An Error Detection and Recovery Scheme for Dynamically Downloadable Modules in Wireless Sensor Networks

Shian-Tai Chiou Hsung-Pin Chang
Department of Computer Science,
National Chung Hsing University, Taichung, Taiwan, R.O.C.
s9556045@cs.nchu.edu.tw hpchang@cs.nchu.edu.tw

Abstract*- Some sensor network operating systems support network reprogramming based on the dynamically loadable modules. Nevertheless, sensor nodes are resource-constrained and may not have the memory management unit. As a result, preventing memory access errors by the downloadable modules from crashing the sensor nodes pose a challenge in the design of wireless sensor networks. In this paper, we propose a software-based memory protection scheme that not only detects memory access faults but can also recovery from the faults.*

Our fault detection scheme is based on the idea of sandboxing. Each application module can only access regions of memory that it resides or is granted. A fault is detected when a module tries to access an invalid memory region. The fault recovery scheme is based on the idea of "n-version programming". When an error is detected in a module, we recover from the error by replacing the buggy module by another version having the same functionality of the original module. We have implemented our memory protection scheme on the SOS operating system on the Mica2 mote. According to the experimental results, our proposed scheme consumes less memory overhead under the sequential execution, which is the most prevalent execution pattern.

Keywords: Software Fault Isolation, Memory Protection, SOS, Sensor Networks.

1. Introduction

A wireless sensor network (WSN) consists of hundreds or thousands of sensor nodes that self-organize into a multi-hop wireless network. The basic building block of a sensor node includes a microprocessor, memory, RF transceiver, battery and sensor modules. Due to the huge amounts of sensor nodes, the cost of a sensor node must be low. As a result, the sensor nodes are usually resource-constrained and have a very simple architecture. Thus, computer architectures, such as memory management units (MMU) and dual modes execution, that are common in desktop computers do not appeared in sensor nodes. As a result, all programs running on a node can access the entire physical memory since they share the same single address space.

This problem is getting serious since some sensor network operating systems supports network reprogramming based on *dynamically loadable modules*. As a result, an ill-behaved program or an accidental programming error would access the memory belonging to other modules or the kernel, and at worst, crash the system.

There have been many approaches addressed this problem by using the software-based approaches to provide memory protection in WSNs. One trivial approach is to use the interpreters such as Mate [7] that provide a safe environment to execute high-level application scripts. Another approach such as Virgil [9] is to use the type-safe language that can flag illegal memory accessed at compiler time or run-time. Sandboxing, which is also called software-based fault isolation (SFI), that rewrites machine instructions is also a common method to enforce restrictions on memory accesses. For example, Harbor is the well-known sandboxing scheme in WSNs [6]. In fact, all these approaches have been seen previously in the researches of extensible operating systems. For example, "sandboxing", which is also called, is first proposed in [10].

In this paper, we also based on the idea of sandboxing. Firstly, we offload some of the checks at off-line time to eliminate the run-time checking overhead. Furthermore, in Harbor, only three memory access instructions are checked: st(store), return, and icall (indirect call). In this paper, we include the ld (load) instruction for checking to improve the safety. Finally, we complete previous memory protection scheme by implementing a fault recovery mechanism. Once an invalid memory access is detected, we identify the buggy module and dynamically download another version of the module having the same function to replace the buggy one. This technique, first proposed by Avizenis [2], is known as *n*-version programming.

We have implemented our system in the Mica2 Motes [3] on the SOS kernel [5]. According to the experimental results, our proposed scheme our scheme consumes less memory overhead under the sequential execution, which is the most prevalent execution pattern. Notably, the techniques proposed in this paper could also be applied into other systems.

The rest of the paper is organized as follows. Section 2 describes the previous work on software-based memory protection schemes in WSNs. Section 3 presents the design and implementation of our memory protection and recovery schemes. The experiment results are shown in Section 4. Finally, Section 5 gives conclusions and future work.

2. Background

Since our work is based on the SOS operating system, thus we introduce the SOS in Section 2.1. Then, previous work on the software-based memory protection schemes is shown in Section 2.2.

2.1. SOS Operating System

SOS consists of a common kernel and a set of dynamic application modules. The common kernel includes scheduling, messaging passing, dynamic memory management, and module management. An application module that implements a specific task or function can be dynamically loaded or unloaded at run time.

Interaction between application modules can be achieved by either passing messages to a module or by calling the exported functions of a module. Message passing is asynchronous in that all messages are queued and scheduled by the kernel scheduler. By contrast, function invocations are synchronous. A module can choose which of its functions to export by explicitly registering these functions to the SOS kernel. The kernel keeps track of all the exported functions of a module in the module's function control block (FCB). Before a module can call an exported function provided by another module, it has to subscribe to the SOS kernel. If the subscription succeeds, the kernel returns a pointer to the function pointer of the subscribed function, which can be de-referenced by the subscriber to invoke the function. Thus, function calls between application modules are possible only after the function registration and subscription procedures.

An application module can also request kernel services. Nevertheless, each system call must go through a system jump table. SOS adopts the jump table to allow application modules to be loosely coupled to the kernel. Thus, when an SOS kernel needs to be upgraded, it does not require all SOS application modules to be recompiled, assuming the structure of the jump table unchanged

2.2. Related Work

Software-based memory protection scheme can be achieved by a variety of ways. For example, Virgil uses the type-safe language to catch potential errors at compiler time or run time [9]. Nevertheless, most software is currently written in C, which is an unsafe language. Another scheme is to use interpreters or virtual machines (VM), which can enforce a VM controlled memory such as checking stack bound during the interpretation of higher-level languages [4, 7]. Nevertheless, similar to the type-safe language approach, applications need to be programmed using another new language. Furthermore, running applications on the VM results in an extremely high overhead due to the interpretation cost.

Harbor adopted another design path that uses the idea of sandboxing and is based on the SOS. In Harbor, the unit of protection is a protection domain. In other words, a module in one protection domain cannot access another protection domain, except via calls to functions exported by the kernel or modules in other modules. The architecture of Harbor is shown below.

Firstly, a compiled binary is rewritten by a *binary rewriter* that inserts run-time checks to sandbox them in the desktop system. Thus, the output of a binary rewriter is a sandboxed binary, which is then distributed to a network of sensor nodes. After receiving the sandboxed binary, a *binary verifier* running on each sensor node verifies that the incoming binary is correctly sandboxed or not. If the verification is successful, the incoming binary is then admitted for execution. During execution lifetime of the binary, a *flow control manager* and a *memory map manager* supervise the execution and catch potential memory access errors. Following, we briefly introduce the four components used in the Harbor: binary rewriter, binary verifier, flow control manager and memory map manager.

The memory map manager is used to provide fine-grained memory protection scheme. It maintains a memory map data structure that keeps track of fine-grained ownership and layout information for the entire address space. The control flow manager ensures that the control can only be transferred to other domains via function calls exported by the kernel or modules in other domains. Furthermore, the control flow manager also supports run-time stack protection and safe stack. The binary rewriter rewrites the binary to protect memory from three kinds of accesses: st(store), ret(return), and icall(indirect call). For example, the store instruction: *st Z, Rsrc*, that stores the content of register *Rsrc* into the memory addressed by Z, i.e, [Z] = Rsrc. After the operation of the rewriter, this instruction is replaced by a sequence of instructions shown in Figure 1.

```
push R27
push R26
push R0
movw X,Z
mov R0,Rsrc
call write_access_check
pop R0
pop R26
pop R27
```

Figure 1. The *store* instruction is rewritten by a sequence of instructions.

The *write_access_check()* function verifies that the value of Z is legal or not, i.e, whether the memory addressed by Z is allowed to be accessed by the module. The return and indirect call instructions are rewritten and verified in a similar way.

The binary rewriter is an application in a desktop computer. By contrast, the verifier is usually performed at the sensor nodes. After receiving a binary image, the verifier examines the content of the image to ensure that the image is sufficiently sandboxed by a writer to prevent any possible protection violation.

Nevertheless, Harbor ignores the protection of load instructions. Without proper protection, a load instruction could access an unauthorized memory to load sensitive data into a register. Furthermore, we complete the protection scheme to incorporate a fault recovery mechanism in this paper. Finally, we incorporate an off-line checking scheme to reduce the run-time checking overhead.

3. System Architecture

Figure 2 shows the system architectures performed at a desktop computer. At the desktop computer, the original module is first inspected by an off-line checker to verify whether the static absolute addresses, i.e., addresses that would not be relocated during run-time, are legal or not. Then, the checked module is modified by a modifier that replaces all memory access instructions by a sequence of instructions so as to sandbox them. Then, the sandboxed module is distributed to a network of sensor nodes.

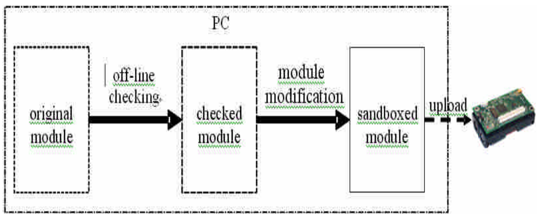


Figure 2. The software architecture at PCs.

By contrast, Figure 3 shows the system architectures performed at the sensor nodes. Notably, the sequence of instructions inserted by the modifier to replace the memory access instruction would pass the target address of a memory access instruction as a parameter to the address checker. After receiving the target address, the address checker verifies the address is valid or not. If an invalid address is found, the address checker invokes the recovery manager to recover the fault by the idea of *n*-version programming.

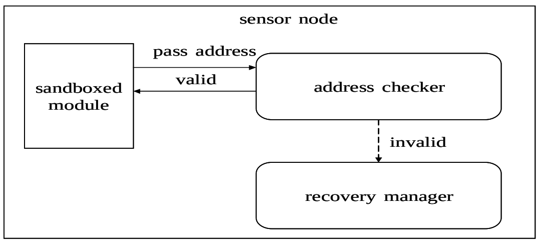


Figure 3. The software architecture at nodes.

3.1. Off-Line Checking

As stated above, we introduce the off-line checking that verifies the static addresses off-line to reduce the run-time checking overhead.

The Atmega 128L, the micro-processor in the Mica2 Mote, uses the AVR instruction set. In the AVR instruction set, instructions that could accept a static address as the operand are the *direct call* and *relative jump*. Table 1 shows the syntax and operations of these instructions.

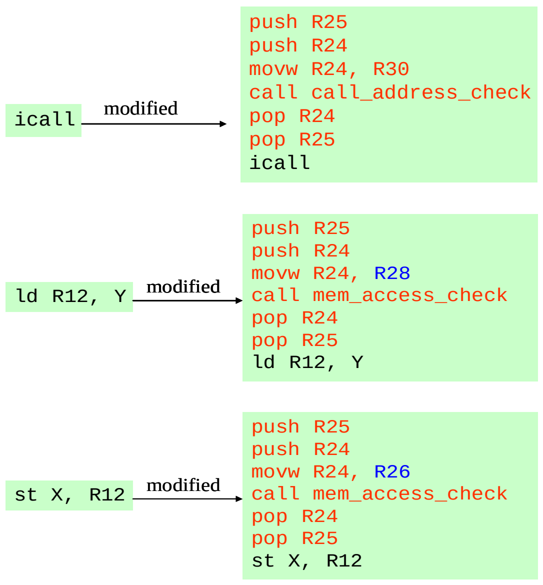
Table 1. The syntax and operating of *call* and *jmp* instructions.

Instruction	Operation
`call k`	PC = k
`jmp .+k`	PC = PC + k
`jmp .-k`	PC = PC – k

The direct call is used to invoke a system call in SOS. As shown in Section 2.1, in SOS, each system call must go through a system jump table. Figure 4 shows a partial layout of the system jump table. The number at the end of each row represents the system call number. Since the system jump table is stored at a fixed memory address, thus, the addresses of all entries in the table can be determined during assembly.

```
jmp ker_sys_codemem_read ;          21
jmp ker_sys_shm_open ;              22
jmp ker_sys_shm_update ;            23
jmp ker_sys_shm_close ;             24
jmp ker_sys_shm_get ;               25
jmp ker_sys_shm_wait ;              26
jmp ker_sys_shm_stopwait ;          27
jmp ker_sys_foo ;                   28
```

Figure 4. A part of the system call jump table.



```
icall       --modified-->   push R25
                            push R24
                            movw R24, R30
                            call call_address_check
                            pop R24
                            pop R25
                            icall

ld R12, Y   --modified-->   push R25
                            push R24
                            movw R24, R28
                            call mem_access_check
                            pop R24
                            pop R25
                            ld R12, Y

st X, R12   --modified-->   push R25
                            push R24
                            movw R24, R26
                            call mem_access_check
                            pop R24
                            pop R25
                            st X, R12
```

Figure 5. Modifications of the *icall*, *ld*, and *st* instructions.

Assume that the starting address of the jump table is ***x*** and the number of entries is ***i***. Since each entry in the system jump table occupies two bytes. Thus, the end address of the jump table ***y*** = ***x*** + ***i*** x 2. Furthermore, assume that the value of the memory address, i.e, ***k*** in Table 1, in a *call* instruction is ***j***. Consequently, to verify whether the value of ***j*** is valid nor not, we only need to check the following two conditions.

$\boldsymbol{x} <= \boldsymbol{j} < \boldsymbol{y}$ (1)

$\boldsymbol{z} = ((\boldsymbol{j} - \boldsymbol{z}) / 2)$ is an positive integer (2)

If the value of ***j*** satisfies the above two conditions, then it must be a valid memory operand.

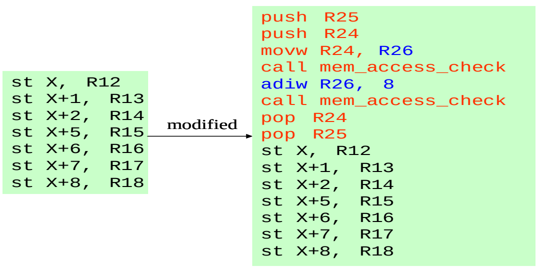
Furthermore, in SOS, the function calls within the same module uses the relative jump instructions. Consequently, to verify whether the offset value, i.e, ***k*** in Table 1, in the *jmp* instruction is valid no not, we calculate the target address by adding or subtracting ***k*** to or from the value of PC (Program Counter). If the target address points the first instruction following a code label in the same module, then the target address, and thus the value of ***k***, must be valid.

Table 2. The syntax and operation of *icall*, *ld*, and *st* instructions.

Instruction	Operation
`icall`	PC = Z(R31:R30)
`ld Rd, X`	Rd = [X(R27:R26)]
`ld Rd, Y`	Rd = [Y(R29:R28)]
`ld Rd, Z`	Rd = [Z(R31:R30)]
`st X, Rr`	[X(R27:R26)]=Rr
`st Y, Rr`	[Y(R29:R28)]=Rr
`st Z, Rr`	[Z(R31:R30)]=Rr

3.2. Module Modification

After passing the off-line checking, the module is then modified by a modifier. The instructions that are modified include: indirect call (*icall*), load (*ld*), and store (*st*) instructions. The syntax and operation of these instructions are shown in Table 2. Furthermore, the sequences of instructions that are used to replace the original instructions are shown in Figure 5. Notably, we now only support the modification of an assembly program. Consequently, a module that is programmed using high-level languages such as C must be first compiled into an assembly program. After modification by the modifier, the sandboxed program is then assembled into object files.



```
st X,   R12                 push  R25
st X+1,  R13                push  R24
st X+2,  R14                movw R24,  R26
st X+5,  R15                call mem_access_check
st X+6,  R16  --modified--> adiw R26,  8
st X+7,  R17                call mem_access_check
st X+8,  R18                pop  R24
                            pop  R25
                            st X,   R12
                            st X+1,  R13
                            st X+2,  R14
                            st X+5,  R15
                            st X+6,  R16
                            st X+7,  R17
                            st X+8,  R18
```

Figure 6. Optimizations for continuous memory accesses.

Furthermore, for continuous memory accesses, we introduce an optimization scheme to reduce the run-time checking overhead. Figure 6 shows a continuous of store instructions that store the values of a set of registers, i.e, *R12* to *R18*, into a continuous memory region. If each store instruction is replaced by a sequence of instructions showed in Figure 4, it would results in a total of 49 instructions. However, since all of the memory operands in these store instructions are continuous, thus we only need to check whether the starting and end memory operands is legal or not before the execution the first store instruction.

3.3. Run-Time Checking

As shown in Figure 5, we add two functions to perform the run-time checking. One is the *call_address_check()* that verifies the target address of an indirect call instruction. The other is the *mem_access_check()* that examines the target address of a load or store instruction.

Firstly, we explain the operation of the *call_address_check()* function. As stated in Section 2.1, in SOS, a module can only invoke the functions within itself, kernel functions exported in the system call jump table, or the subscribed functions belonging in other modules. Thus, the target address of an indirect call instruction is valid only when it belongs to the above any one region. The details are discussed below.

- **The system call jump table:** In Section 3.1, we have learned how to derive the starting and end addresses of the system call jump table. Depending on the value of the target address, there are three different situations.
  - If the target is smaller than the starting address of the jump table, then this indirect call instruction must be invalid. Notably, in SOS, the memory before the jump table is used to store the kernel's data structures and can only be accessed by the kernel.
  - If the target address falls between the starting and end address of the jump table and satisfies equation (2), then the indirect call function invokes a system call and is valid.
  - If the target address larger than the end of the jump table, then we perform the following next checking.
- **Functions that are subscribed:** In SOS, each module has a module header that maintains the functions subscribed by this module. Thus, when loading a module, we modify the loader to read the module header and record the addresses of functions that are subscribed by this module in a *checking table*. Thus, during the execution of this module, the target address of an indirect call is valid when the address is equal to one of the addresses kept in the checking table. Otherwise, we continue to the next checking.
- **Functions that within the module itself:** Since the starting address of a module is determined by the loader, thus, we also modify the loader to record the starting and end address (by adding the starting address with the module length) of a module in the *checking table*. Then, whether a target addressed is valid is verified by the procedure shown above. Notably, if the address is still invalid, no further checking is needed since all three cases have been verified. Thus, the *call_address_check()* invoke the recovery manager to recovery from the fault.

Secondly, we explain the operation of the *mem_access_check ()* function. In SOS, each module can only access the data within itself. Nevertheless, there are kinds of data: one is the static allocated data in the data segment and the other is the dynamically allocated data. Notably, similar to the code segment shown above, the starting address of a data segment is also determined by a loader. Thus, to verify whether a target address points to the static data, we also modify the loader to record the starting and end addresses of a data segment in the checking table. For dynamically allocated data, we modify the memory allocation code in the kernel to record the starting and end address of the dynamically allocated memory to the checking table. Then, the validation of a target addressed is examined by the procedure shown before.

3.4. Fault Recovery

We apply the *n*-version programming to recover from memory protection fault. When a fault is detected by the schemes shown in Section 3.3, we stop the execution of the module, remove the checking table from memory, and send a MSG_REPLACE message to the server. After receiving the message, the server first kills the faulty module by removing the module from the sensor node. Then, it uploads a different version but has the same function of the faulty module to the sensor nodes. The flow is shown in Figure 7.

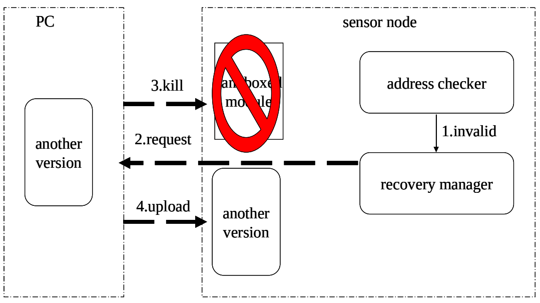


Figure 7. The fault recovery procedure.

4. Experimental Results

We have implemented our system on the SOS operating system running in the Mica2 Mote sensor node.

4.1 Code Size

Table 3 compares the code size and memory overhead of Harbor and our system with a native

SOS. From Table 3, our system has a smaller code size compared to Harbor. Nevertheless, we have a larger memory overhead due to the storage of the checking table.

Table 3. Code and memory overhead for Blank SOS kernel

Memory	Raw	Harbor (2 domains)	Harbor (8 domains)	Our implementation
Flash	41796B	+6146 B	+6228B	+3386B
RAM(MAX)	2892B	+148B	+276B	+319B

Table 4. Code size increase of SOS modules

Module	Raw	Harbor		Our Implementation	
Blink	150B	+48B	+32%	+60B	+40%
Tree Routing	2820B	+1658B	+59%	+1680B	+60%
Surge	542B	+350B	+65%	+372B	+69%
DVM	13072B	+6652B	+51%	+6512B	+50%
FFT	3016B	+894B	+30%	+716B	+24%

Furthermore, since all memory access instructions are replaced by a sequence of instructions by the modifier, we also measure the relative increase in size of modules. Table 4 shows the experimental result. Generally, our system obtains a better perform for modules that have continuous memory accesses such as FFT and DVM due to our optimization scheme. By contrast, we would have a larger code size overhead in non-continuous memory access.

4.2 Execution Overhead

Table 5 shows the performance impact of our system compared to SOS. From Table 5, due to the increased instructions inserted by the modifier, our scheme results in a larger execution overhead. Nevertheless, for a critical application, such an increase of overhead is acceptable since safety is the most important issue.

Table 5. Relative performance of applications

Module	Time(ms)	Slowdown
FFT	3.6	-
FFT-our implementation	20.5	5.7
Outlier Detector	0.18	-
Outlier Detector-our implementation	1.67	9.3

4.3 The Fault Recovery Time

Finally, we measure the fault recovery time. We divide the time into three parts: removing a faulty module, processing in PCs, and uploading a new version. The result is shown in Table 6. Notably, the processing time in PCs have the smallest value since the extremely high performance of the PCs compared to the sensor nodes.

5. Conclusion

In this paper, we propose and implement a software-based memory protection scheme on the SOS operating system. We not only detect memory access faults but also recovery from the fault. Furthermore, all four memory access instructions: st(store), return, icall (indirect call), and ld (load) are checked to provide a complete memory protection system. Finally, we also introduce the off-line checking to reduce the run-time checking overhead.

Nevertheless, not only memory access would result faults, hardware would also be broken during the execution. Consequently, our future work would investigate how to detect and recovery from hardware failures to further enhance the safely of a sensor system.

Table 6. The fault recovery time

remove buggy module	processing in PCs	upload new module	total
1.121 seconds	0.017 seconds	3.575 seconds	4.713

References

[1] A. Arora et al, "A Line in the Sand: A Wireless Sensor Network for Target Detection, Classification and Tracking," Computer Networks, vol. 46, no. 5, pp. 605-634, December 2004.

[2] A. Avizenis, "The N-Version Approach to Fault-Tolerant Software," IEEE Trans. on Software Engineering, vol. 11, no. 2, pp. 1491-1501, 1985.

[3] Berkeley mica motes, http://www.xbow.com/

[4] Rahul Balani, Chih Chieh Han, Ram Kumar Rengaswamy, Ilias Tsigkogiannis, Mani Srivastava, "Multi-level software reconfiguration for sensor networks," Proceedings of the 6th ACM & IEEE International conference on Embedded software, pp. 112 - 121, 2006.

[5] C. C. Han, et. al., "SOS: A dynamic operating system for sensor networks", *the Third International Conference on Mobile Systems, Applications, And Services*, Seattle, pp. 163-176, June 2005.

[6] R. Kumar, E. Kohler, and M. Srivastava, "Harbor: Software based memory protection for sensor nodes," *Proceeding of the 6th International Symposium on Information Processing in Sensor Networks*, pp. 340-349, 2007.

[7] P. Levis, D. Gay, and D. Culler, "Active sensor networks," *Proc. 2nd Symposium on Networked Systems Design and Implementation*, 2005.

[8] Takahiro Shinagawa, Kenji Kono, and Takashi Masuda, "Flexible and efficient sandboxing based on fine-grained protection domains," *Proceedings of the 15th International Symposium on System Synthesis*, pp. 172-184, 2002.

[9] B. L. Titzer, "Virgil: Objects on the head of a pin," *Proc. 21st ACM SIGPLAN Conference on Object-Oriented Systems, Languages, and Applications*, 2006.

[10] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham," Efficient software-based fault isolation," *Proc. 14th ACM SOSP*, pp. 203–216, 1993.