Arbitrary Code Injection through Self-propagating Worms in Von Neumann Architecture Devices

Thanassis Giannetsos[1], Tassos Dimitriou[1], Ioannis Krontiris[2] and Neeli R. Prasad[3]

[1]*Athens Information Tech.*
19.5 km Markopoulo Ave.
Athens, Greece
[2]*Computer Science Dep.*
University of Mannheim
D-68161 Mannheim, Germany
[3]*Department of Communication*
Aalborg University
Fr. Bajers Vej 7A5, DK-9220,Denmark
Email: agia@ait.edu.gr, tdim@ait.edu.gr, krontiris@uni-mannheim.de, np@es.aau.dk

Malicious code (or malware) is defined as software designed to execute attacks on software systems and fulfill the harmful intents of an attacker. As lightweight embedded devices become more ubiquitous and increasingly networked, they present a new and very disturbing target for malware developers. In this paper, we demonstrate how to execute malware on wireless sensor nodes that are based on the Von Neumann architecture. We achieve this by exploiting a buffer overflow vulnerability to smash the call stack and intrude a remote node over the radio channel. By breaking the malware into multiple packets, the attacker can inject arbitrarily long malicious code to the node and completely take control of it. Then we proceed to show how the malware can be crafted to become a self-replicating worm that broadcasts itself and infects the network in a hop-by-hop manner. To our knowledge, this is the first instance of a self-propagating worm that provides a detailed analysis along with instructions in order to execute arbitrary malicious code. We also provide a complete implementation of our attack, measure its effectiveness in terms of time taken for the worm to propagate to the entire sensor network and, finally, suggest possible countermeasures.

1. INTRODUCTION

Sensor nodes are deeply embedded wireless computing devices without tamper-proof hardware due to cost considerations. This, along with the fact that they are deployed in unattended environments makes it possible for an attacker to physically capture a node and run unauthorized software on the device or extract sensitive information like cryptographic keys [1]. Given that sensor networks consist of hundreds or even thousands of sensor nodes, several security protocols have been based on the assumption that an attacker can compromise and control only a small portion of the network.

Recent advances in sensor networks research have shown, however, that an attacker can exploit different mechanisms of sensor nodes and spread malicious code throughout the whole network without physical contact. One such method is to take advantage of the network programming capabilities of sensor networks, which allow the dissemination of code updates through wireless links and reprogramming of nodes after deployment. Over-the-air programming (OAP) is a fundamental service in sensor networks that relies upon reliable broadcast for efficient code dissemination and updates. However, it faces threats from both external attackers and potentially compromised nodes. For example, an adversary may easily subvert such protocols by modifying or replacing the real code image being propagated to sensor nodes, introducing malicious code into the sensor network [2]. Nevertheless, such mechanisms have been secured recently, allowing the propagation of only authenticated program images originating from the base station [3, 4, 5, 6].

Another method for the attacker is to exploit memory related vulnerabilities, like buffer overflows, to launch a worm attack. Since all sensor nodes execute the same program image, finding such a vulnerability can lead to the construction of self-propagating packets that inject

malicious code to their victims and transfer execution to that code. If the malware is constructed such as it resends itself to the neighbors of the node, the attacker can compromise the whole network and quickly take complete control of it. While this attack is extremely dangerous, there has been very little research in this area.

All previous work has concentrated on sensor devices following the Harvard architecture, in which, even though it is possible to construct propagating malicious packets, an attacker cannot directly inject and execute her *own* code inside the mote's memory. This is due to the characteristics of this architecture, since data and code segments are physically and logically separated not allowing the execution of instruction injected in the data memory. Therefore, it is only possible to transfer the program execution to already existed sequences of instructions present in the program memory. But even then, the injection process suffers from code size and time limitations.

This is not true, however, for sensor devices following the Von Neumann architecture. According to this architecture, both instructions and data are stored in the same memory space, allowing the attacker to transfer execution control where the malicious packet is stored. This allows the injection and execution of arbitrary code that does not necessarily exist in the mote's memory. Several of the most popular sensor node platforms today use microcontrollers that are based on the Von Neumann architecture, like for example the Tmote Sky [7], Telos [8], EyesIFX [9], ScatterWeb MSB-430 [10] or even the SHIMMER platform for medical applications [11]. Hence, it is important to show how vulnerable these platforms are against worm attacks and emphasize the need for appropriate prevention measures.

1.1. Our Contribution

In this paper we present the design and implementation of a self-propagating worm for wireless sensor devices based on the Von Neumann architecture. It is the first complete instance of such a worm containing full technical details and all necessary code instructions. In particular, we make three contributions beyond previous work on code injection attacks.

First, we describe an implementation that further advances already existing tendencies in the use of "*multistage buffer-overflow attacks*", in order to inject and execute arbitrarily long code in sensor devices following the Von Neumann architecture. In a multistage buffer-overflow attack, an adversary sends a number of specially crafted packets in order to inject long blocks of code. This type of attack bypasses any code size limitations and its effectiveness does not rely on pre-existing instruction sequences in program's memory as opposed to any previous work in this area. Second, we demonstrate how the injected malware can self-propagate, that is be converted into a "worm". This is a serious threat since the attacker can compromise the entire sensor network by infecting only one node. Additionally, we put out the code instructions of the sensor worm that an adversary needs to inject into a node before spreading to the entire network. Finally, we evaluate the efficiency of worm propagation derived from a real network consisted of Tmote Sky sensor devices. This assessment relates to the time needed by the worm to reach 100% of sensors in a particular neighborhood. However, it is also coupled with extensive simulations that demonstrate the efficiency of worm propagation in large scale networks. Previous code injection approaches relied on the assumption that injection can only be done byte-to-byte, resulting in schemes that require an important amount of time. This, however, may lead to possible detection of the attack in progress.

Even though research in worms against several types of networks has increased significantly over the last years, existing literature in sensor networks is quite limited. To the best of our knowledge, this is the first instance of a self-propagating worm that provides all the necessary tools that can be used by an adversary, along with a complete experimental evaluation of its effectiveness. We expect that our work will be particularly useful in sensor network research for showing the destructive impacts of a sensor worm and highlighting the need to come up with efficient mechanisms to counter such attacks.

1.2. Paper Organization

The remainder of this paper is organized as follows. First, we discuss related work in Section 2 and state our assumptions in Section 3. Then, Section 4 overviews the TI MSP 430 architecture, which is our example platform, while Sections 5, 6 and 7 provide the details of the code injection attack. Section 8 describes how to build a worm so that the malware can be installed in all nodes of the network. In Section 9, we present detailed performance measurements and experimental results, and in Section 10, we discuss possible prevention measures. Finally, Section 11 concludes the paper.

2. RELATED WORK

Although exploitation of code injection attacks due to memory faults have been studied thoroughly in computer systems [**?**, 16], only recently this has been applied to sensor networks as well. Goodspeed first showed how to perform a buffer overflow attack on the MSP 430 microcontroller in order to execute instructions within a received packet [12, 13]. The author noted that packets in TinyOS are always stored at the same address in the data memory, so overwriting the program counter (PC) with that address makes the execution of malicious code possible. Even though

TABLE 1. Comparison of code injection attacks.

Property/Code injection Attack	[12, 13]	[14]	[15]	This Work
Arbitrary injection code size	Partial	No	Partial	Yes
Self-propagation	No	No	Partial	Yes
Attack does not rely on pre-existing code	Yes	No	No	Yes
Stealthiness of attack (i.e., mote's execution is not disrupted)	N/A	N/A	No	Yes
Efficiency Evaluation (i.e., propagation time, etc.)	N/A	N/A	N/A	Yes

it was mentioned that injection of code of arbitrary length can be done through transmission of a number of malicious packets, it was not shown how this can be achieved and more importantly how the injected code can propagate itself.

When a buffer overflow occurs, the program execution is disrupted. Hence, in order for the node to be able to receive further malicious packets, restoration of control flow needs to take place after each reception. In this paper we are based on the above technique to launch a *multistage buffer-overflow* attack ([17, 18]) in order to directly inject arbitrary long code inside the mote and successfully build a worm.

Sensor devices following the Harvard architecture have also been studied with respect to code injection attacks. The execution of instructions carried by malicious packets is not possible in such devices due to the physical separation of program and data memory spaces. However, it has been demonstrated how to invoke functions of already *existing* application code. In [14], the authors showed that sensor applications are susceptible to control-data attacks that alter control flow to utilize existing routines in order to propagate the injected packet further to the network. This attack though does not disrupt the normal operation of the motes, so the security threat is low.

Recently, Francillon and Castellucia [15] took a step further and showed how code injection can be achieved in sensor devices featuring the Harvard architecture. They demonstrated how an attacker can exploit a program vulnerability in order to execute an instruction sequence, called a *gadget*, that already exists in program memory. Through the execution of a gadget chain comprised of *Injection* and *Reprogramming* meta-gadgets, they showed how a fake stack can be injected byte-to-byte into data memory and used for reprogramming the sensor with a new program image.

However, only malware of size up to 256 bytes (one program image) can be injected using this technique. Larger code needs to be split into program images which should be injected separately. Thus, since the injection can only be done byte-to-byte, this requires an important amount of time that may lead to possible detection of the attack in progress. In addition, the described attack is *disruptive* meaning that each injection causes the sensor device to reset itself. This may, again, expose the attack or even lead the mote to an unstable state where further execution of the malicious code is prohibited. Furthermore, although possible convertibility of the injected malware into a worm is hinted, no definite references of how this can be achieved are given (i.e code instructions that must be executed by the worm to replicate and send itself). In our work, the attack is considered to be *stealthy* and not constrained by the assumption of gadgets that must *pre-exist* in program's memory. Instead, the attacker can inject arbitrarily long malware of her own choice.

Finally, some additional research has been conducted examining the destructive effects of worms in several applications in wireless networks. As correct node operation is an important requirement in sensor networks, malicious code injection can be used to disrupt the network operation by deviating from the prescribed protocol or to launch internal attacks. Davis [19] discussed how such attacker techniques severely threaten today's Smart Meter and Advanced Metering Infrastructure (AMI) technology that can be used to measure, collect and analyze energy usage, from advanced devices such as electricity meters, gas meters, etc. He was able to identify multiple programming errors on a series of Smart Meter platforms ranging from the inappropriate use of banned functions to protocol implementation issues. Then, he took advantage of these vulnerabilities and created an in-flash rootkit, which allowed him to assume full system control of all exposed Smart Meter capabilities. Also, Goodspeed [20] talked about stack overflows and how they can be used to infiltrate security of second generation Zigbee radio chips. More particularly, he showed how vulnerable these chips are to key theft due to unprotected Data memory. These works can be thought as complementary to our own.

Concluding, a comparison between the most important code injection attacks and the one described here is shown in Table 1, although most of them target sensor devices featuring the Harvard architecture. A "N/A" indication shows that a property cannot be directly deduced or that is not a part of the proposed attack. A "Partial" characterization indicates that the corresponding property is not entirely achieved.

3. ASSUMPTIONS

Through out this work we assume a sensor network that is homogeneous in both hardware and software. All sensor nodes execute the same program image, as

it is true for the majority of sensor networks today. This means that if a vulnerability that the attacker can exploit for launching a code injection attack is discovered, all the other nodes will be vulnerable to the same attack.

We also assume that sensor nodes are loaded with a simple C-based operating system, like TinyOS [21], which uses the NesC programming language. This allows us to look for well known buffer overflow techniques, since code safety is not considered in such systems. The use of Java in other paradigms seems more secure, as it provides built-in protections against code-based attacks that would exploit array boundaries, unchecked cast, pointer arithmetics, etc. Also, the virtual machine examines compliance of incoming code with the Java standards before execution [22]. Still, TinyOS is the most widely adopted operating system in sensor networks, as it is extremely lightweight for such constrained devices.

4. TI MSP 430 ARCHITECTURE OVERVIEW

The platform targeted in this attack is the MoteIV Tmote Sky [7], as it is one of the most widely used platforms in WSNs. However, any platform following the Von Neumann architecture falls prey to similar attacks. The Tmote Sky module uses the ultra low power TI MSP 430 F1611 microcontroller [23] featuring 10 KB of RAM, 48 KB of flash, 128 KB of information storage, and an IEEE 802.15.4 compliant wireless transceiver [24].

4.1. The TI MSP 430 microcontroller

The Texas Instruments (TI) MSP 430 family of processors [9, 25] are low power 16-bit devices. They incorporate a 16-bit byte-addressed RISC processor, peripherals and a flexible clock system that interconnect with each other using a Von Neumann common memory address bus (MAB) and a memory data bus (MDB) as shown in Figure 1. In such microcontrollers, program and data memories share a common address space. Thus, the CPU can load instructions by addressing the complete memory address range through MAB and MDB.

The CPU implements an instruction decoder and a register file. The instruction decoder is responsible for translating the numeric op-code program instructions into processor actions. As core instructions are stored in words, code is accessed at even addresses. The lower byte of a word is always at an even address and the high byte is at the next odd address. The register file consists of 16 registers of 16 bits, numbered from R0 to R15. The first 4 of them have a special role, whereas the rest are for general use, e.g. holding instruction operands or function arguments.

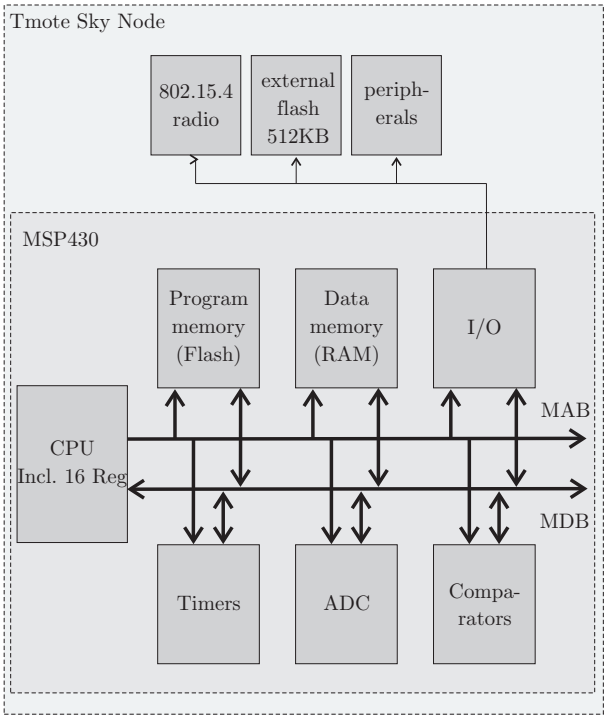

FIGURE 1. Tmote Sky memory architecture showing the common address space used by the CPU.

4.2. Memory Layout

As shown in Figure 1, the MSP 430 Von Neumann architecture has one address space shared with special function registers (SFRs), peripherals, RAM and Flash Code memory. The amount of each type of memory varies with the type of microcontroller used, but the overall layout is common and shown in Figure 2.

The total RAM memory consists of two separate memory modules: lower RAM (0200 - $09FF$) and upper RAM (1100 - $38FF$). However, since RAM must be comprised of consecutive blocks of address space, the lower RAM module is not actually used by the microcontroller and its contents are mirrored inside a specific area of the upper RAM module. Thus, the actual RAM usable by the CPU really starts at address 1100 and it is contiguous.

The internal extended RAM memory implements two main data structures: *stack* and *heap*. The stack is responsible for storing data and the return addresses of subroutine calls and interrupts. It starts at the top of the memory and grows downwards. The special-purpose register R1 is the *stack pointer*, which at any given time points to the last value placed on the stack. Values are pushed as 16-bit words and after each push the stack pointer is decremented by 2. Correspondingly, as values are pulled from the stack, the stack pointer is increased by 2.

The heap is the area of memory used for dynamic allocation and grows upwards from the bottom of memory. However, since TinyOS does not support

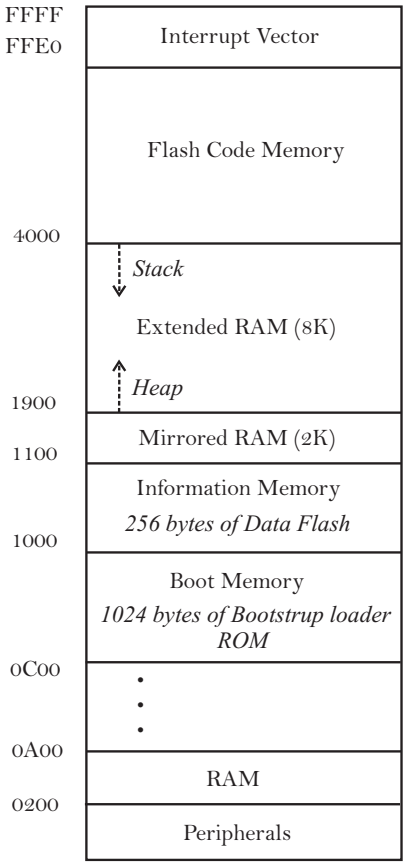

FIGURE 2. Memory map of the MSP 430 controller.

dynamic allocation of memory during runtime, the address region between the heap and the stack will be *empty and unused* during program execution.

The main Flash memory is always at the highest address ($FFFF$). It can be used for storing both code and data. It also contains the interrupt vectors along with the power-up starting address. Each vector contains the 16-bit address of the appropriate interrupt-handler instruction sequence. The boot memory is an unalterable masked ROM containing the serial bootloader. It is actually a factory set program to erase and reprogram the on-board flash memory.

5. CHALLENGES OF CODE INJECTION ATTACKS ON SENSOR DEVICES

Buffer overflows are a leading type of security vulnerability. They are the result of programming flaws and are perfect for code injection attacks. They occur when a malformed input is being used to overflow a buffer, overwriting the return address that is stored on the stack. In this way control can be transferred to code placed either in the buffer, or past the end of the buffer [26].

Even though it is possible to inject and execute malicious code to a platform featuring the Von Neumann architecture, one has to consider several factors in order to launch the attack successfully. First, since code injection attacks are based on changing the flow of control in a program, this may lead the sensor to restart itself or go into an unstable state, where further execution of the attack code is canceled.

Furthermore, sensor nodes characteristics and constraints limit the capabilities of an attacker, who may want to send large blocks of code that exceed the allowed packet size. For example, TinyOS sets the default maximum size of packet payload to be 28 bytes. Although this can be increased, still the maximum payload size in IEEE 802.15.4 is 102 bytes. Thus, in order to send a meaningful piece of code, one has to break it down and send it through multiple packets.

We should also stress that it is best for the whole attack code to reside in a *contiguous* memory region so that it can be executed without any disruptions. Therefore, the attacker must perform a "*multistage buffer-overflow attack*" [17, 18], where she can manipulate an arbitrary address pointer and modify the data it points to. Then by sending a number of packets containing consecutive blocks of code and copying them into the memory space where this pointer shows, she can create a contiguous region containing the attack code.

6. BUFFER OVERFLOW DESCRIPTION

This section describes how a buffer overflow can be exploited in sensor devices featuring the Von Neumann architecture. The input data runs over and overflows the stack, i.e., the section of memory that was set aside to accept it. Since sensor devices are typically designed for specific target applications [27, 28], they are based on a very different memory architecture than commodity embedded devices [29]. Thus, it is reasonable to question how such a vulnerability can be exploited by an attacker in sensor networks.

Two issues need to be addressed in order to understand how stack-based buffer overflows can be performed on sensor networks.

- *How the attack code is sent and stored on sensor nodes.* As described previously, in the MSP 430 family of processors, both code and data memories share a common address space. Therefore, a block of the attack code can be sent as data payload of a message and stored into memory as a piece of data. Exploitation of buffer overflow attacks may then result in alteration of programs execution control flow since in order for a received packet to be processed, a memory buffer is needed.
- *Where the attack code is stored.* Since TinyOS doesn't support dynamic memory allocation, all needed memory for data storage, variables, functions etc. is allocated automatically during compilation. Thus, for a specific program image and hardware platform, memory addresses reserved for particular operations will be the same. Sending

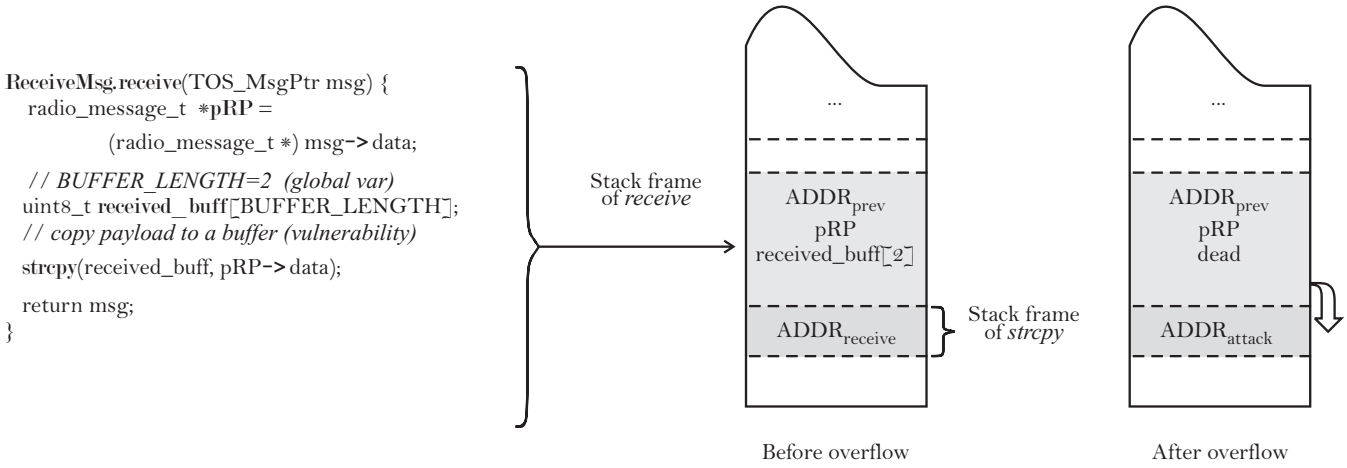

FIGURE 3. Stack frames before and after buffer overflow.

a piece of malicious code in the payload of a message will result in being stored as data at the memory address designated for storing received packets.

In this paper, we focus on stack-based buffer overflow attacks that redirect control flow by "*smashing*" the stack, which was created when a vulnerable function was called. If it succeeds, it will cause the return address of this function to be overwritten and eventually the instruction pointer will point to the location of the injected code and start executing it. Hence, the exploitation technique needs to provide an arbitrary return address that will replace the already existing one. In our case, this address is the one reserved by the compiler for storing the payload of a received packet. Goodspeed, was the first that observed the effectiveness of such an exploitation in MSP 430 microcontrollers, by changing the program counter to point to the address where received packets are stored [12, 13].

In the remainder of this section, we show how the stack can be manipulated in order for the attacker to execute malicious code residing at the memory space of a received packet. We assume that the sensor node has a routine for processing received packets and that this routine has a vulnerability, as shown in Figure 3. Upon reception, it copies the contents of the packet payload into the array `received_buff`, using the standard C *strcpy* method. If the length of the payload data exceeds `BUFFER_LENGTH`, a buffer overflow occurs.

Each time a routine method is called, a stack frame is created for storing temporary data and the return address $ADDR_{previous}$ of the caller function, as we mentioned in Section 4.2. The routine will contain an instruction sequence that ends with a *ret* command for fetching this address from the stack and so control returns to the caller function. For our reception routine, a stack frame of 6 bytes will be created, as shown in Figure 3; two bytes for the message pointer *pRP*, two bytes for the array *received_buff*, and two more bytes for storing the return address. Below this stack frame resides the stack frame of the *strcpy* subroutine call. This contains $ADDR_{receive}$, which points to the next instruction of the receive function that must be executed after *strcpy* returns. This is the address that will be overwritten during the buffer overflow attack.

To start the attack, the attacking node broadcasts a packet, the data field of which contains the attack code to be executed and the address $ADDR_{attack}$ of that code inside the packet. Two extra dummy bytes in the string (i.e., *dead*) are needed in order to overflow the *received_buff* array and overwrite the return address $ADDR_{receive}$. Let us note that, as we mentioned in Section 4.1, the MSP 430 microcontroller architecture requires that the attack code must be evenly aligned, otherwise the sensor resets itself. For the same reason, address $ADDR_{attack}$ must point exactly to the first instruction of the attack code, within the stored packet.

The contents of the stack right after the execution of the *strcpy* function are illustrated in Figure 3. Thus, sending a packet with an appropriate string in the data payload will change the 2-byte return address of a receiving sensor ($ADDR_{receive}$) to the address $ADDR_{attack}$. This will lead the instruction pointer to continue the program execution where the attack code resides in program memory.

Although this attack can be really dangerous for the sensors vitality, it limits the attacker with regards to the length of the attack code that can be executed, as we said in Section 5. The set of malicious instructions is limited to the length of the message payload. In the section below, we will explain how a buffer overflow can be integrated into a non memory-constrained code injection attack where the attacker sends malicious code of arbitrary length.

7. EXPLOITING BUFFER OVERFLOW FOR CODE INJECTION ATTACKS

In order to send arbitrarily long blocks of code, we are using the "*multistage buffer-overflow attack*" ([17, 18]). Multistage buffer-overflow is a type of attack that requires several steps of buffer overflow. It allows the attacker to manipulate an arbitrary address pointer and modify the data it points to. So, by sending a number of specially crafted packets that result in consecutive buffer overflows, the attacker has the ability to copy malicious code from one memory location (payload of received message) to another (region pointed by the selected address pointer), and eventually have her attack code stored in a *contiguous* memory region, starting from a memory address of her choice. This type of attack will bypass the limitations of a single buffer overflow, in which the length of the attack code cannot exceed the size of a message payload.

Fundamental to this attack is that we define an address pointer, namely $ADDR_{copyTo}$, which points to a memory region that is both *writable* and *unused*. Unused means that the address space referring to this memory region is not used by the program during its execution and therefore cannot be altered. This is important since the attack code must remain unaffected and not get overwritten by any program data. As we said in Section 4.2, such a memory space can be found in the MSP 430 microcontroller between the stack and the heap. This address space is the *target region* for an attacker to store the malicious code.

Since this region is unused by the running application, it is also unaffected by possible reboots of the sensor node. Thus, once the attacker injects her code into a sensor node, the code will remain there throughout the lifetime of the sensor node. For the case of a Tmote Sky sensor device, we found that a suitable target region starts at address 2574 and grows upwards.

Let us now overview the steps of a multistage buffer-overflow attack, before we get into details:

1. The attacker sends a specially-crafted packet to the target node that, through buffer overflow, redirects its normal execution to the address of the payload. This results in copying a block of malicious code to the region pointed by the target address $ADDR_{copyTo}$. The last instruction within the packet's payload restores the normal state and program flow of the sensor node.
2. Step 1 is repeated n times, where n is the number of packets needed for injecting the whole attack code into the sensor. At each repetition, an appropriate offset is added to the target address $ADDR_{copyTo}$, in order for the code to reside in consecutive memory addresses.
3. When transmission of the code is finished, the attacker sends one last specially-crafted packet that redirects the control flow to the beginning of the malicious code in the target region, so that it can be executed.

TABLE 2. MSP 430 assembly code instructions.

Instruction	Opcode	Description
MOV src, dst	$0x40b2$	Source operand is moved to the destination.
BR dst	$0x4030$	Branch to an address anywhere in the 64 K address space.

Being aware of the steps that an attacker must follow, two major aspects of a multistage buffer-overflow attack need to be addressed: how to craft the malicious packets so that we can restore the program flow and be able to send more packets, and how to update the target pointer so that malicious code is copied in consecutive memory locations. We show how to achieve each of these steps in separate sections below.

7.1. Composition of Crafted-Packet Payload and Restoration of Program Flow

As described above, an attacker sends a series of malicious packets and executes the code in their payload. The goal of each injected packet is to copy data (malicious instructions) from one memory location to another. Hence, a malicious packet must contain a block of the attack code and all needed instructions for copying this block to the target region.

When a buffer overflow occurs, it brings the sensor device to an inconsistent state. However, since the first step of the multistage buffer-overflow attack must be repeated n times, it is important to restore the control flow, as if program instructions were executed normally. Otherwise, further reception of malicious packets will not be possible. So, after the successful completion of a buffer overflow, a malicious packet needs to further alter the program flow in order to re-establish consistent state. This means that an intermediate packet must also contain a specific instruction that will be executed last and it will restore the program counter.

The MSP 430 assembly language has dedicated instructions for setting the contents of an address to a specific value and for manipulating the program counter. These are the `MOV` and `BR` instructions, respectively, as shown in Table 2. They have unique 2-byte op-codes decoded by the CPU. Since *src* and *dst* operands are defined as data words, they can carry 16-bit values. This means that each `MOV` instruction can copy 2 bytes of the attack code to the target region.

So, the malicious packet will consist of a sequence of `MOV` instructions followed by a final `BR` instruction. Their purpose is to copy bytes of the attack code residing in the payload to the target region and restore control flow of the program. Let's assume that the payload of a packet is set to its default maximum

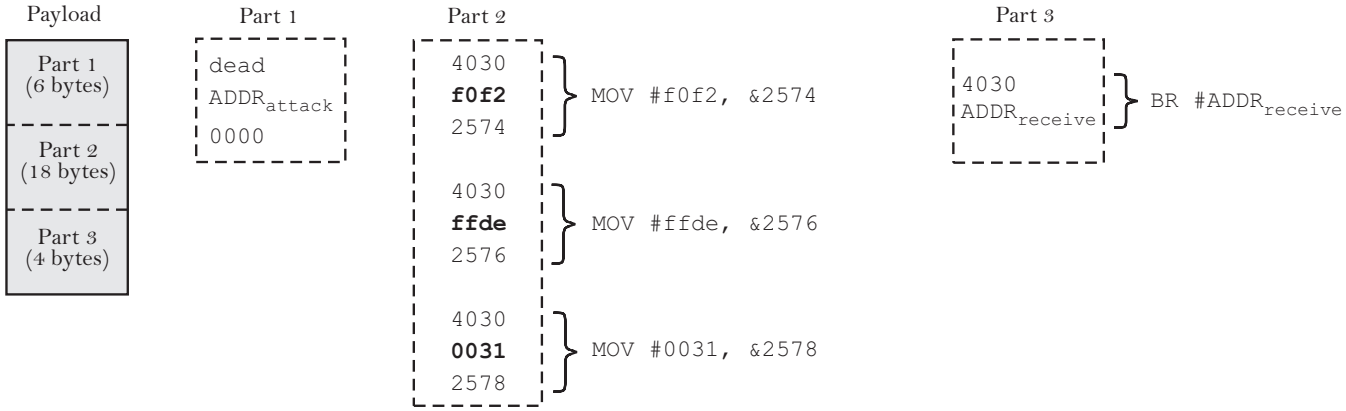

FIGURE 4. Malicious packet payload.

size of 28 bytes[4]. As we described in Section 6, 6 bytes overall (*dead*, $ADDR_{attack}$ and 0000) are needed for overflowing the `received_buff` of the reception routine and overwriting the return address with the starting address of the malicious code. Data bytes *0000* are used as the terminating character for stopping the buffer overflow and prevent further damage of the stack. The remaining 22 bytes are used for carrying the attack instructions to be executed. Since we also need a `BR` instruction that requires 4 bytes, 18 bytes are actually left for sending the necessary `MOV` commands. Therefore, with each malicious packet, a 6-byte block of the attack code can be copied to the target region.

As illustrated in Figure 4, the payload consists of three parts. The first part provides the data for buffer overflow, as well as the attack address $ADDR_{attack}$, at which the control flow is directed when the exploited vulnerable function returns. The second part provides the necessary `MOV` instructions for copying the 6 bytes of the attack code to the target region. Finally, the third part provides the `BR` instruction for restoring the control flow. This is accomplished by setting the program counter to point to the address where the execution would normally return to, after the *receive* function, i.e., $ADDR_{receive}$.

For the example shown in Figure 4, the 6 bytes of the attack code are designated in bold. These specific bytes correspond to the first instruction of the code instance shown below. Its (malicious!) functionality simply turns on the green LED of a sensor node.

AND.B *#ffde,&0031*
BIS.B #2, *&125e*

The target region, where the malcode bytes are copied, starts at address 2574. Following the same process, the whole malware can be installed in the target region. Once this is done, it can be activated by a final buffer overflow exploit. A malicious packet is sent containing only one `BR` instruction for redirecting the program counter to point to the starting address of the malicious code, 2574.

[4]This is the worst case for an attacker, since as we mentioned in Section 5, the actual maximum payload size of a message can be increased up to 102 bytes for radios compliant to IEEE 802.15.4, as it is the case with our example platform.

7.2. Update of the Target Pointer

Fundamental to a multistage buffer-overflow attack is that the attack code must reside in a contiguous memory region. Otherwise, activation of this code may lead the sensor node to an unstable state and cause it to reboot itself.

This issue is resolved through the use of a target pointer. Initially, this pointer is set to the beginning of the unused memory region where the attack code will be stored. In our case, this address is equal to 2574. Every time a `MOV` instruction is executed, a 2-byte block of the malicious code will be copied to the memory location pointed by $ADDR_{copyTo}$.

When a memory injection packet is received by a sensor node, a buffer overflow occurs. After the successful completion of this attack, the `MOV` instructions of the packets payload will be executed and copy k bytes (k is multiple of 2) of code to the target region. These bytes must be stored in $\frac{k}{2}$ consecutive memory addresses, starting where the $ADDR_{copyTo}$ points at the time. Thus, after a `MOV` operation, the target address must be incremented by 2 in order to point to the next memory address.

Algorithm 1: Incrementing *target address*

```
Data: Part 2 of the malicious packet payload
begin
    for each MOV instruction do
        MOV src, ADDR_copyTo
        ADD #2, ADDR_copyTo
    end
end
```

For example, the malicious packet payload that was constructed in the example of the previous section

contained 3 MOV instructions (see Figure 4). As this was the first packet to be sent, the $ADDR_{copyTo}$ pointed to address 2574. An overview of how the target address is incremented after each MOV operation is shown in Algorithm 1.

7.3. Control Flow of the Code Injection Attack

This section summarizes the program's control flow during the progress of a multistage buffer-overflow attack. As described previously, a number of packets need to be sent for the whole attack code to be copied in the target region. Thus, each malicious packet needs to alter control flow several times in order to allow further reception of packets.

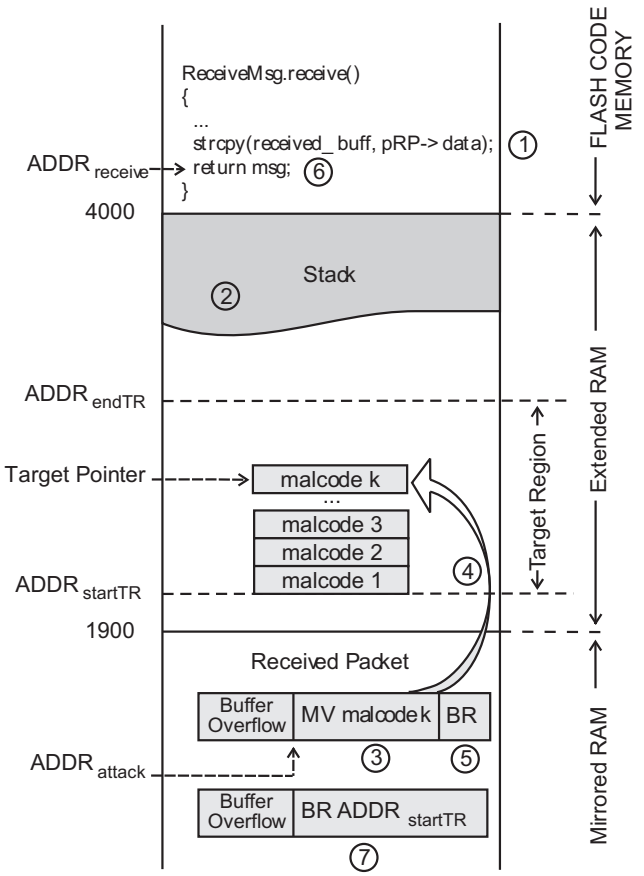

FIGURE 5. Control flow under multistage buffer-overflow attack.

Figure 5 illustrates the execution flow upon reception of the k-th malicious packet. It also shows the specially-crafted packet sent at the end of the attack for activating the injected malware. Details of the operations that take place are provided below:

1. Vulnerable function *strcpy()* is called from the reception routine.
2. A buffer overflow occurs resulting in the overwrite of the return address ($ADDR_{receive}$), stored in the stack frame of the *strcpy()*, with the starting address $ADDR_{attack}$ of the attack code. $ADDR_{attack}$ points to the MOV instructions contained in the packet's payload.
3. When *strcpy()* finishes its execution, control flow is redirected to $ADDR_{attack}$ memory address.
4. MOV instructions are executed for copying malcode bytes to consecutive memory addresses starting from where the target pointer (TP) points at the time.
5. The BR instruction that occupies the last 4 bytes of the malicious packet payload is executed in order to restore program's control flow.
6. Program execution continues normally. This is accomplished by setting the program counter to point to $ADDR_{receive}$ memory address of the receive function.
7. Once the attack code is stored in the target region, the last specially-crafted packet is sent for activating it. Its payload contains a BR instruction that is executed for setting the instruction pointer to the starting address of the target region, $ADDR_{startTR}$ (2574 in our case).

Once the multistage buffer-overflow attack is complete, the attacker would have succeeded to remotely inject a malware into a sensor node and eventually change its functionality. The next section shows how this malware can be converted into a "worm" for propagating itself and infecting the entire sensor network.

8. DISSEMINATION OF ATTACK CODE - WORM CONSTRUCTION

Taking code injection attack one step further, this section describes how the injected malware can self-propagate, i.e., be converted into a "*worm*". Clearly this is a serious threat [30], if not the most dangerous one, since the attacker can compromise the entire sensor network by infecting just a single node.

When a worm is injected to a sensor, it launches a program for broadcasting itself to other neighboring nodes, infecting them as well. At no time does the worm need user assistance in order to spread its "infection". All interconnected nodes are at risk of the attack, as the worm travels over the air and propagates hop by hop.

The main idea is that once the malware is installed, it launches another multistage buffer overflow attack, this time targeting the neighboring nodes. For this purpose, it builds a number of memory injection packets that contain its own code and broadcasts them using the host's radio.

The injected code consists of two parts. The first part provides the necessary instructions for further disseminating the whole attack code. The second part contains the malicious code that was added by the attacker, in order to take control of the infected sensor node.

Once activated, the worm will break down the injected code into malicious packets and start broadcasting them. Each time it needs to send a packet, it has

TABLE 3. Arguments of Transmission Task.

Argument	Description	Register
addr	Destination address	R15
length	Size of payload	R14
*msg	Message	R13

to use a transmission function in the infected sensor. One such function that is widely used in sensor applications is the *SendMsg* routine of the *GenericComm* component:

*GenericComm$Send$SendMsg(uint32_t addr, uint8_t length, message_t *msg)*

In order to invoke the above transmission function, the malware needs to provide specific arguments that are passed through *general purpose* registers. The *GenericComm$Send$SendMsg* function actually posts the transmission task *CC2420RadioM$PacketSent$runTask* that also requires 3 arguments. As shown in Table 3, the arguments are passed via registers R13 to R15.

So, the malware must execute certain instructions for loading the right arguments to registers R13 to R15, before calling the transmission function. In particular, it needs to execute some `MOV` instructions with the *src* operand to be the desired value and the *dst* operand to be the register. After this is done, the invocation of the transmission routine is achieved by simply executing a $CALL\ \#ADDR_{send}$ instruction, where $ADDR_{send}$ is the memory address of the routine.

When the transmission function is called, it loads the necessary data arguments from the corresponding registers and posts a task to the TinyOS scheduler. This task is actually a deferred procedure call. At some point later, the scheduler will run this task through the *runTask* routine that will invoke the *CC2420RadioM$PacketSent$runTask* event with the passed parameters. Since the call of the transmission function is done manually through the attacker's code, the malware is also responsible for the invocation of the *runTask* routine. In Section 8.2, we will cover the details of the above described instruction sequence that is contained in the malware and how it is executed by the scheduler.

8.1. Aftermath of Worm Propagation

After having propagated itself successfully, the execution of the attack code proceeds to the second part of the core sensor worm functionality. This includes code instructions running for the attacker's purpose, i.e., taking over the infected sensor node or stop its execution. Examples of what an attacker can do are listed below:

1. Bring down the entire network by sending a signal to the sensor node for stopping its execution. This is achieved by setting the instruction pointer to the beginning of a special function, which can be found in every sensor device loaded with an executable program image, and call *stop_program_execution*.
2. Create a malicious procedure for draining node battery, or compromising the security level of the network by conducting erasing actions for memory and cryptographic keys.
3. Tell the sensor node to report back vital information, like its neighbor IDs, data structure of the network messages, or any possible stored values and keys etc. The exposure of such information can lead to the total break down of the network's operation.
4. Add new functionalities to the already existing ones. This will allow the sensor node to carry out the attacker's tasks without disrupting it's normal functionality.
5. Achieve full system control of all exposed sensor hardware, including remote power on, power off, usage reporting, and communication configuration [19, 20].

Note that the above described actions are only a subset of what an attacker can actually do. Once the worm has infected the whole sensor network, its administration passes to the attacker's hands.

TABLE 4. Important Memory Addresses.

Memory Address	Description
$ADDR_{startTR}$	Address containing the first instruction of the attack code.
$ADDR_{attack}$	Address where the payload of a received packet is stored.
$ADDR_{packetSent}$	Address of the malicious packet to be sent.
$ADDR_{payloadSent}$	Address of the malicious packet's payload.
$ADDR_{receive}$	Address pointing to the instruction of the reception routine that must be executed after the vulnerable function returns.
$ADDR_{send}$	Address of the transmission function.
$ADDR_{task}$	Address of the *runTask* routine.
$ADDR_{endTR}$	Address containing the last instruction of the attack code

8.2. Implementation Details

In this section, we present the complete code of the sensor worm that we have implemented. For demonstrating its feasibility, we load the sensor nodes with an application that just reports sensor readings to the base station. The sensor application has a vulnerable reception routine that copies the packet payload into a buffer without checking its boundary, as shown in Section 6.

The code needed for self-propagation occupies 166 bytes, whereas the malicious code that is added by the attacker is of arbitrary length. Thus, at least 28 packets are needed for injecting the attack code into the unused memory region. Once the code is injected, it is activated and broadcasts itself by invoking the transmission function in the infected node.

Algorithm 2: Sensor Worm Assembly Code

Data: Memory addresses of Table 4
begin

1 MOV $\#ADDR_{startTR}$, R5
2 MOV #0, R6
3 MOV $\#(ADDR_{payloadSent} + 6)$, R7
4 MOV R5, R8
5 MOV *#dead*, $\&ADDR_{payloadSent}$
6 MOV $\#ADDR_{attack}$, $\&(ADDR_{payloadSent} + 2)$
7 MOV *#0000*, $\&(ADDR_{payloadSent} + 4)$
8 MOV *#40b2*, 0(R7)
9 ADD #2, R7
10 MOV @R8, 0(R7)
11 ADD #2, R7
12 MOV R8, 0(R7)
13 ADD #2, R7
14 ADD #2, R8
15 INC.B R6
16 CMP.B #3, R6
17 JNC -30
18 MOV *#4030*, 0(R7)
19 ADD #2, R7
20 MOV $\#ADDR_{receive}$, 0(R7)
21 MOV $\#ADDR_{packetSent}$, R13
22 MOV.B *#length*, R14
23 MOV *#addr*, R15
24 CALL $\#ADDR_{send}$
25 MOV #0, R9
26 MOV.B #1, R15
27 CALL $\#ADDR_{task}$
28 INC R9
29 CMP #4, R9
30 JNC -14
31 ADD #6, R5
32 CMP $\#ADDR_{endTR}$, R5
33 JNC -114
34 Repeat instructions 5-7
35 MOV *#4030*, $\&(ADDR_{payloadSent} + 6)$
36 MOV $\#ADDR_{startTR}$, $\&(ADDR_{payloadSent} + 8)$
37 Repeat instructions 25-30

ARBITRARY MALICIOUS CODE

end

Two functions are involved in the exploitation of the sensor worm. Function *SendMsg* is the transmission routine that is called by the worm, whenever a malicious packet needs to be sent. Task *runTask* which is invoked by the scheduler, once the *SendMsg* is called.

TABLE 5. Functionality of Instruction Sets

Instruction Set	Description
2-17	Construction of the malicious packet payload.
18-24	Invocation of the transmission function.
25-30	Invocation of the *runTask* routine.
1-33	Repetition of the above instruction subsets as many times as needed for dissemination of the whole attack code.
34-37	Construction and transmission of the specially crafted packet for redirection of control flow.

Table 4 lists some important memory addresses that are used by the worm. As described in Section 7, $ADDR_{startTR}$ is the beginning address of the target region, where the attack code will be stored and, in the case of a Tmote Sky sensor device, is equal to 2574. The values of all other memory addresses depend on the binary representation of the program image that is loaded in the sensor node. For the example of our implemented application, these values were found by looking into the memory of a sensor using the JTAG interface provided by the MSP 430 microcontroller.

Algorithm 2 contains the complete code of the sensor worm. Detailed explanation of the instruction sequence is provided in Table 5 that shows the block structure of the code and the functionality of each block.

As we can see, the malware is a chain of instruction sets (IS) each one of them designated for a specific operation. Instructions 2-17 constitute an IS for creating the payload of a malicious packet to be sent, as described in Section 7.1. This is achieved by setting appropriate values to the memory addresses pointing to the payload starting from address $ADDR_{payloadSent}$.

Instructions 18-24 are the first part of the IS responsible for broadcasting a malicious packet. It calls the transmission function which resides in address $ADDR_{send}$ in the program memory. As mentioned previously, the invocation of such a function requires the upload of proper arguments through registers R13 to R15. Instructions 21-23 are intended for exactly this purpose. Continuing to the second part of this IS, instructions 25-30 call the runTask routine that invokes the *CC2420RadioM$PacketSent$runTask* task for actually broadcasting a malicious packet. The above instruction sets are repeated as many times as needed for the whole malware (stored in address space bounded from $ADDR_{startTR}$ to $ADDR_{endTR}$) to be disseminated to the node's neighbors.

Finally, instructions 34-37 construct and send the specially-crafted packet for redirecting control flow to

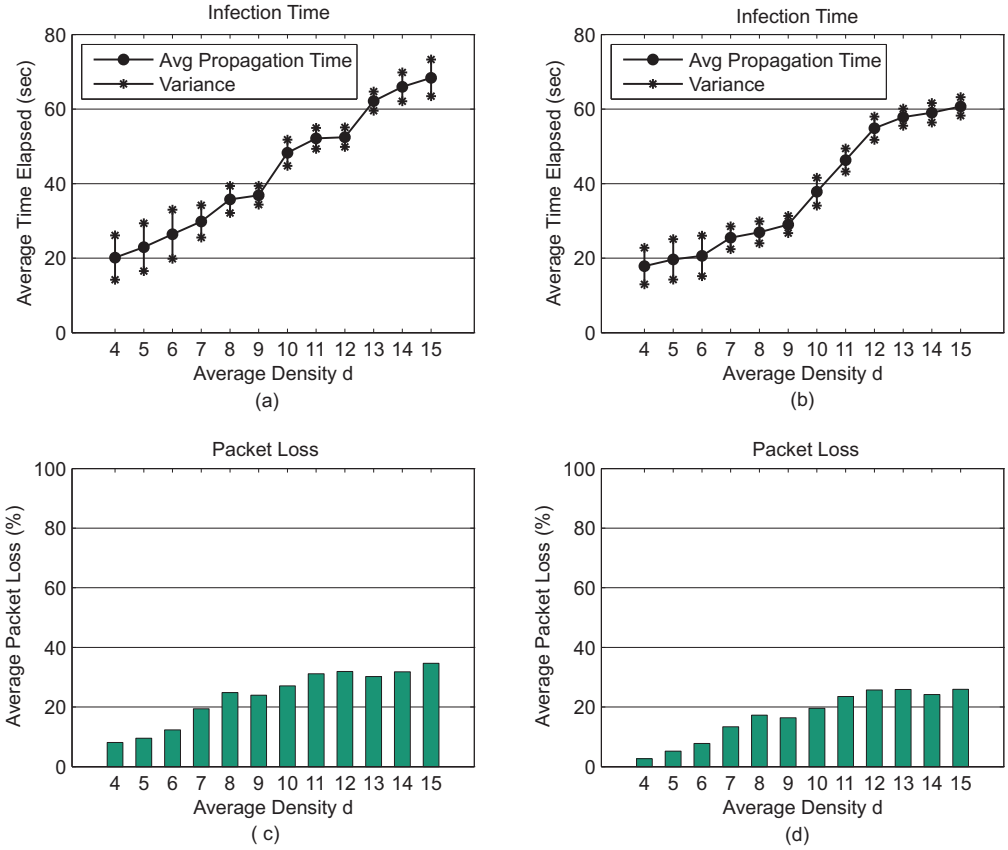

FIGURE 6. Infection time and Packet loss for different packet rates at the application (Delta) and routing (MultihopLQI) layer when the malicious code is 28 bytes. (a) Delta(5 sec) - MultihopLQI(5 sec). (b) Delta(5 sec) - MultihopLQI(30 sec). (c) Packet Loss for case (a). (d) Packet Loss for case (b)

the beginning of the target region.

Note that the above described example intends to demonstrate the effectiveness of a sensor worm, when a program vulnerability is present. It does not show how to find a vulnerability, since this is not the purpose of this paper.

9. PERFORMANCE EVALUATION

In this section, we present the experimental results from our implementation of the described sensor worm. The experiments were deployed both in a simulator and a real sensor environment. One property was of special interest, namely *Propagation Delay*. Since the worm always reaches 100% of exposed sensor nodes, our focus is mainly on the time needed to achieve complete network infection.

9.1. Experiments on real sensor devices

In order to judge the performance of our worm we first evaluated, on real Tmote Sky sensor devices, the time needed for the worm to infect all nodes in the neighborhood of the attacker. The goal is to justify the practicality of the proposed implementation from a real deployment point of view. As the worm will propagate in waves, infecting one neighborhood after the other and many nodes in parallel, this will give us a better feeling of what to expect on large scale networks (this will be examined more thoroughly in Section 9.2).

The experiments were conducted by deploying a varying number of nodes and averaging the results over 20 different runs. We designated a node as the "attack source" who started the infection by injecting the malware to its neighbors. Network nodes were running a typical monitoring application (Delta), as discussed in Section 8.2. We set the size of inserted *malicious code* to be 28 bytes. Note that the fixed length of *self-propagation code* (166 bytes) is also taken into consideration in the results.

Since the worm propagates itself through message transmissions, it is reasonable to mention that its dissemination depends upon successful broadcasts. However, this is not guaranteed, as packets may get lost due to traffic overhead and channel collisions. That is why we performed the experiments having the aggregative data rate of packets at the routing and application layer taken into account. More specifically, we loaded the Delta application, where the motes can

use to report environmental measurements to the base station in user defined intervals – in our case every 5 seconds. We also deployed the MultihopLQI [31] protocol at the routing layer which, by default, is tuned to send control packets and routing information every 30 seconds. Our goal is to demonstrate the propagation delay of the sensor worm, even under the presence of heavy traffic on other layers.

Figures 6(a) and (b), respectively, depict the time needed by the worm to infect all the nodes residing in the neighborhood of the "attack source" for increasing data rate of packets and average density (i.e. *neighborhood size*) d varying between 4 and 15. This increase actually corresponds to different packet rates for Delta and MultihopLQI; for Figure 6(a) they were both tuned at 1 packet per 5 seconds, whereas for Figure 6(b) they were tuned at 1 packet per 5 seconds and 30 seconds, respectively.

What we can infer from these figures is that the propagation delay is low and depends on the success or failure of the broadcast transmissions. When the injected malware is activated, it broadcasts itself by initiating a transmission sequence of all needed malicious packets. Note that the radio component of a neighboring node to be infected may not be ready (or occupied) at that time and thus some of the malicious packets may not be received. Furthermore, a radio transmission may interfere with other signals and fail. Hence, a node might miss to receive a number of malicious packets[5]. This can result in the addition of an extra delay, since the node will receive the missing packets from subsequent transmissions of its infected neighbors.

As more and more packets are sent and received from a node, an increase to the packet loss is unavoidable. This can be seen in Figures 6(c) and (d), where the average packet loss, for the above described scenarios, is illustrated. When the traffic in a neighborhood is heavy (i.e. Delta and MultihopLQI transmit packets every 5 seconds), the number of lost malicious packets is relatively high. This results in an overall increase of the infection time since some of the nodes will have to wait for later transmissions in order to receive all the worm packets. It also explains the high variance seen in the propagation delay, as it is proportional to the number of needed transmission sequences.

We should stress, however, that these figures depict what happens when we focus at each neighborhood and "isolate" it from the remaining network, for densities ranging from 4 nodes (sparse networks) to 15 nodes (dense networks). The increased time for larger neighborhood sizes may seem counterintuitive at first since the attacker node still *broadcasts* its malicious payload and one expects more (if not all) nodes to be infected at once. As, we explained, this is due to the increased number of collisions and missed packets. This also means that nodes in a neighborhood will not be infected in a single round but in more than one, accounting for the increased infection time. This is the reason why dense neighborhoods exhibit such a bad behavior. Does this mean, however, that dense networks will need more time to be infected? The answer is no!

As we will demonstrate in the next section, when we look at what happens in the network level, things will improve dramatically since spreading of infection will start as soon as a node gets infected. This is due to the effect of random worm propagation. This effect allows parallel transfers of data, thus it is possible for a number of nodes that reside in different regions of the network to receive different copies of the worm. Such a case is exemplified in Figure 7.

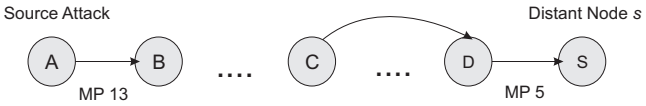

FIGURE 7. Node B needs to wait for later transmission of node A for malicious packet 13 whereas distant node S has already started the reception of the malware. This is based on the existence of a path comprised of already infected nodes through C.

In this case, node B lost a number of malicious packets during the first transmission and needs to wait for later broadcast of node A. However, at the same time, distant node S starts receiving the malware from node D resulting in a significantly decrease in the propagation overhead. This is based on the observation that since the propagation starts once a node is infected, it may be the case that some nodes in a path will actually be infected in a single transmission round and thus will reach some regions of the network faster than others.

In the next section, we we will see that the time needed to infect an entire network really depends on the number of hops required to reach the most distant nodes, as this determines the number of intermediate transmissions. But then, this number is inversely proportional to the density of the network.

9.2. Simulation Results of Large Scale Networks

We used a simulated sensor testbed to measure the performance of our implemented worm, and to demonstrate the feasibility of our approach in large scale networks. Simulation results allowed for further evaluation of the sensor worm with regard to the propagation time needed to infect all the sensor nodes, as the size of the network scales.

[5]We believe that inserting a more reliable transmission mechanism will reduce the number of lost malicious packets and drop the delay further. In future work, we plan to add code instructions that will enable the worm to re-broadcast missed packets, if necessary.

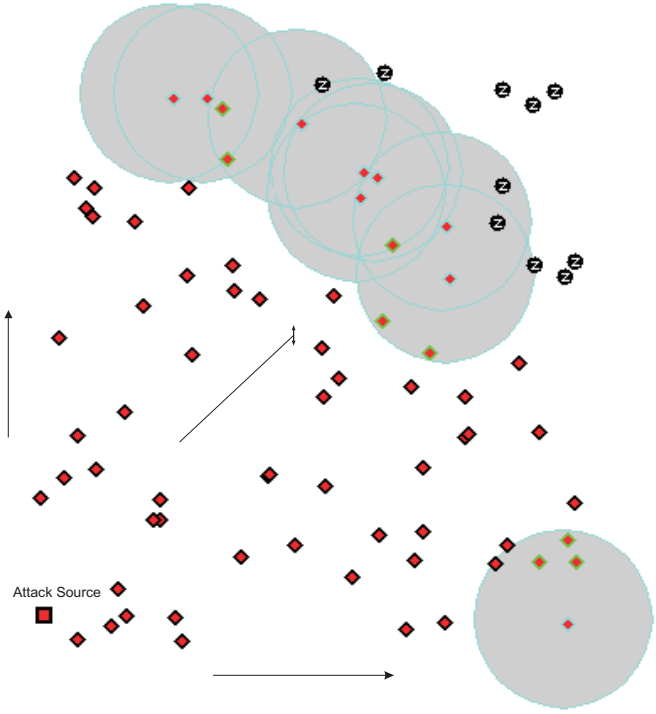

FIGURE 8. Simulated network instance where the worm propagates in waves, infecting one neighborhood after the other and many nodes in parallel. The arrows indicate the direction of these waves.

We generated random network topologies by placing 100, 200, 300, 400 and 500 nodes uniformly at random in the unit square, and selected an appropriate transmission range so that the average density d varies between 4 and 15. Our goal was to check the diffusion of the worm when different kinds of network multi-hop trees were constructed. To ensure statistical validity, we repeated each experiment 1000 times and averaged the results. We also used additional data to quantify the packet overhead induced by the routing and application layer, as described in the previous section. This configuration takes into account packet losses and retransmissions and allows us to analyze the the hop count influence to the overall propagation delay.

In each simulated network instance, we designated a node as the "attack source" who initiated the infection by first injecting the malware to its neighbors. The attacking node was distributed in one of the corners of the network. This represents the worst-case scenario for worm propagation since, as we mentioned in the previous section, the worm's infection time depends on the number of required transmissions for reaching all the sensor nodes. An illustration of such a simulation instance can be found in Figure 8. Here, the squared node indicates the "attack source" whereas all the red triangled nodes have been exposed in terms of worm infection. As we can see, the worm spreads in waves, infecting one neighborhood after the other and many nodes in parallel.

One of the clear observations from the experiments is that the propagation delay depends on the number of transmissions needed to reach the most distant nodes. Since, intermediate transmissions are determined by the number of hops required to reach the most distant nodes, we can argue that the overall propagation delay is proportional to the hop count between the "attack source" and the most distant sensor node s_{dist}, which in turn is inversely proportional to neighbor density. This implies that for higher densities the hop count between the "attack source" and node s_{dist} is smaller, and therefore the number of required intermediate transmissions is less. Figures 9(a), (b), (c), (d) and (e) depict the time needed by the worm to infect all the nodes of a network consisting of 100, 200, 300, 400 and 500 nodes respectively.

What we can infer from these figures is that the overall infection time for low density networks is relatively high whereas for higher densities it decreases dramatically. As we explained before, this is based on the observation that for low densities the number of hops, and thus the number of required intermediate transmissions in order to reach the most distant node s_{dist}, is large. Therefore, counting a small additional delay due to possible packet loss, this results in a high overall propagation delay. On the other hand, for high densities, the number of required hops in order to reach node s_{dist} is much smaller as the number of 1-hop neighbors increases. Hence the overall infection time also decreases as well. In Figure 9(f), we can see the collection of infection times for all the above described network scenarios.

In order to verify the relationship between the worm's propagation delay and the required number of hops (i.e., intermediate transmissions), we have included in Figures 9(a), (b), (c), (d) and (e) an estimate of the propagation time, denoted as *Hops*TransmTime*. This was produced by simply multiplying the mean hop count (see Figure 10) between the "attack source" and the farthest node with the mean 1-hop infection propagation time, without taking into consideration any delay added by possible packet losses. The 1-hop propagation time was found to be approximately 15 seconds on the TMote Sky devices and denotes the time needed for the worm to move from one sensor to another. This curve (shown in green in Figures 9(a)-(e)) closely matches the ones found experimentally and comes to verify our intuition that in large networks *intermediate* packet losses and retransmissions have a very small impact in the overall infection time. What matters is the number of hops between the attacker and the most distant node s_{dist} in the network.

In order to calculate the mean hop count between the attacker and s_{dist}, we conducted simulations and the results are illustrated in Figure 10. As expected the hop count is inversely proportional to the neighbor density. This implies that for sparse networks, the

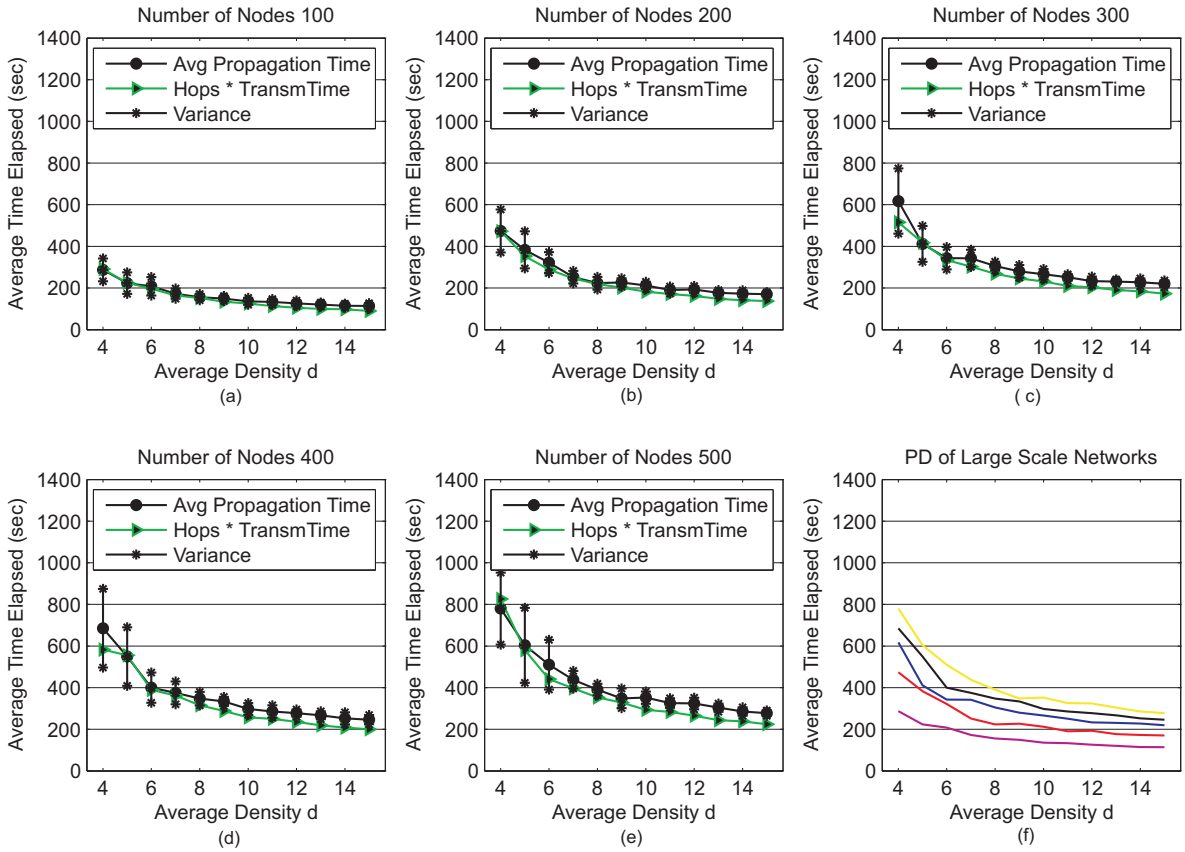

FIGURE 9. Actual average and estimation of the infection network time for varying densities and networks consisted of (a) 100 nodes (b) 200 nodes (c) 300 nodes (d) 400 nodes (e) 500 nodes. (f) Collection of mean Propagation Delays (PD)

average hop length of the target path between the "attack source" and the farthest node is relatively high. Therefore, the number of required intermediate transmissions is high, resulting in an increment in the overall worm's propagation time. On the other hand, for dense networks, the average hop length is rather low and thus the overall infection time is decreased.

Overall, the worm's propagation time is kept low even for large scale networks consisting of 500 nodes. For example, if the average density d is 8, the time needed by the worm to spread to the entire network is under 7 minutes. In any case, the infection time of the worm is relatively small, making the applicability of detection mechanisms a rather hard task to achieve.

10. DEFENSE MEASURES

A defensive mechanism against worm attacks can be the use of a diversified protection scheme, which diversifies data and code by creating different and obfuscated data and code segments for each node in the network [32]. Therefore, the attackers' effort on compromising one node cannot reduce their efforts on compromising another node, as different versions of the same functionality will not have the same vulnerability for the attacker to exploit. This approach is followed by Yang et al. [30], who showed that by assigning each sensor an appropriate version of software among a limited number of versions, the survivability of sensor networks under worm attacks is significantly increased. However, this method also restrains significantly the legitimate functionalities of sensor networks, such as network programmability, that allows nodes to reprogram themselves with new code updates disseminated remotely, over the air, to the entire network.

A different class of defensive measures is to ensure program safety at run time. For example Safe TinyOS toolchain [33] inserts checks into application code and when it detects that safety is about to be violated, it takes action and keeps errors from cascading into random consequences. In this way it ensures that array and pointer errors are caught before they can corrupt RAM. Another example is Harbor [34], which uses software-based fault isolation to enforce restrictions on memory accesses and achieve memory protection. In particular, it uses an additional safe stack to preserve the integrity of control flow within and across modules. Even though the above schemes can protect application modules from each other or themselves, an attacker can still look for vulnerabilities into system routines not included in the modules, in order to evade these schemes [14].

A reactive measure against worm dissemination can

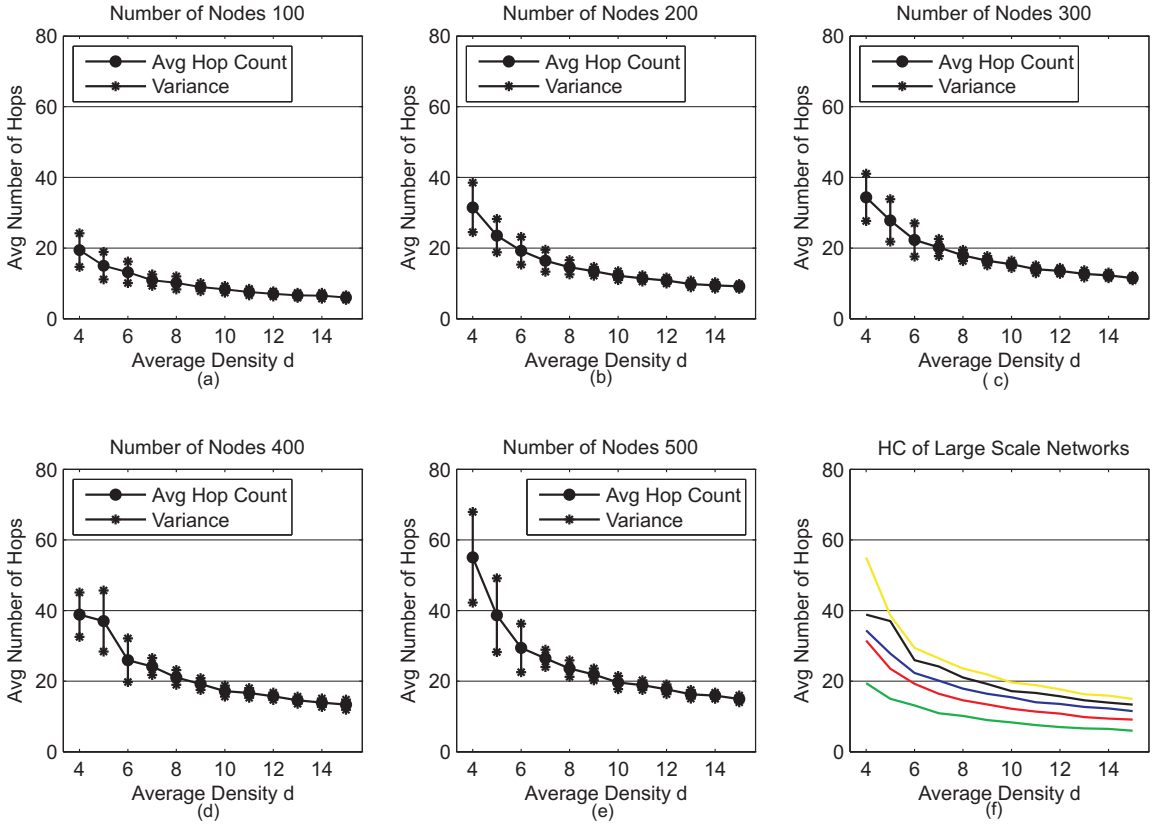

FIGURE 10. Average hop count between the "source attack" and the most distant node s_{dist} for varying densities and networks consisted of (a) 100 nodes (b) 200 nodes (c) 300 nodes (d) 400 nodes (e) 500 nodes. (f) Collection of mean Hop Counts (HC)

be software-based code attestation. For example, SWATT [35] enables an external verifier to verify the code of a running system to detect maliciously inserted or altered code, without the use of any special hardware. Yang et al. [36] took this approach one step further allowing other sensor nodes play the role of the verifier and alert the rest of the network in case an infected node is detected. A similar idea focusing on the concept of cooperative intrusion detection was proposed in [37, 38], where the nodes collaborate to detect and isolate an attacking node. To apply such a mechanism for worm detection, however, one needs to find ways for legitimate nodes to become suspicious that a worm is being propagated (and initiate the intrusion detection process) *before* they get infected as well.

11. CONCLUSIONS AND FUTURE WORK

We have presented how buffer overflow vulnerabilities can be exploited in Von Neumann architecture-based sensor nodes in order to inject arbitrary long code into the program memory of the mote. The attack can be used to add new (malicious) functionalities to sensor nodes (i.e. have the nodes report back vital information) or simply shut down the entire network.

Our attack breaks the malicious code into multiple packets and sends them through radio to the sensor node, where through a multistage buffer overflow it is permanently stored and executed. We have also described how the malicious code can be crafted such that it replicates itself after execution on the mote and propagates to the rest of the network, making it the first instance of a self-replicating worm that can execute arbitrary code as opposed to any previous work in this area. We have illustrated this attack by sending different sizes of malicious code on Tmote Sky sensors and demonstrated the feasibility of taking over the entire network one node at a time. We have also presented an evaluation of the worm's propagation time in order to show that the infection of large scale networks takes up only a short amount of time.

As future work, we plan to investigate how worm propagation is affected by different traffic patterns and packet drops, in a sense focusing on the reliability aspects of the dissemination process. We also plan to identify more vulnerabilities that can be exploited in a similar manner, as well as build efficient mechanisms to counter such attacks.

REFERENCES

[1] Becher, A., Benenson, Z., and Dornseif, M. (2006) Tampering with motes: Real-world physical attacks on wireless sensor networks. *SPC '06: Proceeding of the 3rd International Conference on Security in Pervasive Computing*, Lecture Notes in Computer Science, **3934**, pp. 104–118. Springer.

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

[3] Liu, A., Oh, Y.-H., and Ning, P. (2008) Secure and DoS-resistant code dissemination in wireless sensor networks using Seluge. *Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN 2008)*, San Francisco, California, USA, pp. 561–562.

[4] Lanigan, P. E., Gandhi, R., and Narasimhan, P. (2006) Sluice: Secure dissemination of code updates in sensor networks. 53.

[5] Dutta, P., Hui, J., Chu, D., and Culler, D. (2006) Securing the Deluge network programming system. *Proceeding of the 5th International Conference on Information Processing in Sensor Networks (IPSN 2006)*, April, pp. 326–333.

[6] Krontiris, I. and Dimitriou, T. (2006) Authenticated in-network programming for wireless sensor networks. *ADHOC-NOW*, pp. 390–403.

[7] Tmote Sky Quick Start Guide. Technical Report.

[8] Polastre, J., Szewczyk, R., and Culler, D. (2005) Telos: enabling ultra-low power wireless research. *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, Los Angeles, California 48.

[9] Handziski, V., Polastre, J., Hauer, J.-H., and Sharp, C. (2004) Flexible hardware abstraction of the TI MSP430 microcontroller in TinyOS. *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, Baltimore, MD, USA, pp. 277–278.

[10] Baar, M., Köppe, E., Liers, A., and Schiller, J. (2007) The ScatterWeb MSB-430 platform for wireless sensor networks. *Contiki Hands-On Workshop 2007*, Kista, Sweden.

[11] SHIMMER (2008). Sensing health with intelligence, modularity, mobility, and experimental reusability. `http://docs.tinyos.net/index.php/Intel_SHIMMER`.

[12] Goodspeed, T. (2008) Exploiting Wireless Sensor Networks over 802.15.4. *Texas Instruments Developper Conference*.

[13] Goodspeed, T. (San Diego, 2007) Exploiting Wireless Sensor Networks over 802.15.4. *ToorCon* 9.

[14] Gu, Q. and Noorani, R. (2008) Towards self-propagate mal-packets in sensor networks. *WiSec '08: Proceedings of the first ACM conference on Wireless network security*, Alexandria, VA, USA, pp. 172–182.

[15] Francillon, A. and Castelluccia, C. (2008) Code injection attacks on harvard-architecture devices. *15th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, USA.

[16] Smirnov and Chiueh, T. (2005) Dira: Automatic Detection, Identification and Repair of Control-Data Attacks. *Network and Distributed System Security Symposium*.

[17] Piromsopa, K. and Enbody, R. J. (2006) Defeating Buffer-Overflow Prevention Hardware. *5th Annual Workshop on Duplicating, Deconstructing, and Debunking*.

[18] Piromsopa, K. and Enbody, R. (2006) Buffer-Overflow Protection: The Theory. *Proceedings of the 6th IEEE International Conference on Electro/Information Technology*, East Lansing, Michigan.

[19] Davis, M. (2009) Recoverable Advanced Metering Infrastructure. *Black Hat*.

[20] Goodspeed, T. (2009) A 16 bit rootkit and second generaion zigbee chips. *Blakc Hat*.

[21] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. (2000) System architecture directions for networked sensors. *ACM SIGPLAN Notices*, **35**, 93–104.

[22] Platon, E. and Sei, Y. (2008) Security software engineering in wireless sensor networks. *Progress in Informatics*, **5**, 49–64.

[23] Tmote Sky Datasheet. Technical Report.

[24] Kurtis Kredo, I. and Mohapatra, P. (2007) Medium access control in wireless sensor networks. *Computer Networks*, **51**, 961–994.

[25] Texas Instruments MSP430x1xx Family User's Guide. *SLAU049B*.

[26] One, A. (1996) Smashing the stack for fun and profit. *Phrack*, **7**.

[27] Akyildiz, I. F., Su, W., Sankarasubramaniam, Y., and Cayirci, E. (2002) A survey on sensor networks. *Communications Magazine, IEEE*, **40**, 102–114.

[28] Dirk Trossen, e. a. (2007) Sensor networks, wearable computing, and healthcare applications. *IEEE Pervasive Computing*, **6**, 58–61.

[29] Ranjan Panda, P. and Dutt, N. (2002) Memory architectures for embedded systems-on-chip, . pp. 647–662.

[30] Yang, Y., Zhu, S., and Cao, G. (2008) Improving sensor network immunity under worm attacks: A software diversity approach. *MobiHoc '08: Proceedings of the 9th ACM International Symposium on Mobile Ad Hoc Networking and Computing*.

[31] Woo, A., Tong, T., and Culler, D. (2003) Taming the underlying challenges of reliable multihop routing in sensor networks. *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, New York, NY, USA, pp. 14–27. ACM.

[32] Alarifi, A. and Du, W. (2006) Diversify sensor nodes to improve resilience against node compromise. *SASN '06: Proceedings of the fourth ACM workshop on Security of ad hoc and sensor networks*, Alexandria, Virginia, USA, pp. 101–112.

[33] Cooprider, N., Archer, W., Eide, E., Gay, D., and Regehr, J. (2007) Efficient memory safety for TinyOS. *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, Sydney, Australia, pp. 205–218.

[34] Kumar, R., Kohler, E., and Srivastava, M. (2007) Harbor: software-based memory protection for sensor

nodes. *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, Cambridge, Massachusetts, USA, pp. 340–349.

[35] Seshadri, A., Perrig, A., van Doorn, L., and Khosla, P. (2004) SWATT: Software-based attestation for embedded devices. *Symposium on Security and Privacy*, Los Alamitos, CA, USA, pp. 272–282.

[36] Yang, Y., Wang, X., Zhu, S., and Cao, G. (2007) Distributed software-based attestation for node compromise detection in sensor networks. *SRDS '07: Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, Beijing, China, pp. 219–230.

[37] Krontiris, I., Dimitriou, T., and Giannetsos, T. (2008) LIDeA: A distributed lightweight intrusion detection architecture for sensor networks. *Proceeding of the 4th International Conference on Security and Privacy for Communication (SECURECOMM '08)*, Istanbul, Turkey, September.

[38] Krontiris, I., Benenson, Z., Giannetsos, T., Freiling, F. C., and Dimitriou, T. (2009) Cooperative intrusion detection in wireless sensor networks. *EWSN '09: Proceedings of the 6th European Conference on Wireless Sensor Networks*, Berlin, Heidelberg, pp. 263–278. Springer-Verlag.