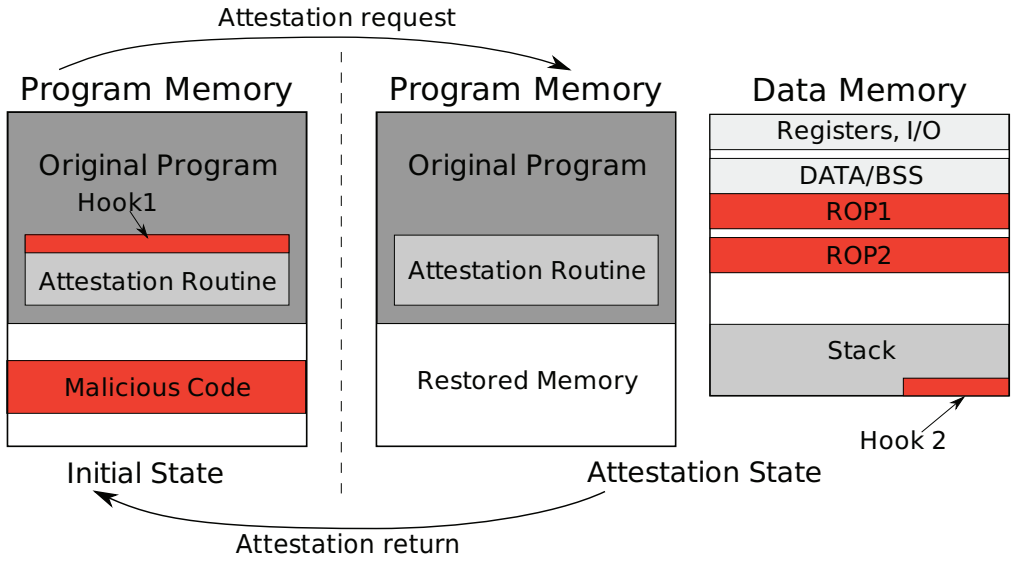


Figure 3.3: Return-Oriented Programming attack.

Figure 3.4 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.

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 3.3). 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 3.4: 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.

Attack description When an attestation request is received (*Initial State* in Figure 3.3), 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 [FC08], 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 3.3). 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:

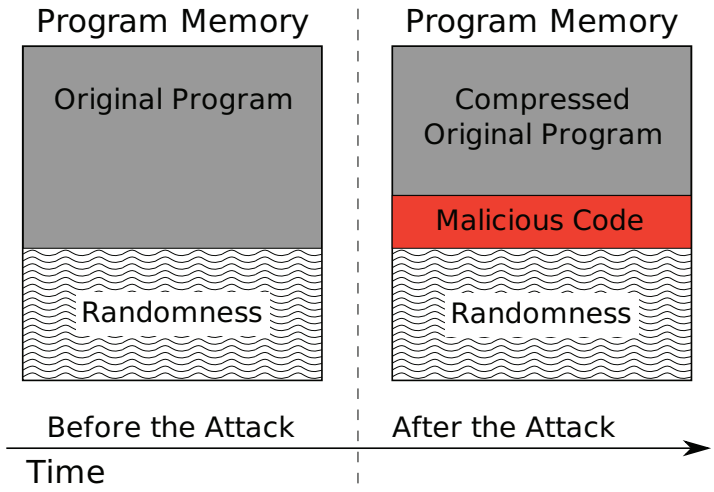


Figure 3.5: Compression Attack.

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*).

Experimental results The rootkit detailed above was implemented on a MicaZ device and used to attack the attestation protocol presented in [SPvDK04]. 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 3.7). Compared to the running time of SWATT, introduced overhead is about 0.3%.

Discussion In our implementation the attack relies on a single reprogramming meta-gadget that is composed of more than 100 instructions. Therefore, it does not require a Turing complete gadget set [3]. It uses instructions that manipulate

[3]Without using a Turing complete gadget set the technique we use could be refereed to as an hybrid between return-oriented programming and the borrowed code chunks [Kra05] techniques. Nevertheless, the availability of a Turing complete gadget set would probably make the attack easier to implement without changing its effectiveness or its results.

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.

3.2.3.2 Compression attack

Common sensor applications are appreciably smaller than the available program memory [4]. 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 [YWZC07] 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 3.5). 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 3.1). 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 [Huf62] 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

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

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

	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 3.2: Compression Attack, using Canonical Huffman encoding.

small gain for the attacker, it is sufficient to implement the attack we presented in Section 3.2.3.1.

Table 3.2 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 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 [YWZC07] do not rely on strict time bounding.

3.2.4 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 [SPvDK04] 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 [SLP+06b] and presents an attack that violates one of its security features.

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 3.6(a); as the address range (0xC000,0xFFFF) is not included in the checksum, the attacker could store the modified attestation code there.

Figure 3.6: Outline of the memory shadowing attack, with inserted instructions and translated addresses.

Method 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 3.7: 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.

3.2.4.1 SWATT

The security of SWATT [SPvDK04] 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.

3.2.4.2 A memory shadowing attack

The authors of [SPvDK04], 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 [5] and (2) they considered the fastest implementation of the attack. Such

[5]Speed requirements force implementation in assembly language, which is a very error-prone

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 [6], 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 3.6(b)). This can be implemented in an AVR micro-controller with two instructions (Figure 3.6(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.

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 3.7). 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 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[7]. 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.

---

process. For example, we found one bug in the original implementation of SWATT provided in [SPvDK04]: 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.

[6] This attack would therefore not be possible if the free program memory is used or filled with randomness (as in [CKN07, YWZC07]), but this is not the case with SWATT.

[7] 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.

3.2.4.3 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 thesis are the most recent Mica2 and MicaZ sensors that use an ATMega128L micro-controller [ATM] 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 [SPvDK04] as well as the memory shadowing one (Section 3.2.4.2) 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 3.7).

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.

3.2.4.4 Preventing the rootkit attack

In [SPvDK04] 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 3.2.3.1, 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 3.2.4.2), 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.

3.2.4.5 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 3.2.3.1, but not the shadowing attack showed in Section 3.2.4.2.

3.2.5 ICE-based attestation schemes

Indisputable Code Execution (ICE) based protocols (such as, SCUBA [SLP+06b], SAKE [SLP08] and Message-in-a-bottle [KLNP07]) are a class of protocols that use

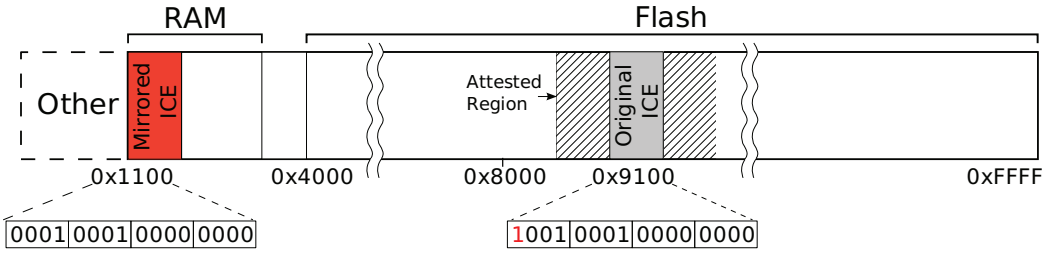


Figure 3.8: While the legitimate ICE routine is stored at address 0x9100, a malicious copy of the routine is stored at address 0x1100. These two addresses differ only in their most significant bit allowing the attacker to run the malicious copy of ICE and still pass attestation.

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 [KS04], 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:

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

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 work, ICE has been originally proposed for TelosB devices based on an MSP430 [Tex] 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 3.8. 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.

3.2.6 Considerations

We investigated the security of existing software-based device attestation protocols. Software based attestation on general purpose operating systems [KJ03] has been previously shown to have serious weaknesses [SCT04]. However, we are the first to present a security analysis of software based attestation schemes specifically designed for low-end embedded systems.

We discussed 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 to design correctly. Time-based attestation schemes must rely on very tight timing bounds. Their implementation must therefore be small, simple and time-optimized. Otherwise, a memory access redirection attack would not be detected as its overhead would be insignificant compared to the time spent by the checksum computation.

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. 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.

In the next section we will investigate the feasibility of a software-based attestation protocol that can guarantee security while being efficient and portable across different architectures.

3.3 Improving Software Attestation via Proof of Secure Erasure

As we showed in the last section designing software-based attestation protocols is a challenging task. The prover has little to no hardware support to guarantee the execution of the attestation routine. As a consequence, the attestation routine has to be designed to be resilient against malicious manipulation.

In this section we will describe PoSE (Proof of Secure Erasure), that performs attestation following a simple premise. If attestation is combined with software updates, then one can guarantee the absence of malware by overwriting the entire memory content of the prover with randomness generated by the verifier. The fact that the randomness is generated by the verifier at each protocol run and with an unknown seed to the prover, guarantees that the prover has to store the randomness entirely and therefore delete any malware present on its memories.

The advantages of our scheme are the following:

1. We suggest a simple, novel and practical approach to secure erasure, code update and attestation that falls between (secure, but costly) hardware-based and (efficient, but uncertain in terms of security) software-based techniques.

2. We show that the problem of secure remote code update can be addressed using Proofs of Secure Erasure (PoSE-s), i.e., a proof that a remote device is not storing anything except a set of random values newly selected by the verifier, which is equivalent to all prior state having been erased.

3. We propose several PoSE variants and analyze their security as well as efficiency features. We also assess their viability on a commodity sensor platform.

3.3.1 Assumptions

As explained, secure attestation involves a *verifier* $\mathcal{VRF}$ and a *prover* $\mathcal{PRV}$. Internal state of $\mathcal{PRV}$ is represented by a tuple $S = (M, RG, pc)$ where M denotes $\mathcal{PRV}$'s memory of size n (in bits), $RG = rg_1, ..., rg_m$ is the set of registers and pc is the program counter. We refer to S_P as the real internal state of the prover and S_V the internal state of the prover, as viewed by the verifier. Secure code update can be viewed as a means to ensure that $S_V = S_P$. Our notation is reflected in Table 3.3.

$\mathcal{PRV}$ is assumed to be a generic embedded device – e.g., a sensor, an actuator or a computer peripheral – with limited memory and other forms of storage. For the ease of exposition, we assume that all $\mathcal{PRV}$'s storage is homogeneous and contiguous. (This assumption can be easily relaxed, as discussed in section

Table 3.3: Notation Summary.

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

3.3.5.2) From here on, the term *"memory"* is used to denote all writable storage on the device. The verifier is a comparatively powerful computing device, e.g., a laptop-class machine.

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

We assume that the adversary cannot modify hardware configuration of $\mathcal{PRV}$[8], i.e., all anticipated attacks are software-based. The adversary has complete read/write access to $\mathcal{PRV}$'s memory, including all cryptographic keys and code. However, in order to achieve provable security, our protocol relies on the availability of a small amount of Read-Only Memory (ROM) that the adversary can read, but not modify. Finally, the adversary can perform both passive (such as eavesdropping) and active (such as replay) attacks. An attack succeeds if the compromised $\mathcal{PRV}$ device passes the attestation protocol despite presence of malicious code or data.

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

As in prior attestation literature, [CKN07, PS05, SLP08, SLP+06b, SLS+05, SPvDK04, SMKK05, YWZC07], we assume that the compromised prover device does not have any *real time* help. In other words, during attestation, it does

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

not communicate with any other party, except the verifier. Put another way, the adversary maintains complete radio silence during attestation. In all other respects, the adversary's power is assumed to be unlimited.

3.3.2 Design Rationale

Our design rationale is simple and based on three premises:

- **First**, we broaden our scope beyond attestation, to include both secure memory erasure and secure code update. In the event that the updated code is the same as the prior code, secure code update yields secure code attestation. We thus consider secure code update to be a more general primitive than attestation.

- **Second**, we consider two ways of obtaining secure code update: (1) download new code to the device and then perform code attestation, or, (2) securely erase everything on the device and then download new code. The former brings us right back to the problematic software-based attestation, while the latter translates into a simpler problem of secure memory erasure, followed by the download of the new code. We naturally choose the latter.

  Correctness of this approach is intuitive: since the prover's memory is strictly limited, its secure erasure implies that no prior data or code is resident; except for a small amount of code in ROM, which is immutable. Because the adversary is assumed to be passive during code update, download of new code always succeeds, barring any communication errors.

- **Third**, based on the above, we do not aim to *detect* the presence of any malicious code or extraneous data on the prover. Instead, our goal is to make sure that, after erasure or secure code update, no malicious code or extraneous data remains.

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

3.3.3 Secure Code Update

The base case for our secure code update approach is depicted in Figure 3.10. It is essentially a four-round protocol, where:

- Rounds one and two comprise secure erasure of all writable memory contents.

- Rounds three and four represent code update.

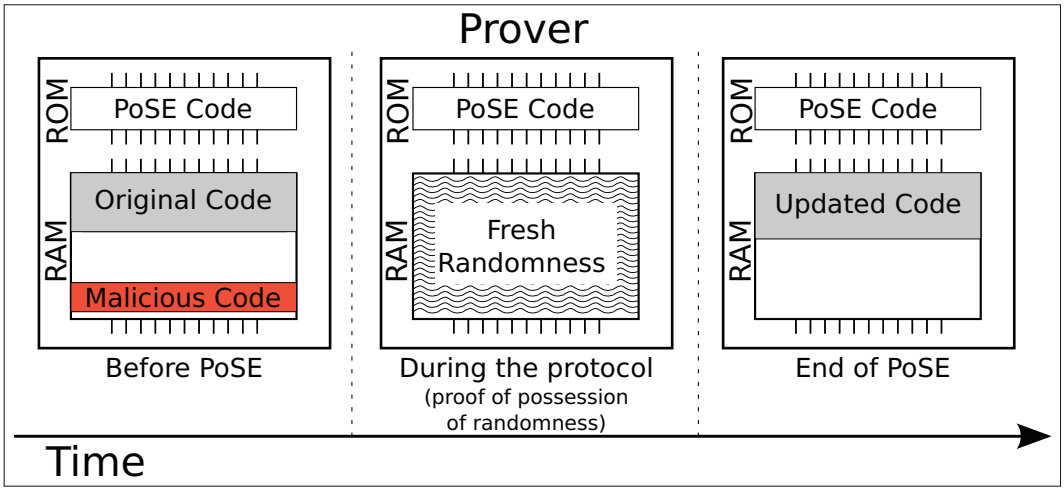


Figure 3.9: Prover's Memory during Protocol Execution

Note that there is absolutely no interleaving between any adjacent rounds. The "evolution" of prover's memory during the protocol is shown in Figure 3.9.

[1] $\mathcal{PRV} \longleftarrow \mathcal{VRF}$: $R_1, ..., R_n$

[2] $\mathcal{PRV} \longrightarrow \mathcal{VRF}$: $R_1, ..., R_n$

[3] $\mathcal{PRV} \longleftarrow \mathcal{VRF}$: $C_1, ..., C_n$

[4] $\mathcal{PRV} \longrightarrow \mathcal{VRF}$: ACK or $H(C_1, ..., C_n)$

Figure 3.10: Base Case Protocol

As mentioned earlier, we assume a small ROM unit on the prover. In the base case, ROM houses two functions: read-and-send and receive-and-write. During round one, receive-and-write is used to receive a random bit R_i and write it in location $M[i]$, for $0 \leq \ i < n$. At round two, read-and-send reads a bit from location $M[i]$ and sends it to the prover, for $0 \leq \ i < n$. (In practice, read and write operations involve *words* and not individual bits. However, this makes no difference in our description.)

If we assume that the $\mathcal{VRF} \leftrightarrow \mathcal{PRV}$ communication channel is lossless and error-free, it suffices for round four to be a simple acknowledgement. Otherwise, round four must be a checksum of the code downloaded in round three. In this case, the checksum routine must reside in ROM; denoted by $H()$ in round four of Figure 3.10. In the event of an error, the entire procedure is repeated.

3.3.4 Efficient Proof of Secure Erasure

As shown in Figure 3.10, secure erasure is achieved by filling prover's memory with verifier-selected randomness, followed by the prover returning the very same randomness to the verifier. On the prover, these two tasks are executed by the ROM-resident read-and-send and receive-and-write functions, respectively.

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

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

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

An alternative is for $\mathcal{PRV}$ to compute a MAC (Message Authentication Code) using the last k bits of randomness – received from $\mathcal{VRF}$ in round one – as the key. (Where k is sufficiently large, i.e., at least 128 bits.) A MAC function can be instantiated using constructs, such as AES CBC-based MAC [BKR94], AES CMAC or HMAC [BCK96] However, minimum code size varies, as discussed in Section 3.3.5. In this version of the protocol, the MAC function must be stored in ROM. Clearly, a function with the lowest memory utilization is preferable in order to minimize the amount of working memory that $\mathcal{PRV}$ needs to reserve for computing MAC-s.

Claim: Assuming a cryptographically strong source of randomness on $\mathcal{VRF}$ and a cryptographically strong MAC function, the following 2-round protocol achieves secure erasure of all writable memory M on $\mathcal{PRV}$:

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

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

Proof (Sketch): Suppose that malicious code MC occupies $b > 0$ bits and persists in M after completion of the secure code update protocol. Then, during

round one, either: (1) some MAC pre-computation was performed and certain bits (at least b) of $R_1, ..., R_{n-k}$ were not stored in M, or (2) the bit-string $R_1, ..., R_{n-k}$ was compressed into a smaller x-bit string ($x < n - k - b$). However, (1) is infeasible since the key K is only communicated to $\mathcal{PRV}$ at the very end of round one, which precludes any MAC pre-computation. Also, (2) is infeasible since $R_1, ..., R_{n-k}$ is originates from a cryptographically strong source of randomness and its entropy rules out any compression.

Despite its security and greatly reduced bandwidth overhead, this approach is still computationally costly considering that it requires a MAC to be computed over entire n-bit memory M. One way to alleviate its computational cost is by borrowing a technique from [ADPMT08] that is designed to obtain a probabilistic proof in a Provable Data Possession (PDP). The PDP scheme in [ADPMT08] assumes that data outsourced by $\mathcal{VRF}$ (client) to $\mathcal{PRV}$ (server) is partitioned into fixed-size m-bit blocks. $\mathcal{VRF}$ generates a sequence of t block indices and a one-time key K which are sent to $\mathcal{PRV}$. The latter is then asked to compute and return a MAC (using K) of the t index blocks. In fact, these t indices are not explicitly transferred to $\mathcal{PRV}$; instead, $\mathcal{VRF}$ supplies a random seed from which $\mathcal{PRV}$ (e.g., using a hash function or a PRF) generates a sequence of indices.

As shown in [ADPMT08], this technique achieves detection probability of: $\mathcal{P} = 1 - (1 - \frac{m}{d})^t$ where m is the number of blocks that $\mathcal{VRF}$ **did not** store (i.e., blocks where malicious code resides), d is the total number of blocks and t is the number of blocks being checked.

Consider a concrete example of a Mica Mote with 128 Kbytes of processor RAM and further 512 Kbytes of data memory, totaling 640 Kbytes. Suppose that block size is 128 bytes and there are thus $5,120$ blocks. If $\frac{m}{d} = 1\%$, i.e., $m = 51$ blocks, with $t = 512$, detection probability amounts to about 99.94%. This represents an acceptable trade-off for applications where the advantage of MAC-ing $\frac{1}{10}$-th of verifier memory outweighs the 0.06% chance of residual malicious code and/or data. Figure 3.11 plots the probability t for different values of m.

3.3.4.1 Optimizing Code Update

Recall that, in the base case of Figure 3.10, round three corresponds to code update. Although, in practice, code size is likely to be less than n, receiving and storing entire code is a costly step. This motivates the need for shortcuts. Fortunately, there is one effective and obvious shortcut. The main idea is to replace a random $(n - k)$-bit string with the same-length encryption of new code under some key K'. This way, after round two (whether as in the base case or optimized as in the previous section), $\mathcal{VRF}$ sends K' to $\mathcal{PRV}$ which uses K' to decrypt the code. The resulting protocol is shown in Figure 3.12.

Note again that, since we assume no communication interference and no packet loss or communication errors, the last round is just an acknowledgement, i.e., not a function of decrypted code or K'. This optimization does not affect the

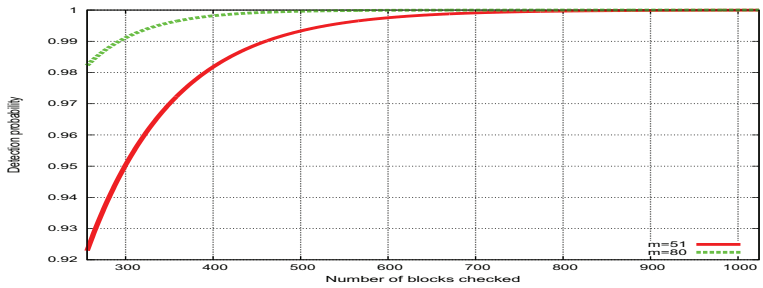


Figure 3.11: Probability of detecting memory modifications for # of checked blocks varying between 256 (5%) and 1024 (20%)

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

Figure 3.12: Optimized Protocol

security of our scheme if a secure block cipher is used, since encryption of code $[C_1, ..., C_{n-k}]$ with key K' is random and unpredictable to the prover before key K' is disclosed. Hence, the proof in Section 3.3.4 also holds for this optimized version of the protocol.

3.3.5 Implementation and Performance

In order to estimate its performance and power requirements, we implemented PoSE on the ATMEGA128 micro-controller mounted on a MicaZ sensor. Characteristics of this sensor [Cro] platform relevant to our scheme are: 648KB total programmable memory; 250kbps data rate for the wireless communication channel. The total memory is divided into: 128KB of internal flash; 4KB of internal SRAM; 4KB of configuration EEPROM; 512KB of external flash memory. The application was implemented on TinyOS.

3.3.5.1 Performance Evaluation

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

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

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

MAC Computation We evaluated the performance of three different MAC constructs: HMAC-MD5, HMAC-SHA1 and SkipJack in CBC-MAC. Note that, even though there are well-known attacks on MD5 that find chosen-prefix collisions [SLW07], the short-lived nature of the integrity check needed in our protocol rules out attacks that require 2^{50} calls to the underlying compression function. Table

Table 3.4: MAC constructions on MicaZ.

(a) Energy consumption and time

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

(b) Code and working memory required

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

3.4(a) shows the results: in each case we timed MAC computation over 644KB of memory on MicaZ.

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

3.3.5.2 Memory Usage

We now attempt to estimate the amounts of code and volatile memory needed to run PoSe. An estimate of code memory needed to run it is necessary to understand ROM size requirements. Furthermore, estimating required volatile memory is critical for the security of the protocol. In fact, in order to correctly follow the protocol, the prover needs a minimal amount of working memory. This memory can not be filled with randomness and hence $\mathcal{PRV}$ could use it to store arbitrary values. However, by keeping the amount of volatile memory to a *minimum* we can guarantee that $\mathcal{PRV}$ can not store both arbitrary values and carry on the necessary computation to complete the protocol.

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

Code Size To estimate code size, we implemented the base case PoSE protocol in TinyOS. It transmits and receives over the wireless channel using Active Messages. The entire application takes $11,314$ bytes of code memory and 200 bytes of RAM. RAM is needed to hold the necessary data structures along with the

Table 3.5: Code and volatile memory size.

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

stack. Our implementation used regular TinyOS libraries and compiler. Careful optimization would most likely reduce memory consumption.

In the optimized version of PoSE, we also need a MAC housed in ROM. Table 3.4(b) shows the amount of additional memory necessary to store code and data for various MAC constructions. Finally, Table 3.5 shows the size of both code and working memory for all presented above.

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

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

3.3.5.3 Read-Only Memory

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

However, there are other means to obtain read-only memory using different and widely available technologies. For example, ATMEGA128 [ATM] allows a

portion of its flash memory to be *locked* in order to prevent overwriting. Even though the size of this lockable portion of memory is limited to 4KB, this feature shows the feasibility of such an approach on current embedded devices. Note that, once locked, the memory portion cannot be unlocked unless an external JTAG debugger is attached to unset the lock bit.

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

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

3.3.6 Limitations and Challenges

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

In terms of provable security, our discussion of Proofs-of-Secure-Erasure (PoSE-s) has been rather light-weight. A more formal treatment of the PoSE primitive needs to be undertaken.

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

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

Jamming can be used to selectively allow the prover to complete the protocol, while preventing it from communicating with any other party. Any attempt to circumvent jamming by increasing transmission power can be limited by using readily available hardware. For example, the CC2420 radio, present on the MicaZ, supports transmission power control. Thresholds can be set for the Received Signal Strength (RSS), RSS_{min} and RSS_{max}, such that only frames

with $RSS \in [RSS_{min}, RSS_{max}]$ are accepted and processed. This is enforced in hardware by the radio chip. Hence, if the verifier wants to make sure that the prover does not communicate, it can simply emit a signal with $RSS > RSS_{max}$. This approach is similar to the one employed in [MPS09], albeit, in a different setting.

3.3.7 Considerations

We considered secure erasure, secure code update and remote attestation in the context of embedded devices. Having examined prior software attestation approaches, we concluded that they offer uncertain guarantees. We explored an alternative approach that generalized the attestation problem to remote code update and secure erasure. Our approach, based on Proofs-of-Secure-Erasure relies neither on secure hardware nor on tight timing constraints. Moreover, although not particularly efficient, it is viable, secure and offers some promise for the future. We also assess the feasibility of the proposed method in the context of commodity sensors.

Chapter 4

Secure Attestation and Dynamic Root of Trust with Minimal Hardware Support

Contents

As explained above, software based attestation techniques aim at providing attestation without any hardware support, this, however, comes at the cost of reduced applicability and security of the those solutions. Hardware based attestation techniques aim at solving the major drawback of software attestation, namely, the strong adversarial silence assumption. This assumption is needed because, when all the memories of the prover can be read and modified by the attacker, no secret keys are left to authenticate the verifier to the prover and vice versa. This limitation underlies any attestation technique that relies purely on software. Hence, it makes sense to design an attestation techniques that uses some form of hardware support to address this limitation. The goal of SMART (Secure and Minimal Architecture for a dynamic Root of Trust) is to provide a *minimal* set of hardware modifications to establish a dynamic root of trust on embedded devices.

Furthermore, typical software attestation protocols require the prover to compute a checksum of its entire memory, this process can take several seconds on low end embedded devices. This extended computation time poses several problems in the context of embedded devices used in cyber physical systems. First, it consumes considerable amounts of memory by requiring intensive computation by a device that would normally be idly responding to periodic events. Second, it requires the entire platform to be uniquely used for attestation, i.e., no other application can run concurrently. All these issues pose severe constraints to the applicability of the proposed software attestation protocols in real world application. We therefore turn our attention to hardware/software co-designs that can achieve the same (in fact more) security guarantees while requiring a fraction of the time.

We start by explaining the hardware attestation techniques proposed in the context of commodity high end devices. Such techniques, are usually based on a secure co-processor that provides a root of trust to base attestation upon. Techniques based on a secure co-processor, however, are ill suited to embedded devices whose computational capabilities are in the same range as the secure co-processor itself. The co-processor would therefore add a considerable overhead to the platform cost. Therefore, we design a solution that relies on a minimal (albeit secure) hardware modification to existing embedded devices to provide attestation and dynamic root of trust on embedded devices. We will describe the concept of a dynamic root of trust in detail in the following section. The solution has been implemented and tested on the MSP430 and the AVR family of micro-controllers.

4.1 State of the Art

4.1.1 Hardware attestation

Many hardware-based attestation have been proposed in the context of commodity devices, like desktop grade or mobile devices. We will survey the state of the art in hardware attestation techniques to better introduce our solution to hardware attestation on embedded devices.

4.1.1.1 Static Integrity Measures

Secure boot [AFS97] was proposed to ensure a chain of trusted integrity checks, beginning at power-on with the BIOS and continuing until the kernel is loaded. These integrity checks compare the computation of a cryptographic hash function with a signed value associated with the checked component. If one of the checks fails, the system is rebooted and brought back to a known saved state.

4.1.1.2 Attestation using Trusted Platform Modules

Trusted Platform Module (TPM) is a secure coprocessor that stores a *integrity measures* of a system according to the specifications of the Trusted Computing Group (TCG) [Trua]. Upon boot, the control is passed to an immutable code base that computes a cryptographic hash of the BIOS, hash that is then securely stored in the TPM. Later, control is passed to the BIOS and the same procedure is applied recursively until the kernel is loaded. In contrast to secure boot, this approach does not detect integrity violations, instead the task is left to a remote verifier. The integrity measures are implemented as cryptographic hashes of the content of the attested memory regions, e.g., code components. The integrity measures are stored in special registers inside the TPM called Platform Configuration Registers (PCR). The TPM has 16 or 24 of such registers (depending on the version) that are zeroed only at boot time. Starting from the BIOS the values of the PCRs are *extended* by hashing their previous value together with the hash of the software component that is being *measured*. In detail this is how the PCR value is updated: $PCR_{new} = H(PCR_{old}||H(code))$.

Attestation is implemented using the TPM by extending the measurements at boot of all the desired software components (BIOS, master boot record, OS Kernel, etc.) and a *quote* of the PCR measurement to the TPM. The quote is a digital signature of the contents of the attestation PCR and a nonce sent by the verifier to assure freshness. The TPM uses the private part of the Attestation Identity Key (AIK) to generate the signature. The verifier can use the public part of the AIK together with the signature sent by the prover to verify the internal software state of the prover.

An experimental embedded device that includes a full TPM as a separate chip (together with a low-end microcontroller) has been proposed [HTC+10]. However,

this approach is expensive for low-end devices, as the cost of a TPM chip is close to that of a low-end microcontroller itself. Indeed, a typical TPM includes a microcontroller, memory and cryptographic accelerators [ATM05]. Thus, adding a TPM to a device doubles the number of actual CPUs. Finally, increase in number of components also impacts devices cost and size.

4.1.1.3 Dynamic Integrity Measures

In [KSA+09], the use of TPM is extended to provide system integrity checks of run-time properties with ReDAS (**Re**mote **D**ynamic **A**ttestation **S**ystem). At every system call, a kernel module checks the integrity of constant properties of dynamic objects, e.g., invariant relations between the saved frame pointer and the caller's stack frame. Upon detection of an integrity violation, the kernel driver seals the information about the violation in the TPM. A remote verifier can ask the prover to send the sealed integrity measures and thus verify that no integrity violations occurred. However ReDAS only checks for violations of a subset of the invariant system properties and nothing prevents an adversary to succeed in subverting a system without modifying the properties checked by ReDAS. Extending the set of attested properties is difficult due to the increased number of false positives generated by this approach, for example in case of dynamic properties classified as invariants by mistake. [SZJvD04] proposed to extend the functionality of the TPM to maintain a chain of trust up to the application layer and system configuration. In order to do so, they extend the Linux kernel to include a new system call that measures files and adds the checksum in a list stored by the kernel. The integrity of this list is then sealed in the TPM.

4.1.2 Dynamic Root of Trust

Recently the TPM specifications [Trub] have been extended to provide a way to perform attestation *dynamically* after boot. This is accomplished by allowing a specific CPU instruction to reset the state of some PCRs (PCR 17 to 24). From that moment on, new measurements can be extended into these PCRs to establish a dynamic root of trust and execute a piece of code in full isolation.

These mechanism have been recently implemented by AMD SVM [Adv] and Intel TXT [Int09] in their respective CPU to perform a *late launch*. Two new instructions, $SKINIT$ on AMD and $SENTER$ on Intel, are invoked to establish a dynamic root of trust. Several hardware protections measures, such as disabling debugging and resetting the TPM PCRs, are included to prevent fraudulent attestation.

McCune, et al. proposed the Flicker system architecture [MPP+08] to establish a dynamic root of trust on commodity computers. Leveraging AMD and Intel advances, Flicker can establish a dynamic root of trust by running a PAL (Piece of Application Logic) on the prover. The execution of the PAL is guaranteed even

if the BIOS, OS and DMA of the system are all compromised. This was further extended into TrustVisor [MLQ+10] which provides a dynamic root of trust for PALs directly from a minimal hypervisor. This allows to significantly improve the performance of the Dynamic Root of Trust mechanism as it can be handled by the hypervisor avoiding to use the relatively slow $SKINIT/SENTER$ process. Trustvisor also allows to use PALs on legacy Operating systems as a hypercall can be done directly from an application without support from the operating system.

4.2 Design Elements and Goals

As mentioned above, the main result discussed in this chapter is the development of a new technique called SMART: Secure and Minimal Architecture for a dynamic Root of Trust. SMART is executed by the prover and, in doing so, attests a region of code and executes it. At the end of the execution, the prover is able to provide proof to the verifier that attests the execution, including the input and output parameters, therefore establishing a dynamic root of trust. SMART guarantees that the attested code is executed even if the entire prover system is compromised.

SMART was designed with one main goal: *minimality* and *efficiency*. Low-end embedded devices are very cost sensitive and the addition of secure co-processors is infeasible. Even extending the core functionality of the MCU must be done to minimize the number of gate equivalences added to the original design. The efficiency of the protocol is important because embedded devices often have to operate in real-time and with rigid time constraints.

4.3 Overview of the Solution

4.3.1 Building Blocks

The main idea behind SMART is to identify a minimal combination of additional features to add to a micro-controller to support attestation and the establishment of a dynamic root of trust. We begin by considering and justifying various components that allow us to assemble a secure attestation primitive. (Note that we are focusing on the prover, i.e., the device being attested.) Our approach relies on three main components:

Attestation Read-Only Memory: Region of memory in ROM inside the MCU. Only this code segment is granted access to a key $\mathcal{K}$, by the MCU. We will sometimes refer to this code as SMART code.

Secure Key Storage: Memory region inside the CPU; this region can be accessed only from SMART code in ROM.

MCU Access Controls: Control access to $\mathcal{K}$ and prevent non-SMART code from accessing it.

4.3.1.1 Attestation Read-Only Memory (ROM)

It is used as the root of trust to faithfully compute an message authentication code (MAC) of the target memory region [1] Generally, ROM incurs very little overhead in the design and construction of the MCU since it constitutes a cheap form of storage. What makes this ROM special is its ability to access $\mathcal{K}$, that is used only for attestation. This rule is enforced in hardware and is clarified in subsequent sections. The SMART code resides in read-only memory and therefore cannot be tampered with by any malicious code. This implicit integrity guarantee is one of the foundations of our technique.

While the SMART code is the only one that can access the attestation key $\mathcal{K}$ there are certain corner cases that have to be properly handled to preserve the secrecy of $\mathcal{K}$. In fact, part the entire key $\mathcal{K}$ or parts of it necessarily have to be handled by SMART code and copied in register or temporarily stored in RAM. Therefore it is important that, before the SMART code terminates its execution, all the data structure derived from $\mathcal{K}$ are deleted.

To enforce this, in SMART, a memory cleanup is performed by immutable code in ROM that is executed upon every boot or reset. Also, special care must be taken to avoid that the key is accidentally leaked by SMART for example, as a result of buggy SMART code.

This phenomenon is conceptually similar to cold boot attacks [HSH+08] where a computer is stopped during execution and its memory is removed in order to recover keying material. However, since a typical microcontroller features processor and memory in a single "package", the latter cannot be accessed directly. If debugging interfaces are permanently deactivated and memory is freed upon each reset, only hardware attacks (that are out of scope of SMART) would allow recovery of parts of memory.

4.3.1.2 Secure Key Storage

The next question is where to store the key used for computing the keyed hash function (HMAC). We note that any un-keyed function, such a cryptographic hash (e.g., SHA), would be unsuitable for attestation. This is because, without a secret key, anyone can compute a hash of any input and impersonate the prover's reply. In particular, malware that has infected the prover can do so.

Therefore, secret storage is needed. In SMART, we use it to house a single (symmetric or private) key. $\mathcal{K}$ is kept in a secure key storage supported in hardware. This storage must be immune to software attacks. However, we do not require

[1] In our implementation we use a hash-based message authentication code (HMAC) that uses SHA-1.

it to be secure against hardware (physical) attacks, since such attacks are out of scope of this work (i.e., not part of our adversary model). Furthermore, hardware attacks could be mitigated using well-known tamper-resistance techniques, such as anomaly detection, internal power regulators, additional metal layers or meshes for tamper detection.

Although we do not consider the details of initializing the attestation key, we note that there are at least two ways to do it: (1) either $\mathcal{K}$ is assigned at production time and never changed again, or (2) a secure way to *modify*, but not read, $\mathcal{K}$ is provided, e.g., by relying on a secure debug channel.

If a public-private key pair is used, the public key can be signed by a certification authority that can certify that the SMART module was built according to specifications.

4.3.1.3 MCU Access Controls

To enforce secrecy of $\mathcal{K}$, it can be accessed only when the program counter (PC) is in the memory region of Attestation ROM. This prevents software running in other memory regions from accessing $\mathcal{K}$ directly (See Section 4.4.2.2). However, additional memory access checks are necessary in the MCU to prevent code reuse attacks that would invoke code within SMART region to obtain the key, e.g., via ROP attacks [Sha07, BRSS08, CDD+10] (See Section 4.4.2.3).

4.3.2 Adversarial Assumptions

We assume that verifier, $\mathcal{VRF}$, and prover, $\mathcal{PRV}$, share a secret key $\mathcal{K}$. The adversary, $\mathcal{ADV}$, has complete control over the software state, code and data, of $\mathcal{PRV}$. In particular, $\mathcal{ADV}$ can modify any writable code on $\mathcal{PRV}$ and learn any secret that is not protected by the MCU. Furthermore, $\mathcal{ADV}$ has complete control over the communication channel and – during the protocol – can use multiple colluding devices in order to pass or subvert attestation.

We also assume that $\mathcal{ADV}$ does not perform hardware attacks on $\mathcal{PRV}$. Specifically, it can not alter code stored in ROM or retrieve $\mathcal{K}$ using side-channel attacks. Likewise, $\mathcal{ADV}$ has no means of interrupting execution of ROM-resident code on $\mathcal{PRV}$. A software only timing side-channel attack against the HMAC-SHA function used in SMART is not possible either. Code used for the HMAC computation does not have conditional branching instructions, resulting in constant execution time. Furthermore, MSP430 nor the AVR possess caches, that could be used for timing attacks due to cache hits and misses. Also, differences in execution time due to bus contention are data independent and cannot leak $\mathcal{K}$. Also, to the best of our knowledge no timing attacks have been reported against HMAC-SHA in the literature. Protection against hardware-based attacks could be added by, for example, encasing the MCU in tamper-resistant coating and employing standard

techniques to prevent side-channel key leakage. (Since our approach is confined within the MCU, employing such techniques is quite natural.)

4.3.3 Security Objectives

- *Prover Authentication:* Upon successful completion of the attestation protocol, $\mathcal{VRF}$ obtains entity authentication of $\mathcal{PRV}$.

- *External Verification:* Upon successful completion of the attestation protocol, $\mathcal{VRF}$ is assured that memory segment $[a, b]$ on $\mathcal{PRV}$ has the expected content.

- *Guaranteed Execution:* Upon successful completion of the attestation protocol, $\mathcal{VRF}$ is assured that code at location a was executed by $\mathcal{PRV}$.

4.4 SMART in Detail

This section describes, in detail, SMART features and components.

As mentioned earlier, SMART is executed by $\mathcal{PRV}$It attests a segment of memory and optionally jumps to it. SMART guarantees that this code is executed without modifications, even if the entire prover system is compromised. At the start of the protocol, $\mathcal{VRF}$ sends several parameters to $\mathcal{PRV}$: attestation region boundaries: a and b; address $x \in [a, b]$ where $\mathcal{PRV}$ passes control to after attestation; and a nonce r to prevent replay attacks. A code segment on $\mathcal{PRV}$ residing in ROM computes a cryptographic checksum C of a region $[a, b]$ in $\mathcal{PRV}$'s memory (using nonce r) and then passes control to x. After execution of code starting at x, $\mathcal{PRV}$ sends C to $\mathcal{VRF}$. The latter verifies correctness of C by re-computing it using the same parameters and attestation key $\mathcal{K}$. We refer to the code in ROM as $\mathcal{RC}$ and the code optionally executed after completion as $\mathcal{HC}$. A protocol view of SMART is shown in Figure 4.1 and the corresponding pseudo-code $\mathcal{RC}$ is illustrated in Algorithm 1.

The cryptographic checksum function is implemented as an HMAC keyed with $\mathcal{K}$ stored in secure storage in $\mathcal{PRV}$ MCU. Usage of, and access to, $\mathcal{K}$ is restricted by the MCU such that only (trusted and immutable) $\mathcal{RC}$ is allowed to use it. For its part, $\mathcal{RC}$ only uses $\mathcal{K}$ to compute HMAC and then passes control to $\mathcal{HC}$. Furthermore, $\mathcal{RC}$ is instrumented using both static and dynamic analysis tools to prevent accidental leakage of $\mathcal{K}$ or other cryptographic material. During this process, all interrupts are masked such that execution cannot be interrupted.

Additionally, when the x_{flag} is set interrupts remain masked after the execution of $\mathcal{RC}$. This is to ensure that $\mathcal{HC}$ is actually executed. This could be prevented by an attacker that installs a malicious interrupt handler and prepares an interrupt (e.g., a timer) to happen during the first instructions of $\mathcal{HC}$. Such an interrupt

Algorithm 1: SMART code in ROM.

input : a, b start/end addresses for attestation
x address to jump to after attestation
x_{flag} jump or not?
r nonce sent by verifier
out output address where to store checksum
in (optional) input parameter

output: output of HMAC

begin
// Attestation key is unlocked automatically by the MCU
// read $\mathcal{K}$ into k_a
$k_a \longleftarrow$ `loadAttestKey()`;
// Initialize HMAC with k_a
`initHmac(`k_a`)`;
// Attest all parameters
`HmacProcess(`$a||b||x||x_{flag}||r||in||out$`)`;
for $i \in [a, b]$ **do**
// Attest memory region $[a, b]$
`HmacProcess(`$Mem[i]$`)`;
end
$C \longleftarrow$ `finishHmac()`;
`copy(`$*out, C$`)`;
// Erase cryptographic material
$k_a \longleftarrow 0$;
`resetMemory()`;
// If execute flag set, go to address x
if $x_{flag} = True$ **then**
`call(`x, in`)`;
end
end

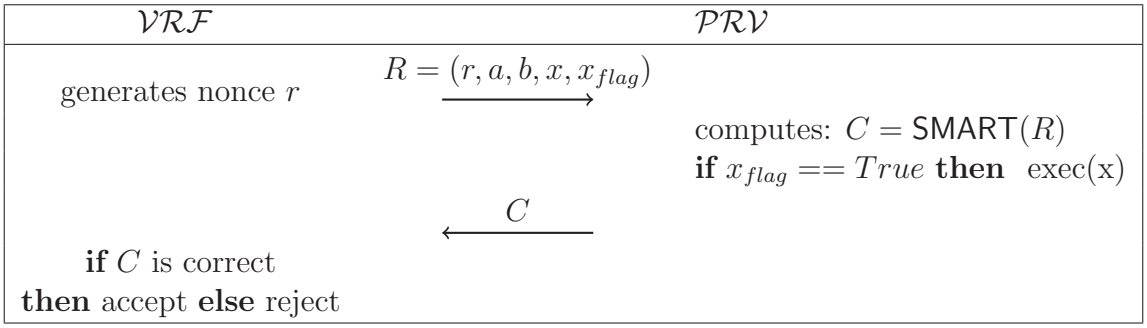


Figure 4.1: Protocol description

handler could allow for example to read or modify memory between the execution of $\mathcal{RC}$ and $\mathcal{HC}$.

4.4.1 Attestation ROM

As shown in Algorithm 1, SMART computes an HMAC of a particular memory segment and then jumps – without ever being interrupted – to a verifier-specified address within that segment. All code is written in C and consists of approximately 500 lines of code. This makes checking its correctness both feasible and relatively easy. Furthermore, our implementation experience shows that ROM code remains largely unchanged for various MCU platforms, such AVR and TI-MSP430. Thus, porting it to other commodity MCUs should be a fairly simple task.

ROM code must guarantee the following properties:

1. *Key Isolation:* $\mathcal{K}$ must not be leaked from ROM during normal operation.

2. *Memory Safety:* no bugs exploitable for temporary memory exposure or key leakage.

3. *Atomic Execution:* can be executed only from the very beginning and must not be interruptable until it terminates.

Property (3) is guaranteed by the MCU, as discussed in Section 4.4.2 below. Property (1) and (2) are guaranteed by using two code instrumentation tools: `CQUAL` [FTA02] and `Deputy` [CHA+07].

4.4.1.1 Key Isolation

Upon termination, SMART passes control to the untrusted portion of $\mathcal{PRV}$, where malicious code can sift through memory and search for traces of the attestation key or intermediate states used in HMAC computation. This could lead to disclosure of $\mathcal{K}$. For this reason, we instrumented SMART code with `CQUAL` – a tool that detects information leakage in C programs. Specifically, $\mathcal{K}$ is marked with a

SECRET type. CQUAL propagates this type to each variable that is computed with any involvement of any other variable of type SECRET. Each function is equipped with a check that no SECRET variable was leaked. Using CQUAL provides insights on preventing key leakage. The end-result is simple: each variable marked as SECRET by CQUAL is zeroed out in the epilogue of each function [2]. The only variables not erased are the outputs of each function. Also, upon SMART completion, the memory location of the attestation key ($\mathcal{K}$) is no longer accessible.

4.4.1.2 Memory Safety

Key isolation alone does not prevent key leakage, since our code could contain vulnerabilities that permit $\mathcal{ADV}$ to retrieve $\mathcal{K}$ by running SMART on malicious (or malformed) inputs. Fortunately, SMART is comprised of only around 500 lines of code. This relatively small size allows manual inspection for memory corruption bugs. However, we enhance manual inspection using Deputy – a C compiler built upon GCC, that provides an annotation language to describe memory boundaries in C. For example, a C array can be augmented with information about its size. The compiler adds instructions to check all memory accesses to the array and detects memory corruptions. Once SMART code is reinforced with Deputy, whenever a memory corruption is detected, a special reset is performed by Deputy instrumentation code. As for other error conditions that could cause a reset, we deal with them by making sure that, at each reset, all memory (stack, heap, and registers) is erased. Figure 4.2 shows a SMART code snippet with annotation.

```
hmac_sha1_nonce
(uint8_t * COUNT(klen) k, size_t klen,
uint8_t * COUNT(noncelen) nonce, size_t noncelen,
void * COUNT(exec_len) exec, size_t exec_len,
uint8_t exec_flag,
uint8_t *  in, size_t inlen,
uint8_t * COUNT(SHA1HashSize) resbuf);
```

Figure 4.2: Annotated HMAC invocation in SMART. Note the use of COUNT annotation for pointers. It specifies the maximum size of buffers used.

4.4.2 MCU Modifications

Simplicity and minimal cost are two important design objectives of SMART . Hardware modifications are therefore limited to memory access checking and the addition of ROM.

[2] One must check that the compiler does not optimize it.

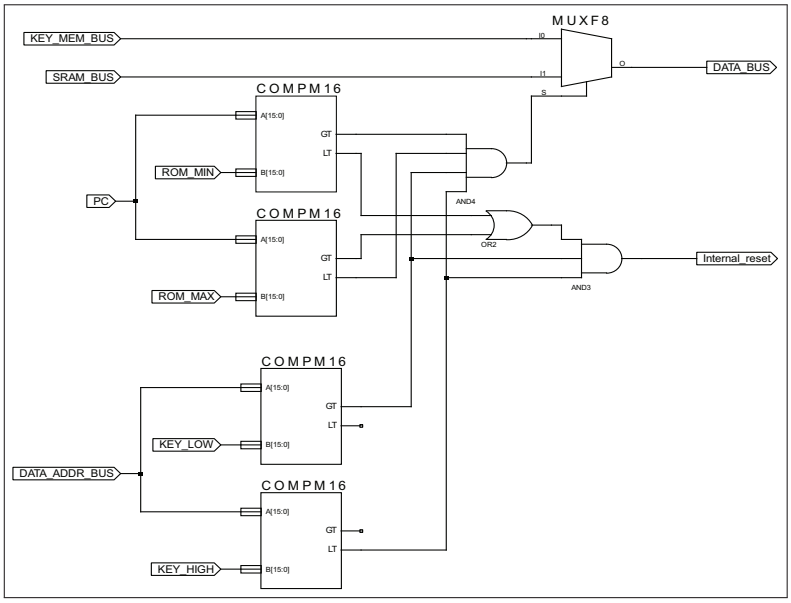


Figure 4.3: Schematic view of access control for attestation key

4.4.2.1 ROM

ROM is a standard feature in many commodity microcontrollers. Typically, a portion of memory is hardwired during manufacture, rendering it immutable. In fact, because it requires little logic it can be built by connecting "wires" and transistors, ROM is the cheapest memory type. In SMART ROM is used for both holding the SMART code and the key.

4.4.2.2 Key Access Controls

Figure 4.3 shows how access to the key is controlled in the MCU. Essentially, the data bus is connected to the key memory bus when the program counter is in ROM and the data address is within the range of addresses where the key is mapped. The internal reset signal is set if the key memory is accessed while the program counter is not in ROM memory range.

4.4.2.3 ROM Execution Control

Because ROM is authorized to access the key its usage must be controlled to prevent key recovery by malicious code on the platform. The adversary can, for example, attempt to selectively execute (invoke) portions of ROM code by using code reuse techniques (e.g., return to libc [Sol97b], borrowed code chunks [Kra05] or Return-Oriented Programming [BRSS08]). To prevent such attacks, we provide additional address controls upon ROM entry and exit. The program counter is

only allowed to move into ROM starting at SMART initial address. Similarly, the program counter can leave ROM only from the last SMART address. This way, ROM code cannot be invoked partially: once any attempt to do otherwise is detected, the MCU is immediately reset.

Those constraints on entry and exit from SMART and the data memory usage needs to be closely controlled. Therefore, SMART code is designed to use only stack allocated memory and is compiled and linked with custom scripts; more details are provided in Section 4.7.

4.5 Security Analysis

Our security argument is informal. A more substantial argument (or a proof) would require formal analysis and verification of SMART code, which is planned as part of future work. The security argument is based on the following assertions:

A1 Cryptographic checksum C computed by $\mathcal{PRV}$ can not be forged. Since C is a result of secure MAC function (e.g., HMAC-SHA1) we assume that, for any $\mathcal{ADV}$ – external to $\mathcal{PRV}$ – that observes a polynomial number of such checksums, finding MAC collisions and/or learning bits of the attestation key is infeasible.

A2 Physical and hardware-based attacks on $\mathcal{PRV}$ are beyond $\mathcal{ADV}$'s capabilities.

A3 Attestation key $\mathcal{K}$ can be accessed only from within ROM-resident SMART code. This is guaranteed by MCU-based access controls.

A4 SMART code can not be modified since it resides in ROM.

A5 SMART code can be only invoked at its beginning. The hardware checks that when the program counter is in ROM code range, excluding the first address of it, it had to be in ROM code range before.

A6 Any invocation of SMART starts by masking (disabling) all interrupts, they remain disabled if, on SMART code completion, control is passed to $\mathcal{HC}$.

A7 Once invoked, SMART code can not be interrupted. This also implies that, regardless of the input, SMART does not encounter any exceptions or reach any undefined state. Hardware checks that if the current value of the program counter is not in SMART code range it must have been previously outside of SMART code range or at the address of the last instruction of SMART code.

A8 Attestation key $\mathcal{K}$ can not be extracted by any software-based $\mathcal{ADV}$ internal to $\mathcal{PRV}$. Before SMART terminates execution, $\mathcal{K}$ (actually, k_a) is securely

erased and the only value based on (statistically dependent on) $\mathcal{K}$ is the output C.

A9 For each invocation, SMART computes a correct value of C based on the contents of the requested memory segment $[a, b]$. Although C is guaranteed to be computed correctly, it may or may not result in $\mathcal{PRV}$ passing attestations, since $[a, b]$ might be previously corrupted by $\mathcal{ADV}$.

A10 Any erroneous state (e.g., violation of assertions A3, A5, A7) lead to a hardware reset. Upon reset, all data memory and registers are erased, which prevents $\mathcal{K}$ leakage. This boot-time memory erasure also guaranties that if power loss occurs during SMART execution, no $\mathcal{K}$ information remains available in memory to untrusted code at next boot.

A11 No observations of the normal execution of SMART when called by an untrusted code on the MCU should reveal information about the key. This could be used to gather information on the key value by comparing several SMART executions with different parameters. This means that execution time and amount of memory used must not be key dependent.

Armed with the above assertions, we can argue that postulated three security objectives are satisfied.

4.5.0.4 Prover Authentication

Once again, since C is correct and r is a random challenge (nonce) of sufficient bit-length, $\mathcal{VRF}$ concludes that C was computed by $\mathcal{PRV}$ within the interval of time between the initial request message and the receipt of C. This yields fresh and timely entity authentication of $\mathcal{PRV}$.

4.5.0.5 External Verification

We assume that $\mathcal{VRF}$ receives and successfully verifies C.[3] Assertions A1-A9 imply that C was computed by SMART code on $\mathcal{PRV}$. Therefore, memory region $[a, b]$ on $\mathcal{PRV}$ contained code or data expected by $\mathcal{VRF}$.

4.5.0.6 Guaranteed Execution

Assertion A8 implies that, immediately after computing C, $\mathcal{PRV}$ commenced execution of code referred to by optional parameter *in*. If C is deemed correct by $\mathcal{VRF}$ and $in = a$, $\mathcal{VRF}$ is assured that the correct code, start at location a was executed. However, this does not mean that the entire code segment, e.g., between

[3]Note that, if C is incorrect, $\mathcal{VRF}$ can not distinguish between cases of: (1) corruption of memory region $[a, b]$ on the actual $\mathcal{PRV}$ and (2) another entity ($\mathcal{ADV}$) attempting to impersonate $\mathcal{PRV}$ or manipulating input or output of SMART.

a and b was executed, which is dependent on the objectives and correctness of this code.

4.5.0.7 Key Protection Guarantee

Assertion A3 implies that $\mathcal{K}$ is not directly available to untrusted software. Assertions A6 and A8 guaranties that code reuse attacks to recover the key are impossible. Assertions A10 implies that when error condition occurs execution is stopped and that there is no information leakage about $\mathcal{K}$. Assertion A11 guarantees that side channels cannot be used to gather information about the key by untrusted software executing on the processor. Other side channels commonly used in key recovery attacks rely on power consumption analysis and electromagnetic emanations[SLP06a], however, those are hardware or physical attacks and we assume that they are infeasible by a "software" local attacker [4].

4.6 Protocols with SMART

In this section we describe several protocols or mechanisms that can be implemented by software components that rely on SMART as a building block.

4.6.1 Remote Attestation of Memory Parts

The most natural usage of SMART is to attest a certain range of memory and verify that it contains the data (or code) that it is supposed to contain. This can be achieved by invoking SMART with the start and end address of the memory range to be attested, as shown in Algorithm 2.

Algorithm 2: SMART usage to attest a range in memory.

input : r nonce sent by verifier
a memory to be attested start address
b memory to be attested end address
$H1$ Hmac function (global variable)
output: the output of the HMAC
begin
SMARTa, b, $\emptyset$, $False$,r,&$H1$,$\emptyset$;
send(H1);
ReActivateInterrupts();
end

[4]We note that those channels might be exploitable in very specific cases, e.g., if the hardware to perform such measurements is available as a peripheral of the device, e.g., a coulomb counter that measures remaining battery power. This could, in theory, provide information on power consumption of SMART code. We assume that such features are not available on the device.

4.6.2 Remote Proof of Reset

In some applications there is a need to ensure that a device has been reset successfully. This can be easily achieved by utilizing SMART as a building block to construct the protocol shown in Algorithm 3. The HMAC guaranties that the R function has been verified and executed,

In that case we asume that the HMAC result can be stored on a memory not erased during reset (e.g. Flash, EEPROM), or is sent before the reset.

Algorithm 3: SMART usage to attest a value, e.g., a reading form a peripheral accessed from memory mapped I/O.

input : r nonce sent by verifier
in address to read from device
R reset function address
$|R|$ the reset function size
H Hmac (global variable)
output: the output of the HMAC
begin
SMARTR, $R + |R|$, R, $True$,r,&$H1$,$\emptyset$;
end
// ResetFunction: R()
begin
$V \longleftarrow$ `ReadValueFromHW()` ;
ShutdownDevices();
EraseAllMemoryButH();
PC = 0 ;
end
// The value H will be returned to verifier after boot
// is completed.

4.6.3 Attested Reading of Measurements

Some applications need to make sure that values read from a peripheral device cannot be forged by malicious code possibly present on that device. For example, large-scale incorrect reports of current electricity consumption by smart meters might lead to power outages. An IMD that returns incorrect values when queried by a physician might result in a incorrect prescription being issued to a patient, with potentially catastrophic consequences.

Predictably, measurements attestation should satisfy the following properties:

- Freshness of the value(s) read.
- Proof of reading the value(s) from the peripheral.

- Integrity of the messages involved.

Freshness is provided via a nonce, present by default in the SMART setup. Proof of reading the value is provided by calling SMART to attest and run $\mathcal{HC}$, which will read the value. Finally $\mathcal{HC}$ will call SMART a second time, as a normal HMAC function, to protect the integrity of the read value. Algorithm 4 presents this primitive.

In summary, the final HMAC contains the following information: the reading function was executed atomically, the verifier can confirm that the function actually reads the value from the correct built-in address and immediately attests it. HMAC correctness therefore ensures integrity of the reported value.

This approach is similar to the extend TPM functionality. Since hashes are chained, they attest all previously hashed data. There are however important differences: HMAC attests each output of SMART with the secret key of the device. This allows for a simpler design. Besides integrity, HMAC correctness confirms that it was produced by SMART. This is fundamentally different from the extend operation preformed by a TPM, since integrity of the PCR is enforced by hardware.

It can be noted that the *Send* function, which sends the HMAC to $\mathcal{VRF}$, is not guaranteed to be executed as it is not verified by SMART . However, this does not impact the validity of the HMAC or the read measurements. On the other hand, it is an availability issue, but malicious code on the MCU could just prevent the whole process to happen (and still be detected by not doing so), solutions for this it is left for future work.

4.6.4 Other Uses and Extensions

In addition to the protocols presented above, a primitive providing a dynamic root of trust can be used for many purposes in embedded devices. For example, if certain known malicious code propagates on networked devices, the verifier can send detection or disinfection code. This code would be launched by SMART to perform remote search for known malicious patterns in code or data. Using SMART, validity of returned HMAC would guarantee that detection code was executed uninterrupted and that the detection result is genuine.

SMART also makes it possible to perform mutual authentication as well as shared key generation between two (or more) previously paired devices that share the same key. In this process, each device acts both as a prover and a verifier. Because SMART guarantees that, even in the presence of full software compromise of either device, a device's long-term attestation key cannot be modified or disclosed. Consequently, the adversary cannot clone a genuine device or eavesdrop on communication between two devices.

Fine-grained access control to sensitive peripherals can be limited to $\mathcal{HC}$ only with simple hardware extensions to SMART. For example, $\mathcal{HC}$ code can be provided

Algorithm 4: SMART usage to attest a measurement, e.g., a reading from a peripheral accessed from memory mapped I/O.

input : r nonce sent by verifier
in address to read from device
R reading function address
$|R|$ the reading function size
$H1$ first Hmac (global variable)
$H2$ second Hmac (global variable)
output: the output of the HMAC
begin
SMARTR, $R + |R|$, R, $True$,r,&$H1$,$\emptyset$;
send(V,H2);
end
// ReadingFunction: R()
begin
$V \longleftarrow$ `ReadValueFromHW()` ;
IN=$V||H1$;
SMART&IN,&$IN + sizeof(IN)$,0,$False$,r,&$H2$,$\emptyset$;
ReActivateInterrupts();
end

in a bundle with its own HMAC and a bit field that describes authorization to access specific memory regions corresponding to memory mapped peripherals. Access to these memory regions would, in turn, be authorized only if HMAC is validated. This is useful in many applications, e.g., pacemakers where it could control delivery of pacing impulses.

4.7 Implementation

To assess the feasibility and impact of SMART we implemented it on two low-end commodity MCU platforms. We believe that this is the best way to understand its advantages and limitations as well as to evaluate the impact of required processor modifications.

We chose to base our implementation on two fully open-source clones of widely used off-the-shelf MCU-s: Atmel AVR and Texas Instruments MSP430. The two processors share many features. They both have a limited memory address space with 16-bit addresses. Common memory sizes in both devices are between $2 - 16$ KBytes of SRAM used as data memory and between $16 - 64$ KBytes of flash memory used for program storage [5]. Both are designed for low-power, low-cost

[5]Some can have slightly larger memory sizes with small changes to the bus size and the instruction set

and are widely adopted in many application areas e.g., in the automotive industry, utility meters, consumer devices and peripherals in addition to wireless sensor networks.

AVR and MSP430 also have some major architectural differences. Notably, MSP430 is a 16-bit Von Neumann architecture processor with common data and code address spaces. Whereas, AVR is an 8-bit Harvard architecture processor that has separate address spaces for data and program memory. Another prominent difference is in the instruction set: AVR is a RISC architecture with most instructions requiring a single 16-bit word and executing in one clock cycle. In contrast, MSP430 can perform multiple memory accesses within a single instruction. Its instruction execution time can range from 1 to 6 clock cycles, and its instruction length can vary from 16 to 48 bits.

The differences between AVR and MSP430 make them good representatives of architectures commonly used in many modern embedded systems. We demonstrate feasibility of SMART by the small increase in area and low impact in terms of performance that we experienced in the process of integrating it into both processors.

4.7.1 Implementation details

SMART implementation consists of three main components:

- Processor modifications to add ROM code memory, key storage and memory access controls.
- Largely architecture-independent SMART routine stored in ROM memory that implements Algorithm 1. This C code has a small number of architecture-dependent lines.
- One or more software protocol implementations that utilize the SMART primitive.

4.7.1.1 Implementation on AVR and MSP430 Cores

We first implemented the hardware part of SMART on the AVR processor, an Atmega103[ATM] clone from the Opencores Project [ope]. Figure 4.4 illustrates the execution core and its memory. Parts that had to be modified or added are shaded. They mainly correspond to memory and memory access controls on memory buses.

Next, we implemented SMART on the MSP430. We used the open-source openmsp430 core from the Opencores Project [ope] and ported SMART to it. The port consists of processor modifications, adaptation of ROM code to MSP430 architecture as well as testing and synthesizing the resulting core. These tasks

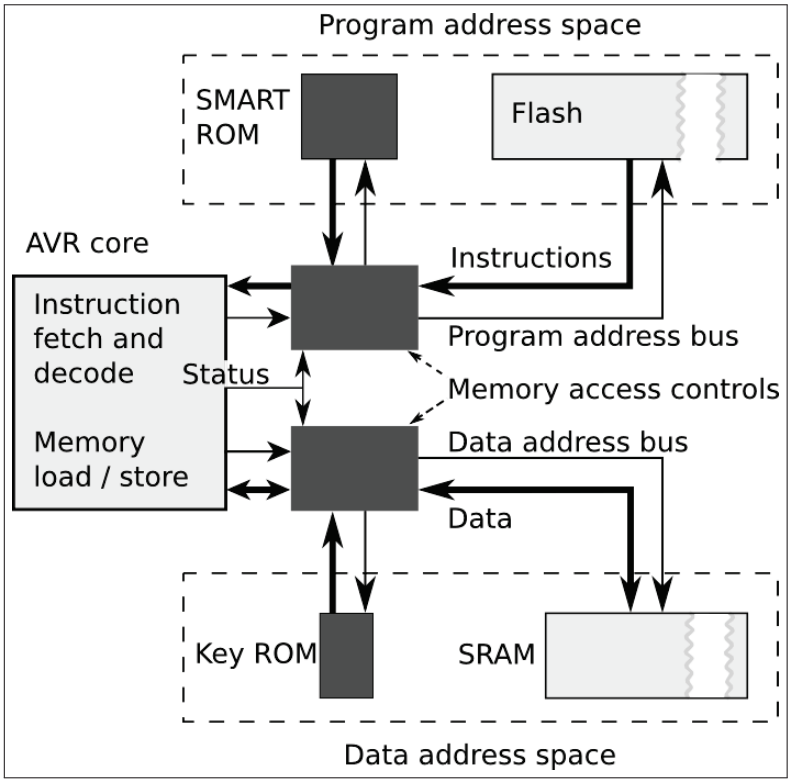


Figure 4.4: Modifications to AVR. Dark gray boxes represent logic added to the processor. Core control signals provide information about internal processor status to memory bus controls.

were performed in one week by one developer with moderate Verilog knowledge and no previous experience with the openmsp430 core. Processor modifications were limited to implementing and adding modules for ROM code and key memory. In addition, minor modifications and address checks were required in the memory backbone module of the openmsp430 core. The memory backbone module performs arbitration of memory accesses. Figure 4.5 presents required modifications (shaded) for MSP430.

In both processors, less than 200 lines of code (Table 4.2) were changed to implement these modifications. In addition to processor modifications, we extended existing regression tests (or test benches) to verify correct implementation of each of assertion from Section 4.5 that is relevant here: A3, A6, A8, and A11.

4.7.1.2 ROM-Resident Code

This code corresponds to 487 lines of portable C and uses a standard SHA-1 implementation [EJ01]. It requires 4KBytes of ROM for the AVR and 6KBytes for MSP430. It executes in 10-s to 100-s of milliseconds (Table 4.7.1.2), depending

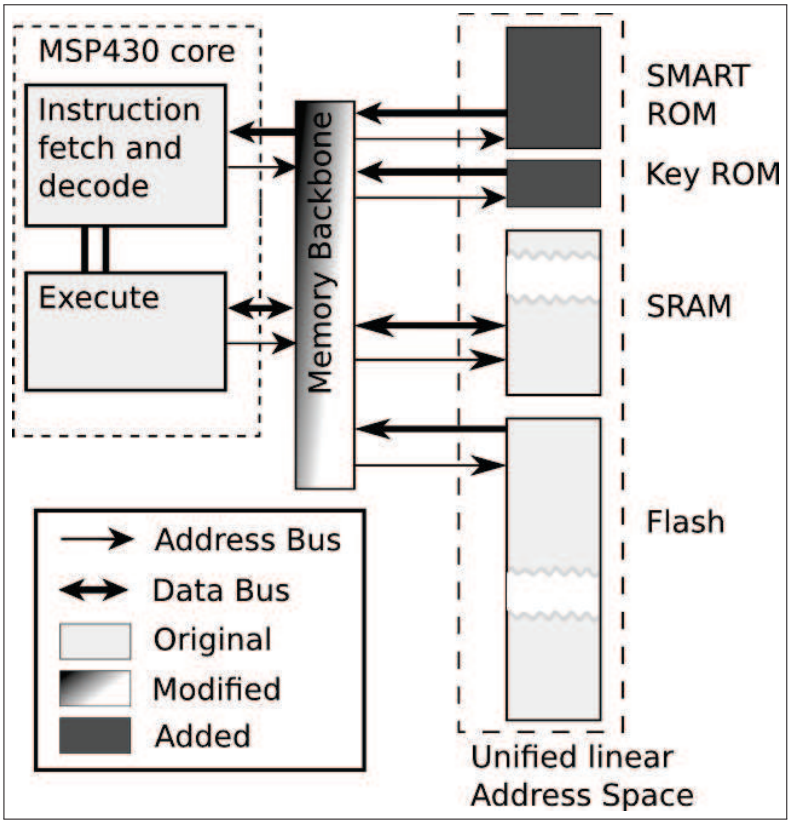


Figure 4.5: Modifications to MSP430. Memory backbone was modified to control access to ROM and $\mathcal{K}$. Since MSP430 is based on Von Neumann architecture, concurrent access can occur to different memory parts (e.g., instruction fetch and read data). In that case, memory backbone arbitrates bus access and temporarily saves/restores data.

on the size of the memory to attest.

Memory usage in SMART has to be carefully managed. SMART code cannot reserve memory for its own usage. Memory should only be allocated on the stack (i.e. local functions variables). It should not attempt to use global variables or heap allocated memory. Finally, the code is compiled and linker scripts are used to generate the ROM image suitable to the modified processor.

4.7.1.3 Hardware Footprint

Simulating the design demonstrates its functional status. Whereas, comparing the number of lines of code of its implementation provides insight about the effort required to implement SMART on a given MCU. However, this is insufficient to assess real impact of SMART in terms of hardware overhead, i.e., surface increase due to its presence on an actual manufactured device. We note that a single line

Data Size	Cycles	Time at 8Mhz
1 KByte	2302281	287 ms
512 Bytes	1281049	160 ms
32 Bytes	387471	48 ms

Table 4.1: HMAC execution timing

of HDL can add a simple wire, a register or an entire memory block; all these would be counted as one line of code, although each has very different impact on synthesized hardware.

To this end, we synthesized the original and SMART-ified designs for both AVR and MSP430. This provides an initial estimate of the impact of SMART on the final devices. Synthesizing is the act of transforming (or compiling) the design from a high-level description language (Verilog or VHDL) into a set of wires and elementary gates that serve as building blocks of an Application-Specific Integrated Circuit (ASIC).

Synthesizing needs to be performed for a specific target hardware. We used the library from UMC 180nm process [Far04c] and Synopsys Design Compiler [Syn10]. Since memory cannot be generated the same way, we used a specific tool [Far04b, Far04a]. However, because we did not have access to a tool to generate FLASH memory, we used the numbers gleaned from publicly available information [Chi11]. Table 4.3 shows the resulting surface as gate equivalents for the original processor and its SMART-ified version.

We stress that these numbers can vary greatly depending on many parameters, such as: required maximum frequency, latency, placement and routing and availability of better memory IP. However, our specific numbers show that the impact of SMART on surface area is minimal. Adding SMART to both AVR and MSP430 caused only a 10% increase in their respective surface areas. As mentioned before, most of that added area is due to ROM housing SMART code. Modifications to the core required only 1K and 0.7K gate equivalents in AVR and MSP430, respectively This could probably be reduced as we did not perform any optimizations.

4.7.2 Lessons Learned from Experiments

The first observation from our experiments is that implementing SMART is not a complex task and porting it to a different architecture is even easier. Second, the additional footprint of our implementation is minimal. One change that impacted the chip surface area the most is the additional ROM memory required to store SMART code.

Another important issue is that, in both cases, we did not have to change the

Component	Original	Changed	Ratio
AVR, core (VHDL)	3932	151	3.84%
AVR, tests	2244	760	
MSP430, core (Verilog)	4593	182	3.96%
MSP430, tests	17665	1122	

Table 4.2: Changes made (in # of HDL lines of code) in AVR and MSP430 processors, respectively, excluding comments and blank lines.

Component		Original Size in kGE	Changed Size in kGE	Ratio
AVR MCU		103	113	10%
Core		11.3	11.6	2.6%
Sram	4 kB	26,6	26.6	0%
Flash	32 kB	65	65	0%
ROM	6 kB	-	10.3	-
MSP430 MCU		128	141	10%
Core		7.6	8.3	9.2%
Sram	10 kB	55.4	55.4	0%
Flash	32 kB	65	65	0%
ROM	4 kB	-	12.7	-

Table 4.3: Comparison of chip surface used by each component of the original MCU to its modified version. kGE stands for thousands of Gate Equivalents (GE-s). One GE is proportional to the surface of the chip and computed form the surface of the module divided by the surface of a NAND2 gate, $9,37 * 10^{-6} mm^2$ with this library.

processor core itself, we only modified the memory access controller.[6] Therefore, SMART might be also well-suited to settings where the processor core is available only as a "black box" and provides enough information about accessed memory on its external interface.

One limitation of our approach is that we rely on reasonably fast HMAC computation, which might make SMART too slow for some applications. This is a consequence of the conscious trade-off made when we chose to limit the amount of hardware changes in the processor. Depending on the application, it may be possible to use a hardware SHA-1 implementation (e.g., [EJ01]), which would significantly improve performance without requiring major processor modifications.

[6]The only exception is that, in some cases, we needed information about the execution engine state (e.g. detection of wait states).

4.8 Discussion

SMART is motivated by lack of currently feasible techniques for providing dynamic a root of trust on remote embedded devices. We proposed SMART – a very simple, lightweight and low-cost architecture that nonetheless offers concrete security guarantees in the presence of any kind of non-physical attacks. Future work will consist in formally verifying the ROM-resident code in order to obtain a strong security proof for the entire architecture; this is likely to be a challenging task. More experiments using current MCU implementations need to be performed to better assess the overhead. We also plan to implement and evaluate SMART on several other common MCU platforms and among a larger project we plan to produce a few test ASIC samples of micro-controllers with SMART.

Chapter 5

Conclusion and Future Directions

Contents

In this work we aimed at laying a foundation for securing the execution of code on embedded devices. As explained, embedded devices have a remarkably simple design when compared to higher end CPU, however they suffer from the same security vulnerabilities. While in the past such devices have benefited from a blanket protection because of their general lack of connectivity, a recent trend is pushing those devices to be increasingly interconnected. Both amongst them and to the outside world via the Internet. As a result, these devices might be remotely attacked and manipulated in the future.

Such devices are widely used in cyber physical systems to operate critical systems, such as industrial systems, energy production and distribution, automotive systems and medical devices. While the reliability of such devices has been widely studied, their security has only recently come under the scrutiny by the research community. As a result, several attacks and vulnerabilities have been discovered in medical devices, automotive electronic control devices, industrial control systems and more. These attacks use different techniques, ranging from remote exploitation to subversion of the update and management protocols. However, their end result is generally the injection of malicious code on the victim device.

5.1 Objectives

Our main objective has been to design efficient and effective solutions to prevent and detect malicious code on embedded devices. On the side of prevention, we

designed IBMAC a hardware supported stack integrity enforcement technique. IBMAC can guarantee that the return addresses stored in the stack are not maliciously manipulated, by restricting access to the return address only to the `ret` and `call` instructions. The solution designed is lightweight in terms of additional gates added to the MCU and does not added any overhead to the computation.

However, as explained, remote exploitation is not the only way malicious code can be injected on an embedded device. It is therefore imperative to design solutions to detect malicious code and behavior in embedded devices via attestation. To this end we first analyzed state of the art software attestation protocols and identified vulnerabilities and hidden assumptions in their design. Specifically, we devised a ROP based rootkit that can defeat any software attestation scheme that only attests code memory, if this specific attack is not taken into account. Furthermore, we devised attacks tailored to two protocols, SWATT and SCUBA, that challenge the assumption that malicious modifications of the attestation routine produce significant time overheads. We then introduced PoSE, a software attestation protocol that relies on a simple premise to achieve attestation: combining attestation with the erasure and update of the previous code on the embedded device. Such a solution has the advantage of being easily proven secure.

Software attestation protocols, however, have quite severe limitations given by their performance and their strict adversarial assumption. We therefore designed a secure attestation solution based on a hardware and software co-design, SMART . In addition to attestation SMART gives the ability to establish a dynamic root of trust on an embedded device and, thus, execute a piece of code untampered even if the entire platform is compromised. SMART was designed to minimize the overhead to the embedded platform both in terms of additional gates (that affect cost and energy consumption) and in terms of time overhead to complete the protocol. The project was inspired by the *late launch* technology implemented via TPMs, however the goal was to have a minimal solution that could be implemented and used on embedded devices.

5.2 Future Directions

5.2.1 Extensions to our Work

The solutions proposed in this work provide a clear basis for the protection of embedded devices, however they are still limited in some factors.

Our control flow protection technique, only prevents stack based buffer overflows. There are other classes of exploits, like heap based overflows, that are common on high end embedded devices and are not covered by our solution. Although we note that the use of a heap is not common on low-end embedded devices, it might still be desirable to design a solution to cover all possible remote

exploitation attacks. IBMAC can be a part of this broader solution, yet to come, as it imposes no real time overhead and only a minimal addition to the hardware.

SMART currently uses an HMAC based on a secret key shared between the verifier and the (secure part) of the prover. A reasonable and needed addition would be to use public key cryptography instead, so that the prover provide attestation of its internal state to any verifier. Similarly to the TPM, the prover would need to be given a public/private key pair certified by the hardware producer to guarantee that the prover has indeed been manufactured according to specification.

Furthermore, SMART now requires the computation of a SHA based MAC over the memory region to be attested. However, computing cryptographic hash functions on embedded devices is a costly operation. It would be desirable to either implement the cryptographic hash function in hardware (which would increase cost) or rely on a lighter weight implementation of a MAC. This latter approach would be based on the fact that the life span of a typical attestation protocol is short, therefore preventing sophisticated and expensive attacks against the MAC function, given the short time available to an attacker.

5.2.2 Future Applications

Medical Devices We plan on adapting the solutions designed in this work to the specific needs of medical devices. In such devices, and especially in implantable medical devices, many of the security goals conflict with the normal operation and requirements for such devices. For example, the computation of an HMAC or a digital signature can incur in too much overhead, both in terms of time and battery consumption. In fact, these devices often operate under tight time constraints and at periodic intervals. Solutions like SMART however require complete control of the device to assure untampered execution and suspend interrupts to avoid malicious software to interfere with their operation. As a solution, one can envision a possible extension in which SMART allow an embedded device to run attested and trusted routines at specific time intervals.

Smart Phone Security This work was mainly focused on low-end embedded devices that have a comparatively simple design compared to commodity devices (e.g., laptop and desktop computers). One possible way to extend this line of research would be to focus on solutions to prevent and detect malicious activity on higher end CPUs, like the ones found in smart phones. In fact, as smart phones become ubiquitously connected to the Internet, we envision that they will become a central target for all sorts of abuse: from remote exploitation, to data theft and participation in botnets. This calls for solutions that will prevent remote exploits and guarantee that the software running on the smart phones remains untampered. Similarly to embedded devices, smart phones have few legacy problems as software and hardware is changed at a fast pace. This calls for clean solutions that can

provably guarantee control flow integrity while remaining efficient for battery powered devices.

Bibliography

[08302] Atmel.
8-bit Microcontroller with 1K Byte of In-System Programmable Flash , AT90S1200, 2002.

[ABEL05] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti.
Control-flow integrity.
In *Proceedings of the 12th ACM conference on Computer and Communications Security, CCS*, pages 340–353. ACM, 2005.

[ADPMT08] Giuseppe Ateniese, Roberto Di Pietro, Luigi V. Mancini, and Gene Tsudik.
Scalable and efficient provable data possession.
In *SecureComm '08: Proceedings of the 4th international conference on Security and privacy in communication networks*, pages 1–10. ACM, 2008.

[Adv] Advance Micro Devices.
AMD, Secure Virtual Machine Architecture Reference Manual.
Publication No. 33047, Revision 3.01, May 2005.

[AFS97] W. A. Arbaugh, D. J. Farber, and J. M. Smith.
A secure and reliable bootstrap architecture.
In *Proceedings of the IEEE Symposium on Security and Privacy, S&P*, page 65. IEEE Computer Society, 1997.

[AK96] Ross Anderson and Markus Kuhn.
Tamper resistance - a cautionary note.
In *Proceedings of the Second Usenix Workshop on Electronic Commerce*, pages 1–11, 1996.

[Ale05] Steven Alexander.
Defeating compiler-level buffer overflow protection.
USENIX ;login:, Vol. 30(3), 2005.

[ATM] ATMEL Corporation.
8-bit microcontroller with 128k bytes in-system programmable flash.

[ATM05] ATMEL Corporation.
ATMEL Trusted Platform Module AT97SC3201, June 2005.

http://www.atmel.com/atmel/acrobat/doc5010.pdf.

[bbM] Pin Flash based bit and Cmos Microcontrollers.
Pic16f688 data sheet.

[BBP+11] Elie Burzstein, Romain Beauxis, Hristo Paskov, Daniele Perito, Celine Fabry, and John Mitchell.
The failure of noise-based non-continuous audio captchas.
In *Proceedings of the IEEE Symposium on Security and Privacy, S&P*, May 2011.

[BCK96] Mihir Bellare, Ran Canetti, and Hugo Krawczyk.
Keying hash functions for message authentication.
In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology, CRYPTO*, pages 1–15. Springer-Verlag, 1996.

[BKR94] Mihir Bellare, Joe Kilian, and Phillip Rogaway.
The security of cipher block chaining.
In *Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology, CRYPTO*, pages 341–358. Springer-Verlag, 1994.

[BRSS08] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage.
When good instructions go bad: generalizing return-oriented programming to RISC.
In *Proceedings of the 15th ACM conference on Computer and Communications Security, CCS*, pages 27–38. ACM, 2008.

[BST00] Arash Baratloo, Navjot Singh, and Timothy Tsai.
Transparent run-time defense against stack smashing attacks.
In *Proceedings of the USENIX Annual Technical Conference*, pages 251–262. USENIX Association, 2000.

[CAE+07] Nathan Cooprider, Will Archer, Eric Eide, David Gay, and John Regehr.
Efficient memory safety for TinyOS.
In *SenSys '07*, pages 205–218. ACM, 2007.

[CCP10] Claude Castelluccia, Emiliano De Cristofaro, and Daniele Perito.
Private information disclosure from web searches.
In *Privacy Enhancing Technology Symposium, PETS*. Springer-Verlag, July 2010.

[CDD+10] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy.
Return-oriented programming without returns.
In *Proceedings of the 17th ACM conference on Computer and Communications Security, CCS*, pages 559–72. ACM Press, October 2010.

[CDP12] Claude Castelluccia, Markus Duermuth, and Daniele Perito.
Adaptive password-strength meters from markov models.
In *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, San Diego, CA, 2012.

[CFK+09] Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav Shacham.
Can DREs provide long-lasting security? the case of return-oriented programming and the avc advantage.
In *Proceedings of the 2009 conference on Electronic voting technology/workshop on trustworthy elections*, EVT/WOTE'09, pages 6–6. USENIX Association, 2009.

[CFPS09a] Claude Castelluccia, Aurelien Francillon, Daniele Perito, and Claudio Soriente.
Defending embedded systems against control flow attacks.
In *ACM Conference on Computer and Communications Security, Workshop on Virtual Machine Security*, November 2009.

[CFPS09b] Claude Castelluccia, Aurelien Francillon, Daniele Perito, and Claudio Soriente.
On the difficulty of software-based attestation of embedded devices.
In *Proceedings of 16th ACM Conference on Computer and Communications Security, CCS*, November 2009.

[CHA+07] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George Necula.
Dependent types for low-level programming.
In *Programming Languages and Systems*, volume 4421 of *Lecture Notes in Computer Science*, pages 520–535. Springer Berlin / Heidelberg, 2007.

[Chi11] Chingis Technology Corporation.
Embedded e^2 Flash IP, PF32K17E, 32Kbyte (16K x16) Embedded Flash Macro (EFM), 2011.
Details available online at: `http://www.chingistek.com/pfusion_03.asp?seq=9`.

[CKMP09] Claude Castelluccia, Mohamed Ali Kaafar, Pere Manils, and Daniele Perito.
Geolocalization of proxied services and its application to fast-flux hidden servers.
In *ACM Internet Measurement Conference , IMC*, November 2009.

[CKN07] Young-Geun Choi, Jeonil Kang, and DaeHun Nyang.
Proactive code verification protocol in wireless sensor network.
In *ICCSA*, volume 4706 of *Lecture Notes in Computer Science*, pages 1085–1096. Springer, 2007.

[Cro] Crossbow Technology Inc.
Micaz datasheet.
http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf.

[CXS+05a] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer.
Non-control-data attacks are realistic threats.
In *Proceedings of the 14th USENIX Security Symposium*, pages 12–12. USENIX Association, 2005.

[CXS+05b] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer.
Non-control-data attacks are realistic threats.
In *Proceedings of the 14th USENIX Security Symposium*, pages 177–192. USENIX Association, 2005.

[DFPT12] Karim El Defrawy, AurÃl'lien Francillon, Daniele Perito, and Gene Tsudik.
Smart: Secure and minimal architecture for (establishing dynamic) root of trust.
In *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, San Diego, CA, 2012.

[EJ01] D. Eastlake and P. Jones.
Rfc 3174 - us secure hash algorithm 1 (sha1).
IETF RFC, September 2001.

[Far04a] Faraday Technology Corporation.
0.18μm synchronous via1 programmable rom compiler, fsa0a_c_sp, 2004.
Details available online at: http://www.faraday-tech.com.

[Far04b] Faraday Technology Corporation.
0.18um synchronous high speed single port memory compiler fsa0a_c_su, 2004.
Details available online at: http://www.faraday-tech.com.

[Far04c] Faraday Technology Corporation.
Faraday fsa0a c 0.18 μm asic standard cell library, 2004.
Details available online at: http://www.faraday-tech.com.

[FC08] Aurélien Francillon and Claude Castelluccia.
Code injection attacks on Harvard-architecture devices.
In *Proceedings of the 15th ACM conference on Computer and Communications Security, CCS*, pages 15–26. ACM, 2008.

[fCIP07] European Programme for Critical Infrastructure Protection.
Communication from the commission on a european programme for critical infrastructure protection.

Technical report, 2007.
`http://eur-lex.europa.eu/LexUriServ/site/en/com/2006/com2006_0786en01.pdf`.

[FMC11] Nicolas Falliere, Liam O Murchu, and Eric Chien.
W32.stuxnet dossier, version 1.4.
Symantec Security Response, February 2011.

[FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken.
Flow-sensitive type qualifiers.
In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 1–12. ACM, 2002.

[GN07] Vanessa Gratzer and David Naccache.
Alien vs. quine.
IEEE Security and Privacy, 5:26–31, 2007.

[HHF09a] Ralf Hund, Thorsten Holz, and Felix Freiling.
Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms.
In *Usenix security*, 2009.

[HHF09b] Ralf Hund, Thorsten Holz, and Felix C. Freiling.
Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms.
In *Proceedings of the 18th USENIX Security Symposium*. USENIX Association, August 2009.

[HRMM+11] Steve Hanna, Rolf Rolles, Andr's Molina-Markham, Pongsin Poosankam, Kevin Fu, and Dawn Song.
Take two software updates and see me in the morning: The case for software security evaluations of medical devices.
In *2nd USENIX Workshop on Health Security and Privacy*. USENIX Association, 2011.

[HSH+08] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten.
Lest we remember: Cold boot attacks on encryption keys.
In *Proceedings of the 17th USENIX Security Symposium*, pages 45–60. USENIX Association, 2008.

[HTC+10] Wen Hu, Hailun Tan, Peter Corke, Wen Chan Shih, and Sanjay Jha.
Toward trusted wireless sensor networks.
ACM Trans. Sen. Netw., 7:5:1–5:25, August 2010.

[Huf62] Huffman, D.A.
A method for the constructionof minimum redundancy codes.

Proceedings of the IRE, 40:1098–1101, 1962.

[Int09] Intel Corporation.
Intel Trusted Execution Technology (Intel TXT) – Software Development Guide, December 2009.
Document Number: 315168-006.

[JJ10] Markus Jakobsson and Karl-Anders Johansson.
Assured detection of malware with applications to mobile platforms.
Technical report, DIMACS, February 2010.
available at http://dimacs.rutgers.edu/TechnicalReports/TechReports/2010/2010-03.pdf.

[KJ03] Rick Kennell and Leah H. Jamieson.
Establishing the genuinity of remote computer systems.
In *Proceedings of the 12th USENIX Security Symposium*, pages 21–21. USENIX Association, 2003.

[KK10] Alexei C. Karl Koscher.
Experimental Security Analysis of a Modern Automobile.
In *Proceedings of the IEEE Symposium on Security and Privacy, S&P*, May 2010.

[KLNP07] Cynthia Kuo, Mark Luk, Rohit Negi, and Adrian Perrig.
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*, pages 233–246. ACM, 2007.

[Kra05] Sebastian Krahmer.
x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique.
Technical report, suse, September 2005.
available at http://www.suse.de/ krahmer/no-nx.pdf.

[KS04] Alexander Klimov and Adi Shamir.
New cryptographic primitives based on multiword t-functions.
In *Fast Software Encryption, 11th International Workshop, FSE 2004*, pages 1–15, 2004.

[KSA+09] Chongkyung Kil, Emre C. Sezer, Ahmed M. Azab, Peng Ning, and Xiaolan Zhang.
Remote attestation to dynamic system properties: Towards providing complete system integrity evidence.
In *DSN 09: Proceedings of the 39th IEEE/IFIP Conference on Dependable Systems and Networks*, June 2009.

[Lee08] Edward A. Lee.
Cyber physical systems: Design challenges.

In *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, May 2008.
Invited Paper.

[MLQ+10] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig.
Trustvisor: Efficient tcb reduction and attestation.
In *Proceedings of the IEEE Symposium on Security and Privacy, S&P*, May 2010.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp.
Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security).
Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[MPP+08] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki.
Flicker: an execution infrastructure for tcb minimization.
In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 315–328. ACM, 2008.

[MPS09] Ivan Martinovic, Paul Pichota, and Jens B. Schmitt.
Jamming for good: a fresh approach to authentic communication in wsns.
In *Proceedings of the 2nd ACM conference on Wireless network security, WiSec*, pages 161–168. ACM, 2009.

[Mud95] Mudge.
How to write buffer overflows.
`http://www.insecure.org/stf/mudge\_buffer\_overflow\_tutorial.html`, 1995.

[NN07] "Federal Networking and Information Technology R&D Program (NITRD)".
Leadership under challenge: Information technology r&d in a competitive world.
Technical report, 2007.
http://www.whitehouse.gov/sites/default/files/microsites/ostp/pcast-07-nitrd-review.pdf.

[One96] Aleph One.
Smashing The Stack For Fun And Profit.
Phrack, 7(49), November 1996.

[ope] The Opencores Project.
`http://opencores.org/`.

[PCKM11] Daniele Perito, Claude Castelluccia, Mohamed Ali Kaafar, and Pere Manils.

How unique and traceable are usernames?
In *Privacy Enhancing Technology Symposium, PETS*. Springer-Verlag, 2011.

[PS05] Taejoon Park and Kang G. Shin.
Soft tamper-proofing via program integrity verification in wireless sensor networks.
IEEE Trans. Mob. Comput., 4(3):297–309, 2005.

[PT10] Daniele Perito and Gene Tsudik.
Secure code update for embedded devices via proofs of secure erasure.
In *Proceedings of the 15th European conference on Research in computer security, ESORICS*, pages 643–662, 2010.

[RRW05] John Regehr, Alastair Reid, and Kirk Webb.
Eliminating stack overflow by abstract interpretation.
Trans. on Embedded Computing Sys., 4(4):751–778, 2005.

[SCT04] Umesh Shankar, Monica Chew, and J. D. Tygar.
Side effects are not sufficient to authenticate software.
In *Proceedings of the 13th USENIX Security Symposium*. USENIX Association, August 2004.

[Sha07] Hovav Shacham.
The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86).
In *Proceedings of the 14th ACM conference on Computer and Communications Security, CCS*, pages 552–61. ACM, 2007.

[Sko05] Sergei P. Skorobogatov.
Semi-invasive attacks – A new approach to hardware security analysis.
Technical Report UCAM-CL-TR-630, University of Cambridge, Computer Laboratory, April 2005.

[SLP06a] Kai Schramm, Kerstin Lemke, and Christof Paar.
Embedded cryptography: Side channel attacks.
In *Embedded Security in Cars*, pages 187–206. Springer Berlin Heidelberg, 2006.

[SLP+06b] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla.
SCUBA: Secure code update by attestation in sensor networks.
In *WiSe '06: Proceedings of the 5th ACM workshop on Wireless security*, pages 85–94. ACM, 2006.

[SLP08] Arvind Seshadri, Mark Luk, and Adrian Perrig.
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*, pages 372–385. Springer-Verlag, 2008.

[SLS+05] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla.
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*, pages 1–16. ACM, 2005.

[SLW07] Marc Stevens, Arjen Lenstra, and Benne Weger.
Chosen-prefix collisions for md5 and colliding x.509 certificates for different identities.
In *Proceedings of the 26th annual international conference on Advances in Cryptology, EUROCRYPT*, pages 1–22. Springer-Verlag, 2007.

[SMKK05] Mark Shaneck, Karthikeyan Mahadevan, Vishal Kher, and Yongdae Kim.
Remote software-based attestation for wireless sensors.
In *ESAS*, pages 27–41, 2005.

[Soc] IEEE Computer Society.
Wireless medium access control (mac) and physical layer (phy) specifications for low-rate wireless personal area networks (wpans).
`http://standards.ieee.org/getieee802/download/802.15.4-2006.pdf`.

[Sol97a] Solar Designer.
Non-Executable User Stack, August 1997.

[Sol97b] Solar Designer.
return-to-libc attack.
Bugtraq mailing list, August 1997.

[Spa89] Eugene H. Spafford.
The internet worm program: An analysis.
Computer Communication Review, 19, 1989.

[SPP+04] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh.
On the effectiveness of address-space randomization.
In *Proceedings of the 11th ACM conference on Computer and Communications Security, CCS*, pages 298–307. ACM Press, October 2004.

[SPvDK04] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep K. Khosla.
SWATT: SoftWare-based ATTestation for embedded devices.

In *Proceedings of the IEEE Symposium on Security and Privacy, S&P*, pages 272–. IEEE Computer Society, 2004.

[SSW11] Steffen Schulz, Ahmad-Reza Sadeghi, and Christian Wachsmann.
Short paper: lightweight remote attestation using physical functions.
In *Proceedings of the 4th ACM conference on Wireless network security, WiSec*, pages 109–114. ACM, 2011.

[Sta] StackShield.
http://www.angelfire.com/sk/stackshield/.

[Syn10] Synopsys, Inc.
Design compiler 2010, 2010.
`http://www.synopsys.com/home.aspx`.

[SZJvD04] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn.
Design and implementation of a tcg-based integrity measurement architecture.
In *Proceedings of the 13th USENIX Security Symposium*, pages 16–16. USENIX Association, 2004.

[Tex] Texas Instruments.
Msp430 f1611 datasheet.

[The] The PaX Team.
Pax address space layout randomization (aslr).
http://pax.grsecurity.net/docs/aslr.txt.

[TLP05] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg.
Avrora: scalable sensor network simulation with precise timing.
In *IPSN*, page 67. IEEE Press, 2005.

[Trua] Trusted Computing Group.
Tcpa main specification, version 1.1b.

[Trub] Trusted Computing Group.
Tpm main specification level 2 version 1.2.

[WWH+10] Scott Wolchok, Eric Wustrow, J. Alex Halderman, Hari K. Prasad, Arun Kankipati, Sai Krishna Sakhamuri, Vasavya Yagati, and Rop Gonggrijp.
Security analysis of india's electronic voting machines.
In *Proceedings of the 17th ACM conference on Computer and communications security, CCS*, pages 1–14. ACM, 2010.

[XKPI02] J. Xu, Z. Kalbarczyk, S. Patel, and R. Iyer.
Architecture support for defending against buffer overflow attacks, 2002.

[YWZC07] Yi Yang, Xinran Wang, Sencun Zhu, and Guohong Cao.
Distributed software-based attestation for node compromise detection in sensor networks.

In *SRDS*, pages 219–230. IEEE Computer Society, 2007.