ABSTRACT
Since sensors do not have a sophisticated hardware architecture or an operating system to manage code for safety, attacks injecting code to exploit memory-related vulnerabilities can present threats to sensor applications. In a sensor's simple memory architecture, injected code can alter the control flow of a sensor application to either misuse existing routines or download other malicious code to achieve attacks. To protect the control flow, this paper proposes a self-healing scheme that can detect attacks attempting to alter the control flow and then recover sensor applications to normal operations with minimum overhead. The self-healing scheme embeds diversified protection code at particular locations to enforce access control in program memory. Both the access control code and the recovery code are designed to be resilient to control flow attacks that attempt to evade the protection. Furthermore, the self-healing scheme directly processes application code at the machine instruction level, instead of performing control or data analysis on source code. The implementation and evaluation show that the self-healing scheme is lightweight in protecting sensor applications.

Categories and Subject Descriptors
D.4.6 [Operating Systems]: Security and Protection

General Terms
Security

Keywords
Sensor application, Control flow, Access control, Self-healing, TinyOS

1. INTRODUCTION
Applications in sensor networks have been researched and developed for years. However, most security work focused on threats to networking and communication protocols. Lessons learned from worm attacks that exploit memory fault vulnerabilities show that attackers can compromise an entire network without hacking legitimate accounts or breaking protocols. On the Internet, malicious attackers often break into computer systems by exploiting vulnerabilities arising from low-level memory faults, e.g., stack overflow [6], format string vulnerability [26], integer overflow [33], double free [7], heap overflow [19], return-to-libc [25], etc. Such cyber attacks in regular computer systems are drawing attention to similar threats to sensor networks.

As sensors use very simple embedded systems due to cost and resource limitations, sensors do not have sophisticated operating systems (OSs) to manage code for safety. Simple OSs [3, 5] have been developed for embedded systems. However, they do not distinguish kernel mode or user mode when executing an instruction, and application data is adjacent to system data. Hence, one application routine can easily access the data of the system or other application routines. Furthermore, high-level programming languages [4] have become popular in developing sensor applications because of their convenience for coding and maintenance over assembly languages. Open source based sensor applications have been developed as well. Consequently, applications share more and more common code as they use similar development environments. Memory fault attacks based on the same principle in regular computers become threats to sensor networks.

Many defense schemes have been proposed for protecting regular computers. However, some critical issues cannot be addressed by simply applying these schemes in resource-limited sensors. First, sensors do not have an architecture to effectively enforce access control in program memory. A few schemes have been proposed to enforce access control in a sensor's data memory by using software-based memory management [29, 23]. These approaches do not prevent exploiting packets from accessing other code segments in the same program memory. Second, sensors do not have an effective recovery mechanism. Illegally accessing instructions in program memory normally causes the crash of the running sensor applications and results in a long restart period. In a regular computer, such a crash can be detected by page fault and then an interrupt will be invoked to recover the
computer. However, these kinds of recovery mechanisms are not present in sensors. Sensors do not have a mechanism to invoke interrupts corresponding to fatal system errors. Neither are they easily administrated after being deployed. To address these issues, this paper presents a self-healing scheme for detecting attacks altering control flow and recovering sensors from compromised tasks. The features of the self-healing scheme are summarized below.

- The scheme embeds randomized marks and access control code at particular locations to detect malicious control flow manipulation. The access control code effectively enforces access control in program memory such that the control flow cannot be maliciously altered. The access control code itself is designed to be resilient to control flow attacks that attempt to evade the access control.
- The scheme provides a self-healing recovery routine to quickly remove a compromised task from the application and restore the sensor to a normal state. The routine cleans up sabotaged data in data memory and releases the resources taken by the compromised task.
- The scheme works at the machine instruction level and directly processes an application’s machine code instead of the application’s source code. The scheme diversifies the protected code images for different sensors.

The remainder of the paper is organized as follows. Section 2 discusses the related work on attacks and defenses. Section 3 introduces control flow attacks and overviews the proposed self-healing scheme. Section 4 explains the access control mechanism and its security properties in detail. Section 5 describes the recovery mechanism and its security properties. Section 6 presents the implementation and evaluation of the self-healing scheme. Finally, Section 7 concludes the paper and discusses future work.

2. RELATED WORK

2.1 Memory Fault Attacks

Many computer attacks exploit vulnerabilities due to memory fault in current computer systems. Such attacks can be categorized as control flow attacks [16, 30] and data flow attacks [9, 10]. This paper focuses on control flow attacks that overwrite control data to alter control flow so that a process will execute an unexpected sequence of instructions. For instance, in a stack smashing attack [6], the return address of a vulnerable function in the current stack frame is overwritten to the address of injected code. When the function (corresponding to the current stack frame) returns, the injected code will be executed. Attackers can overwrite control data to alter control flow via exploiting vulnerabilities of format string error [26], double-free error [7], heap overflow [19], return-to-libc [25], etc. Attackers can alter control flow to execute injected malicious code or to bypass conditional branches or invoke indirect jumps. The self-healing scheme is designed to address these control flow attacks.

2.2 Defense Techniques

Many defense techniques have been proposed against control flow attacks in regular computer systems. Main ideas include preventing overwriting control data [12, 13], preventing executing injected code [31, 27], randomizing address and code [8, 24], bug detection [14, 22] and run-time detection [11, 21]. However, they cannot be simply applied in sensor nodes due to their limitations in defense or the resource limitation of sensors.

Therefore, a few techniques have been developed to provide memory safety. Safe TinyOS [29] performs complex pointer and type analysis. It treats pointers as safe, sequence or wild (the later two are unsafe), and transforms an application’s source code to code using safe pointers. Safe TinyOS also provides a software framework that has a kernel space for OS-related modules and an extension space for untrusted application modules. The framework has a set of proxies to provide kernel services to applications. Its design relies on the assumption that few typecasts will be used in a normal C program. However, actual application code does perform arbitrary typecasts, resulting in sequence and wild pointers. These unsafe pointers sometimes fail Safe TinyOS in generating safe code for certain applications.

Harbor [23] uses software based fault isolation to restrict data memory accesses. It puts application modules in different protection domains. Then, it uses a memory map manager to provide separate data memory space for each domain and ensures that a memory write does not cross memory boundaries. Furthermore, it uses a control flow manager to ensure that function calls never flow out of a domain except via calls to functions exported by the kernel or modules in other domains. Limitations of Harbor include its reliance on a kernel that is relatively well tested and free of programming errors and its checks on every memory write and control flow transfer. It cannot address all possible memory corruption faults within a domain. Finally, it needs a third party to detect corrupted system routines.

The self-healing scheme in this paper takes a different design principle. In stead of preventing illegal data memory access, the self-healing scheme enforces program memory access of sensor applications. It also has a recovery routine to handle compromised tasks such that sensors can quickly return to normal operations.

3. BACKGROUND AND OVERVIEW

3.1 Control Flow Attacks

Since sensors use a different architecture for program and data storage, it is reasonable to question what vulnerability could be exploited in sensor nodes and how attacks can be carried out. In this research, we use Atmel’s ATmega128 [1] processor as an example. The processor has a RISC core running single cycle instructions. The program memory of the processor is write-protected such that the application code can reliably work in the field. One of the attacks targeting this architecture is to alter the control flow of a sensor application [17].

Attackers can alter the control flow via many well known buffer overflow techniques [6, 26, 7, 19]. In sensor nodes, attackers could find more approaches as the sensor’s architecture is very simple. Attackers can directly overwrite kernel data or registers that are memory-mapped. In the data memory of the ATmega128, the CPU registers and the IO registers are mapped to the first 256 bytes in the data memory address space. Application code can directly access these critical addresses as a normal data access. In addition,
TinyOS running in ATmega128 does not distinguish kernel mode or user mode when executing an instruction, and user data is adjacent to system data. Hence, application code can directly access system data. These facts indicate that the defense schemes in regular computers that rely on safe memory space are under challenge in sensors.

With the possibility of exploiting vulnerabilities in sensor applications, researchers have found techniques that use non-control data carried in exploiting packets to redirect the control flow to achieve certain attacks. The attacking packets can invoke existing application routines to propagate or inject false messages in a sensor network [17]. If sensors are allowed to use networking programming [18], control flow attack techniques can enable attackers to invoke the networking programming routines to re-program the compromised sensor with a malicious code image or temporarily disable it by reloading the Golden image into it [15]. Hence, countering control flow attacks is critical to safeguard sensor applications.

3.2 Overview of Self-healing

The proposed self-healing scheme is illustrated in Figure 1 to handle control flow attacks. It has two modules: (a) access control module that enforces the control flow of a running task, and (b) recovery module that recovers the control flow of the sensor application from a compromised task. The execution of a sensor application is managed by the task scheduler of the sensor’s OS. When a sensor receives a packet, a task will be dispatched by the task scheduler to process the packet. Once the task finishes, the execution of the application will return to the task scheduler so that the next pending task can be dispatched.

The self-healing scheme embeds small blocks of access control code in all code segments in the program memory. In a normal situation (Figure 1(a)), all code segments being accessed by a task are in fact determined by the sensor application. Hence, each task has a pre-determined control flow. A non-compromised task should not have any abnormal access to a code segment that is not in its control flow. Thus, the access control code will allow the execution of any regular task.

If a packet exploits a vulnerability in the code of the running task, we consider the task to be compromised. The vulnerability of the running task in fact allows the exploiting packet to evade the access control code of the compromised task and thus redirect the control flow from the running task to a code segment in the program memory for the attack purpose. However, the re-direction will be captured by the access control code in the destination code segment, because the execution of the code segments deviates the normal control flow of the compromised task.

Then, the access control code hands over the compromised task to the recovery routine that cleans the compromised task and returns the execution to the task scheduler for the next pending task. Both the task scheduler and the recovery routine are protected with access control code to prevent attackers from exploiting them.

The self-healing scheme protects sensor applications directly at the machine instruction level in stead of the C-like source code. As we examine the machine code after compiling the source code, we notice that the actual executable code is not exactly what can be observed in the C-like source code.

First, compilation of sensor applications must have optimization turned on so that the code can be minimized. The access control code is added to the minimized executable at the machine level so that the security mechanism will not affect the optimization of a sensor application. In addition, the control flow of a sensor application is analyzed at the machine code level so that it reflects the actual execution of the application.

Second, in sensor applications, system and library functions (such as IO functions, string manipulation functions, memory management functions, etc.) are not loaded as shared libraries in the program memory. They are statically linked in sensor applications so that other functions can call them. If protection only works on the C-like source code, the security of library functions are usually not inspected. Any malicious control flow alteration to system and library functions can evade detection.

Third, to counteract control flow attacks, we diversify the access control code in all individual sensor code images. This protection mechanism needs to be conducted on the original executable code image. In addition, to separate kernel data from application data, the self-healing scheme re-organizes the data memory layout such that a compromised task will not affect the kernel. The self-healing scheme achieves this at the machine code level as well.

4. ACCESS CONTROL IN PROGRAM MEMORY

The objective of access control in program memory is to ensure that the code in program memory is executed according to the control flow. Hence, in this section, we first present the control flow graph by analyzing the executable code. Then, we discuss how to embed the access control code into the executable and how to secure the access control code so that the code itself is resilient against control flow attacks that attempt to evade the defense.

4.1 Control Flow Graph

A sensor application consists of a sequence of instructions which can be partitioned into code segments. The control flow of the sensor application can thus be modeled as the code segments and the transitions among them. The transitions are made of various branch, jump and call instructions.
We classify them into three categories (detailed instructions can be found in Appendix A).

C1: conditional transition. A transition that takes place according to a condition. With this type of instruction, the control flow will be altered to another address if the condition is satisfied.

C2: non-conditional and returnable transition. A transition that will always happen but the control flow will return to the instruction next to where the transition originates. Function calls make this category of transitions.

C3: non-conditional and non-returnable transition. A transition not belonging to the first two categories. When a transition of this category happens, the control flow will be transited to an address in another code segment and will not return to the originating address.

The C3 transitional instructions are used to partition the application code into atomic code blocks (Definition 1). The ACBs are then grouped into control nodes (Definition 2). The control flow graph of a sensor application is thus composed of CNs.

**Definition 1.** Atomic Code Block (ACB). An atomic code block is a block of consecutive instructions that has only one C3 transitional instruction at the end of the block.

**Definition 2.** Control Node (CN). A control node is a set of ACBs connected via internal transitions.

Internal transitions are transitions among ACBs inside a CN, and external transitions are transitions among CNs. An example of the control flow graph is shown in Figure 2, where the application code is partitioned into 8 ACBs. Their layout in program memory and their transitions are shown in Figure 2(a). Two CNs are formed as in Figure 2(b). One has five gray ACBs and the other has three white ACBs.

The strategies of making CNs from ACBs may vary according to the security requirement of the application. The self-healing scheme prevents attacks based on the occurrence of abnormal transitions among CNs, but not within CNs. Hence, the protection requirement of the application determines the granularity of the CNs. However, the analysis in Section 4.3 shows that more CNs do not necessarily provide stronger security, but increase the overhead of the self-healing scheme.

This paper groups all ACBs of the same function as one CN. In other words, a CN is equivalent to a function or an interrupt this paper. This strategy allows the self-healing scheme to capture abnormal transitions among functions. Therefore, with this strategy, the transitions between the two CNs in Figure 2 correspond to function calls and returns. Using this strategy is sufficient to illustrate the security features of the self-healing scheme.

### 4.2 Access Control Algorithms

After obtaining CNs in a flow graph, the self-healing scheme assigns a mark (CNMK) to each CN and enforces access control for protecting transitions among CNs. Two access control algorithms are designed for non-returnable and returnable transitions respectively. The access control algorithms only allow outgoing transitions originated from a CN that the control flow legally got in previously. If attackers alter the control flow to get into a CN, then any outgoing transition from the CN will be denied by the access control code.

The algorithm for non-returnable transitions is shown in Algorithm 1. Rx and Ry are two temporary variables used in the algorithm. Before a transition takes place, the CNMK of the current CN is always stored in Rx. When a transition happens, the algorithm first moves the CNMK stored in Rx into Ry and loads the CNMK of the destination CN into Rx. Then, the algorithm checks if the CNMK in Ry is really the mark of the current CN. If so, the transition is allowed. Otherwise, the self-healing scheme invokes the recovery routine, which will be discussed in Section 5.

Algorithm 2 for returnable transitions extends the idea of Algorithm 1. For a call instruction, Algorithm 2 saves Rx before calling, uses Algorithm 1 to verify the validity of the call, and restores Rx after the call returns. For a return instruction, Algorithm 2 checks if the current CNMK matches the CNMK stored in Rx before returning to the calling function.

To secure the access control code and to ensure that no two sensor nodes have the same code, the self-healing scheme uses a crypto-random function to generate random CNMKs and assigns them to all sensors. Therefore, each sensor will be loaded with a unique code image with randomized CNMKs. The diversification of executables can better protect a sensor network even when some sensors are compromised [32]. Note that since CNMKs are generated, they are not modifiable because they are immediate operands of instructions that are stored in the write-protected program memory.

Sensors may use an indirect jump or call under complicated situations. For example, TinyOS needs indirect transitions in its scheduler. Because the new CNMK loaded into Rx after transition is determined by the destination CN, the self-healing scheme cannot just load a fixed CNMK into Rx.

Thereby, variants of the two access control algorithms are developed for indirect transitions. In stead of replacing an indirect transition, the variants create an intermediate CN for each destination CN. When an indirect transition happens, the variants first redirect the indirect transition from the originating CN to the intermediate CN, and then to the destination CN, if the transitions are al-
Algorithm 1 Pseudo access control code for a non-returnable transition from CN$_A$ to CN$_B$.

The following code replaces the transitional instruction in CN$_A$. CNMK$_A$ is supposed to be in Rx before the execution of the detection routine if no attack.

1: Move the value in Rx to Ry
2: Store CNMK$_B$ in Rx
3: if $Ry \neq$ CNMK$_A$ then
4: Invoke the recovery routine
5: end if
6: Transit to CN$_B$

Algorithm 2 Pseudo access control code for a returnable transition from CN$_A$ to CN$_B$

The following code replaces the call instruction in CN$_A$. CNMK$_A$ is supposed to be in Rx before the execution of the detection routine if no attack.

1: Push Rx
2: Move the value in Rx to Ry
3: Store CNMK$_B$ in Rx
4: if $Ry \neq$ CNMK$_A$ then
5: Invoke the recovery routine
6: end if
7: Call to CN$_B$
8: Pop Rx

The following code replaces the return instruction in CN$_B$. CNMK$_B$ is supposed to be in Rx before the execution of the detection routine if no attack.

1: if $Rx \neq$ CNMK$_B$ then
2: Invoke the recovery routine
3: end if
4: Return

The detection routine if no attack.

The following code replaces the return instruction in CN$_A$. CNMK$_A$ is supposed to be in Rx before the execution of the detection routine if no attack.

1: Move the value in Rx to Ry
2: Store CNMK$_B$ in Rx
3: if $Ry \neq$ CNMK$_A$ then
4: Invoke the recovery routine
5: end if
6: Transit to CN$_B$

The following code replaces the call instruction in CN$_A$. CNMK$_A$ is supposed to be in Rx before the execution of the detection routine if no attack.

1: Push Rx
2: Move the value in Rx to Ry
3: Store CNMK$_B$ in Rx
4: if $Ry \neq$ CNMK$_A$ then
5: Invoke the recovery routine
6: end if
7: Call to CN$_B$
8: Pop Rx

The following code replaces the return instruction in CN$_B$. CNMK$_B$ is supposed to be in Rx before the execution of the detection routine if no attack.

1: if $Rx \neq$ CNMK$_B$ then
2: Invoke the recovery routine
3: end if
4: Return

followed. In each intermediate CN, the CNMK corresponding to one destination CN is loaded into Rx, and the transition is checked as well.

The interrupt routines raise another problem of using CNMKs in access control. First, all interrupt routines are modeled as normal CNs with the CNMKs and the access control code. Because an interrupt could happen at any time, it is not possible for a normal CN to set the valid mark for the upcoming interrupt in Rx. Hence, the interrupt will be treated as an abnormal transition. To solve the problem, the self-healing scheme adds a special handler to the interrupt vector, which is at the entrance of the interrupt routines. The handler initializes Rx with the CNMK of the upcoming interrupt before the interrupt routine is executed.

4.3 Security Analysis

For security analysis, assume attackers cannot compute correct CNMKs without knowing the key used in the self-healing scheme’s crypto-random function. The assumption is reasonable in that the keys are only used and the CNMKs are only computed offline when the diversified executables are generated. Hence, the access control algorithms have the following security properties.

**Theorem 1.** Assume the control flow has been altered from CN$_A$ to CN$_B$. Algorithms 1 and 2 will deny the attack when the control flow departs CN$_B$, if the attacker cannot provide the valid CNMK$_B$ in Rx before the control flow was altered from CN$_A$ to CN$_B$.

**Proof.** In Figure 3, an attacker has altered the control flow from CN$_A$ to CN$_B$, and then the control flow departs from CN$_B$ to CN$_C$. There are four scenarios showing how Algorithm 1 works.

**S1:** The control flow is altered to any address at or before address $(1)$ in CN$_A$. Apparently, because the attacker does not provide a valid CNMK$_B$ in Ry, the altered control flow will be detected at address $(1)$ and then the recovery routine will be invoked.

**S2:** The control flow is altered to address $(2)$ in CN$_B$. The recovery routine will be invoked, which is exactly what the self-healing scheme needs to do.

**S3:** The control flow is altered to address $(3)$ in CN$_B$. The flow will directly transit to CN$_B$ without executing any instruction in CN$_B$. This scenario is equivalent to that control flow is altered from CN$_A$ to CN$_C$. The altered control flow will be denied in CN$_C$ as in S1 or S2.

**S4:** The control flow is altered to an address after $(3)$ in CN$_B$. The attack will be captured at the next outgoing transition in CN$_B$ as in S1, S2 or S3.

Hence, Algorithm 1 denies the attack when the control flow departs CN$_B$. Similarly, Algorithm 2 can be proved to deny the attack. $\square$

Note that it is hard for the access control algorithms to capture the malicious transition when the control flow gets into CN$_B$, because the control flow may be altered to any address in CN$_B$. Thereby, Algorithms 1 and 2 are designed to capture the attack when the altered control flow departs from CN$_B$. In addition, if the instruction at $(0)$ is after the instruction at address $(1)$, attackers can simply alter the control flow to address $(0)$ to avoid the comparison and thus evade the access control. Hence, comparison of Ry with CNMK$_B$ must happen after loading CNMK$_C$ into Rx.

**Theorem 2.** Assume all CNs are assigned with random CNMKs. The probability that an attacker can evade the access control algorithms is $\frac{1}{2^{|CNMK|}}$, where $|CNMK|$ is the size of a CNMK (the number of bits of a CNMK).

**Proof.** Assume that an exploiting packet can overwrite Rx with CNMK$_B'$ before the packet alters the control flow from CN$_A$ to CN$_B$. The access control in CN$_B$ can be evaded only if the CNMK$_B'$ matches the real CNMK$_B$. Because the attacker does not have the valid CNMK$_B$ for CN$_B$, the probability of guessing the valid CNMK$_B$ is $\frac{1}{2^{|CNMK|}}$. $\square$

In this paper, the self-healing scheme uses 8-bit CNMKs. The evasion probability is $\frac{1}{256}$. Note that the self-healing scheme does not affect control flow and thus will not raise false positive. Therefore, the security of the self-healing scheme is measured by the false negative rate, i.e., the possibility that an attacker can invoke an unexpected routine and evade the access control.

**Theorem 3.** Let $|CNMK|$ be the size of a CNMK. The application code in a sensor has at most $2^{|CNMK|}$ effective CNs.
The limited size of CNMK (for example, 8 bits) affects the security of the access control code inside a sensor. When the control flow is complex, it would be easy to find more than 256 CNs in one sensor. Hence, multiple CNs could possibly have the same CNMKs. They effectively function as one CN, as maliciously altering control flow among them cannot be detected. Thereby, the self-healing scheme groups CNs into at most 256 CNs in one sensor. The self-healing scheme can double the size of CNMK to 16 bits for accommodating multiple CNs to execute after the current running task is finished. Such an OS is normally migrated from the non-preemptive OS in sensors.

Theorem 4. Interrupt routines can only be invoked via their entrances in the interrupt vector.

Proof. An exploiting packet can alter the control flow either directly to an interrupt routine or to the entrance of an interrupt in the interrupt vector. Because all interrupt routines are protected as normal CNs, the first alteration approach will be denied in the interrupt due to the same reason approved in Theorem 1. Therefore, in order to invoke an interrupt, an exploiting packet has to redirect the control flow to the entrance of the interrupt in the interrupt vector.

Although interrupt routines can be invoked by an exploiting packet, current design of interrupt routines leaves attackers limited space for exploitation. Interrupt routines process IO data and post tasks into the scheduler’s queue in TinyOS. Attackers may use the routines to (1) send false information to other sensors or (2) post additional tasks in the compromised sensor, which are also the functions provided to normal sensor applications. The first scenario can be prevented via deploying link or network layer security protocols [20, 28]. These protocols are handled as normal application routines, and interrupt routines simply process data generated by these protocols. Hence, interrupt routines cannot help attackers forge authenticated information. The second scenario may affect the performance of sensors due to unnecessary tasks being posted. TinyOS v2 allows up to 255 tasks simultaneously. Many sensor applications do not need so many tasks. If attackers post a task unspecified in applications, the task will simply be discarded.

5. Recovery of a Compromised Task

As kernel routines are very crucial in a system, restarting the whole system is the ideal, safe and straightforward response to eliminate kernel attacks in sensors. Hence, in this paper, we focus on recovering the system when application tasks are being exploited. Because it is possible that an exploited function may affect other functions of the same task, the recovery is task-based.

The general recovery procedure includes three steps as illustrated in Algorithm 3. The recovery first releases resources allocated to the compromised task, then releases the compromised task from the kernel, and finally guides the kernel to execute the next pending task. In this section, we first discuss the idea of the recovery approach normally used in preemptive OSs and then the recovery approach for the non-preemptive OS in sensors.

5.1 Recovery in Preemptive OSs

A preemptive OS allows a task to be swapped out before it is finished. Such an OS is normally migrated from regular OSs or designed according to the similar principles of regular OSs. For example, most Linux systems running on embedded systems are preemptive. Some more capable sensors may use these kinds of OSs.

To support preemption, the OS normally uses memory pages to organize the memory and assign memory pages to the running tasks. The memory pages are supported by the CPUs having memory management units (MMUs), and the kernel keeps a list of memory pages allocated to the running tasks. A task is not allowed to access a memory page of another task. If a task tries to access a memory page of another task, a fault will be raised by the kernel. Hence, we can assume the damage in task memory is well confined in the task scope, given that the kernel is not compromised. Apparently, a preemptive OS can easily heal itself because damage is confined.

As in Algorithm 3, the self-healing scheme first releases all memory pages and other resources allocated to the compromised task. Then, the recovery routine needs to prevent potential future damage caused by the compromised task. Hence, the recovery routine needs to repair the list of memory pages by removing all entries of the compromised task. The recovery routine also needs to inspect any shared resource being locked by the compromised task and release the lock. Finally, the compromised device should continue its operation. So, the recovery routine should guide the kernel to execute the next pending task.

5.2 Recovery in Non-preemptive OSs

A non-preemptive OS only allows the next task to be executed after the current running task is finished. Such OSs are likely to be used in sensor applications that are not task intensive. In a sensor network, when a sensor detects an event, it wakes up and completes a few tasks to collect information and send information back. Sensors stay in sleep status to save energy most of the time. The tasks usually
do not require intensive computation. Therefore, in such applications, preemption is not necessary as most tasks are finished in a short period.

The processor in a sensor is designed with a simple memory structure and a simple memory management mechanism. For example, the ATMega128 processor used in MICA sensors has a very simple memory structure in which the kernel memory and the user memory share the same address space. The processor only supports a very simple scheme for memory page management. Such scheme is far from sufficient to support preemption. Due to the simple architecture, it is more challenging to recover a sensor from a compromised task in a non-preemptive OS.

In the following, we propose a recovery approach that first restructures the data memory layout to separate the data used by the kernel and the task, and then takes proper operations to recover the sensor application.

### 5.2.1 Restructure of Data Memory Layout

It is observed that tasks in a non-preemptive OS are in fact executed in first-in-first-out order. Thereby, the control flow of task execution can be modeled as in Figure 4(a). First, a task is dispatched by the kernel. At some point in the task’s execution, the task requests a system service from the kernel. For example, the task may request the kernel to complete an IO operation or invoke a system function.

Accordingly, the data memory has three blocks as shown in Figure 4(b). The first block is the stack and heap of the kernel before the kernel dispatches the task. The second block is the stack and heap of the task before the task requests a system service. The third block is the stack and heap of the service routine. The task’s memory is just adjacent to the kernel’s memory and the service’s memory. It is possible that a compromised task sabotages the kernel’s memory and the service’s memory without being detected. Simply following the recovery procedure in preemptive OSs cannot ensure that the kernel’s memory is clean after removing a compromised task.

Therefore, to make recovery feasible in sensors, we need to separate the memory used by the kernel from the task. Based on the unique control flow of non-preemptive OSs, we propose two approaches to restructure the data memory layout as shown in Figure 5 so that the memories of the kernel and the running task can be held in a single memory without being intertwined.

In Figure 5(a), the kernel-based approach is used. The data memory is partitioned to two memory pages. One memory page (in gray) holds the kernel-related memory, i.e., the stack and the heap of the kernel and the service routines. The other (in white) holds non-kernel-related memory, i.e., the memory of the running task. The memory of the system service routine is held in the kernel’s memory page, as the service routine is not application specific. Following the control flow, the stack and the heap of the service routine is within the stack and the heap of the kernel.

Although the memory is divided, the kernel-based approach has two problems. One is that the size of the kernel-related memory varies. A service routine could possibly request a big chunk of memory which could go beyond the limit of the pre-allocated kernel-related memory. The second is that the data in the kernel-related memory is affected by the running task. If the running task is compromised, the sabotaged data in the kernel-related memory should also be cleaned.

To address the two problems raised by the kernel-based restructure approach, we propose the task-based approach as shown in Figure 5(b). We can observe that the system’s memory is only changed when a task is running. Therefore, we divide the system’s memory into two parts. One part is the memory block when no task is running. The other part is the memory when a task is running. Because only one task is running at any time in a non-preemptive OS, the second part is only corresponding to the current running task and will not be affected by any other task. Hence, the memory layout can be restructured as task-related and non-task-related as in Figure 5(b). The task-related memory includes the task’s memory and the service routine’s memory. The non-task-related memory is the kernel’s memory before running any task.

In order to restructure the data memory layout, the executable code will be analyzed to identify all non-task-related memory blocks offline before deploying the sensor application. The code analysis will modify the kernel such that (a) the kernel is initialized in the given non-task-related memory and (b) when the kernel is ready to dispatch tasks, all memory-related variables will be adjusted to point to the task-related memory. The code analysis will also modify the interrupt vector such that all interrupt routines will only use the task-related memory for their temporary data.

### 5.2.2 Recovery Procedure

The recovery routine in a non-preemptive OS also follows
Algorithm 3. When a compromised task is captured by the access control code, the recovery routine will first release the task-related memory of the compromised task. As the data memory has been restructured, simply removing the task-related memory will clean all sabotaged data of the compromised task. The recovery approach also releases the compromised task from any IO resource (such as a radio communication device) by setting the resource to its initial state. The resources that could be taken by a running task can be identified by analyzing the IO instructions of the task code offline before deploying the sensor application.

Then, the recovery routine removes the compromised task from the task queue in the kernel. Because the kernel memory is separated from the task memory, it is hard for attackers to manipulate the task queue in the kernel. Hence, the recovery routine can find the current running task in the task queue and remove it.

Finally, the recovery routine redirects the kernel to execute the next pending task. The recovery routine will reposition the stack pointer to the bottom of the task-related memory and restructure the heap to start from the top of the task-related memory. Then, the recovery routine hands over the control to the task scheduler as if the compromised task did not happen.

Note that a compromised task could be detected when an exploiting packet uses the task to exploit a vulnerability in a service routine. Hence, the self-healing scheme protects sensor applications from not only potentially vulnerable application routines but also system routines.

5.2.3 Security Analysis

As shown in Figure 6, the recovery routine also has a piece of access control code at its exit. Similar to the interrupt routines, the recovery routine has a special handler at its entrance such that the recovery routine can only be invoked from the entrance.

**Theorem 5**. The exploitation of the recovery routine results in the removal of the running task.

**Proof.** An exploiting packet can exploit the recovery routine in two manners.

**C1 (Figure 6(a))**: The control flow is altered to the entrance of the recovery routine. The recovery routine will remove the running task from the task queue in the kernel.

**C2 (Figure 6(b))**: The control flow is altered to somewhere inside the recovery routine. The alteration is an illegal access to the recovery routine and thus the detection code will be triggered to remove the running task.

Therefore, exploiting the recovery routine will always result in the removal of the running task.

6. IMPLEMENTATION AND EVALUATION

In this section, we present the implementation of the self-healing scheme in TinyOS and evaluate the overhead and performance of the scheme.

6.1 Implementation

The self-healing scheme is implemented in MICA2 motes running TinyOS v2\(^1\). The implementation of the self-healing scheme consists of three components.

- **Control flow analysis component.** It identifies CNs that include the code of interrupt routines, TinyOS routines and application routines. It also identifies and restructures the data memory layout with task-related memory and non-task-related memory.

- **Recovery code insertion component.** It appends the recovery routine to the original application code. It also fills NOPs to all empty addresses in the program memory.

- **Access control code insertion component.** It assigns a random mark to each CN and inserts the access control code to enforce access control in program memory. The safety of the access control code is based on the fact that both marks and code are stored in the write-protected program memory and cannot be modified.

The three components work together as in Figure 7. The self-healing scheme first reads the unprotected code image of a sensor application. Then, the code image is analyzed by the control flow analysis component to obtain information for further protection. The recovery code insertion component generates the recovery routine and attaches it to the original code. Finally, the access control code insertion component safeguards the unprotected code and also diversifies the protection code to ensure that each individual sensor obtains a unique protected code image.

\(^1\)The demonstration can be downloaded from http://www.cs.txstate.edu/~qg11/download.htm.
According to Algorithms 1 and 2, the self-healing scheme needs two temporary variables (Rx and Ry) for access control. We studied two possible implementations of access control: register-based access control and memory-based access control. The register-based implementation reserves two dedicated registers for Rx and Ry and the memory-based implementation reserves two dedicated data memory addresses. Our study shows that the memory-based implementation is more lightweight than the memory-based implementation. However, not all sensor processors can afford two dedicated registers. Hence, the memory-based implementation is more flexible.

6.2 Evaluation

This section examines the overhead of the self-healing scheme in program memory and how much it affects the execution of normal routines. For comparison purposes, we also implemented StackGuard [12] in TinyOS. StackGuard puts a canary in a function stack to prevent stack smashing. It adds code to set a canary when a function is called and checks the canary when the function returns.

6.2.1 Control Flow Analysis

Table 1 shows the control flow analysis of applications provided in the package of TinyOS v2. Because our implementation makes each function and each interrupt as one CN, the transitions among CNs are thus function calls. The table shows that for larger applications that have more than 100 calls, each CN has about 3.26 transitions to other CNs.

6.2.2 Evaluation of Access Control

Code Size.

The size of the access control code is determined by the number of transitions among CNs, because each transition is protected with a piece of the access control code. As discussed in Section 4, the self-healing scheme applies two access control algorithms for different types of transitions. Table 2 shows the comparison of the code sizes of sensor applications with and without protection. Table 2 shows that StackGuard adds 10.7% extra code to the original application code.

Compared with StackGuard, the memory-based access control adds 39.4% extra code to the original application code, while the register-based access control adds 31.7%. Both implementations incur more overhead than StackGuard. However, StackGuard actually only checks the control flow when a function returns. The self-healing scheme ensures that all transitions among CNs follow the control flow of the application.

The data also shows that the memory-based access control has more overhead. The sensor processor has the RISC architecture that does not allow memory-to-memory access but only allows register-to-memory or register-to-register access. Hence, the memory-based access control implementation needs more instructions when swapping CNMKs in Rx and Ry in the data memory. The register-based access control avoids this problem.

Computation Time.

The computation time is measured as CPU cycles. The computation time incurred by the self-healing scheme is corresponding to not only more computation in the sensor processor, but also more power consumption. The CPU cycles are measured in AVR Studio [2], which is a simulator provided by the processor’s manufacturer. Both the unprotected code and the protected code of the sensor applications are tested in AVR Studio. The computational overhead of the self-healing scheme is mainly determined by the number of transitions, because each transition involves the access control code for verification. Table 3 summarizes the CPU cycles at each occurrence of transitions.

We measure the CPU cycles for two routines. One is the RESET routine which initializes a sensor, and the other is an arbitrary task running in the sensor applications. Table 4 summarizes the comparison of the CPU cycles of the RESET routine and the comparison of the CPU cycles of a task with and without protection.

For the RESET routine, StackGuard takes 0.2% more time than the original RESET, the memory-based self-healing takes 1.0% more time, and the register-based self-healing takes 3.7% more time. Because the unprotected RESET routine has many instructions, the percentage increase of the routine’s computational overhead is small.

For the tasks, StackGuard takes 6.6% more time than the
original tasks, the memory-based self-healing takes 28.2% more time, and the register-based self-healing takes 8.1% more time. Since sensor processors are energy efficient in the sleep status, the CPU cycles of a running task (i.e., duty cycles) reflect the actual sensor performance. Because the tasks themselves only have a few cycles, the percentage increase of the task’s computational overhead is larger than that of the RESET routine.

Table 4 also shows that the register-based protection is much more lightweight than the memory-based protection for tasks. In contrast, the register-based protection is a little more demanding for the RESET routine. We select two registers that are the least used according to the compiler’s compilation rule. Then, we modify the executables to avoid the usage of the two registers in the register-based protection. Such modification adds much more overhead to the RESET routine than to the tasks, because the original RESET routine is directly made in assembly and frequently uses the two registers. On the contrary, the tasks seldom use the two registers, because their codes are generated from their source codes by the compiler.

6.2.3 Evaluation of Recovery

Code Size.

The overhead of the recovery routine is not affected by the number of transitions and the number of CNs in an application. In TinyOS, the recovery routine takes 37 words of code in program memory to restore stack and heap. For releasing the resources taken by a compromised task, it needs to reset the status of the resources. The operations are mainly to change the status flag of each resource in the data memory, which requires a single memory access. For example, releasing the radio taken by a compromised task needs 3 words of code to restore the radio’s ready flag.

Computation Time.

The recovery routine in the self-healing scheme is quite lightweight. It takes 38 CPU cycles to release memory of a compromised task and 3 CPU cycles for each resource. Compared with the RESET routine that normally needs more than 150K CPU cycles to restart a crashed sensor, the recovery routine can quickly help the kernel regain control of the sensor application.

7. CONCLUSION AND FUTURE WORK

This paper proposed a self-healing scheme that enforces access control in the control flow of sensor applications and recovers the sensor application from compromised tasks when a control flow attack is captured. The scheme embeds randomized marks and access control code at particular locations to detect malicious control flow manipulation. The recovery routine can quickly remove a compromised task from the application and restore the sensor to a normal state. The routine cleans up sabotaged data in data memory and releases the resources taken by the compromised task. The scheme works at the machine instruction level and directly processes an application’s machine code instead of the application’s source code. The scheme diversifies the protected code images for different sensors. The security analysis shows that the scheme itself is resilient to control flow attacks. The performance evaluation shows that the scheme is lightweight comparable to StackGuard in sensor applications.

In the future, we will study stateful recovery approaches. The current self-healing scheme uses a stateless recovery approach that simply releases memory and resources taken by a compromised task. Good tasks may be affected by such releases, because they may assume and thus be waiting for the results to be generated by the compromised task. Removing the compromised task could cause the related good tasks to obtain no result. How this affects the actual application will be studied. Correspondingly, we will study strategies to handle such situation.

8. ACKNOWLEDGMENTS

This work was partially supported by the Research Enhancement Program at Texas State University-San Marcos and the National Science Foundation under grant CNS-0423386.

9. REFERENCES


APPENDIX

A. TRANSITIONAL INSTRUCTIONS

There are 31 transitional instructions in total for the ATMega128 processor that include branch, jump and call. They can be categorized as in Table 5.

<table>
<thead>
<tr>
<th>Transitions</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional</td>
<td>BRBS, BRBC, BREQ, BRNE, BRCX, BRCC, BRSH, BRLO, BRMI, BRPL, BRGE, BRLT, BRHS, BRHC, BRTS, BRTC, BRVS, BRVC, CPSE, SBRC, SBR, SBSC, SBIS,</td>
</tr>
<tr>
<td>Non-conditional and returnable</td>
<td>RCALL, CALL, ICALL</td>
</tr>
<tr>
<td>Non-conditional and non-returnable</td>
<td>RJMP, JMP, RET, RETI, LIMP</td>
</tr>
</tbody>
</table>

Table 5: Transitional Instructions