Towards a Resilient Operating System for Wireless Sensor Networks

Hyoseung Kim and Hojung Cha
Department of Computer Science, Yonsei University
Seodaemun-gu, Shinchon-dong 134, Seoul 120-749, Korea
{hskim, hjcha}@cs.yonsei.ac.kr

Abstract

Active research has recently been conducted on large scale wireless sensor networks, especially network management and maintenance, but the technique for managing application errors on MMU-less sensor node devices has not been seriously considered. This paper presents a resilient operating system mechanism for wireless sensor networks. The proposed mechanism separates the kernel from the user execution area via dual mode operation, and the access violation of applications is controlled by static/dynamic code checking. The experiment results on a common sensor node show that the proposed mechanisms effectively protect the system from errant applications.

1. Introduction

Wireless sensor networks normally consist of battery-operated, memory-limited and low performance node devices. Although the research on radio communication techniques and system software developments for resource constraint devices has recently been active [1][2], little effort has been expended on the reliability issue on sensor network systems running on MMU-less devices. Application errors on sensor nodes can affect the entire system, and the current system is ignorant of problems caused by application faults, such as immediate hardware control, kernel code execution, and kernel data corruption, so the system may collect incorrect data, or reduce the node availability. As we cannot write safe applications all the time and sensor node hardware does not easily detect application errors, techniques to ensure system resilience at the operating system level should be developed.

Currently available operating systems for wireless sensor networks include TinyOS [3], MANTIS [4], and SOS [5]. The component-based and event-driven TinyOS produces a single code image where the kernel and application are statically linked. There is no distinction between kernel and application, so a badly written application can cause the system to fail [6]. MANTIS provides a multithreaded programming model, but it is not free from the possibility of user errors due to a statically linked image, as is the case of TinyOS. SOS separates the kernel and application modules via dynamically loadable modules. This technique, however, does not include measures to restrict the application from accessing kernel data or other application data, and calling kernel code abnormally. Concerning the system errors, some operating systems use a watchdog timer, but it is not easy to recognize and handle problems such as memory access beyond the application area, immediate hardware control, and error repetition. Users have to reset sensor nodes directly to recover from specific errors [7]. Meanwhile, Maté [6] is a virtual machine for wireless sensor networks. The interpreter in the virtual machine enables the detection of hazardous instructions in the runtime, but there are also limitations, such as performance degradation or additional efforts to learn new programming languages for the virtual machine [5].

The operating system mechanism for error-free wireless sensor networks should operate with software assistance. The mechanism ought to protect the kernel from applications and support diverse applications in a multithreaded environment. However, previous work on the software approach to ensure system safety has typically focused on the MMU-equipped general purpose systems, and on the single-threaded model. SFI (Software Fault Isolation) [8] modifies the data access via indirect addressing or jump instructions to be executed in a single allocated segment, which requires MMU. SFI was designed for RISC architectures that have fixed-length instructions. [12] suggests that SFI can cause illegal jump operations when it is implemented on variable-length instruction architecture, which are usually adopted in sensor node platforms. Proof-carrying Code [9] evaluates the application using a safety policy in compile time. However, as the automatic policy generator does not exist, the technique cannot be applied directly to real systems. Programming language approaches include Cyclone [10], Control-C [11], and Cuckoo [12]. All of these are based on the C programming language, but users should be aware of the different usages of pointers and arrays. Cyclone requires hardware supports for stack safety; Control-C aims to offer system safety without runtime checking, although additional hardware is required for stack safety and the language does not guarantee fault-

free array indexing; Cuckoo provides system safety without hardware supports. The overhead of Cuckoo is, however, not trivial – being almost double the size of the optimized GCC execution time.

This paper presents a resilient operating system mechanism for wireless sensor networks. The proposed mechanism is designed to apply to the common sensor node platform with RETOS, the preemptively multi-threaded operating system we are currently developing, and enables sensor node systems to run safely from errant applications without hardware supports. The mechanism implemented in the RETOS kernel detects harmful attempts on system safety by applications, and terminates the badly-written application programs appropriately. The effectiveness of the proposed mechanism is validated by experiments conducted on a commercial sensor node device running the RETOS operating system.

The rest of this paper is organized as follows: Section 2 describes background on the system software and the hardware platform used in the paper; Section 3 explains the proposed safety mechanism; Section 4 validates the effectiveness of the mechanism via real experiments; and Section 5 concludes the paper.

2. RETOS Architecture

The proposed safety mechanism is based on the RETOS operating system. Figure 1 illustrates the system organization. RETOS provides preemptive multithreading based on a dual mode operation that cooperates with static/dynamic code checking to protect the system from application errors. (Dual mode operation normally refers to running with two hardware privilege levels, but in this paper, we use the terminology as the kernel/user separated operation.) To elevate the memory efficiency of dual mode operation, a single kernel stack is maintained in the RETOS kernel. This technique restricts thread preemption to be performed in user mode, but the amount of required system memory is decreased. With the thread preemption, hardware contexts are saved in each thread's TCB (Thread Control Block) due to kernel stack sharing. The mechanism for ensuring system safety with dual mode operation and static/dynamic code checking is described in Section 3 in detail.

RETOS supports the POSIX 1003.1b real-time scheduling interface. Threads are scheduled by three scheduling policies: SCHED_RR, SCHED_FIFO, and SCHED_OTHER. Thread scheduling is done in kernel level, while synchronization methods reside in user level. When synchronization primitives like *mutex* are executed, the thread library disables interrupts in user level and tries to acquire the resource. On an unsuccessful case, the library inserts the thread information to the resource waiting list and blocks the thread. After the resource is unlocked, the library sequentially wakes up the threads in the waiting list.

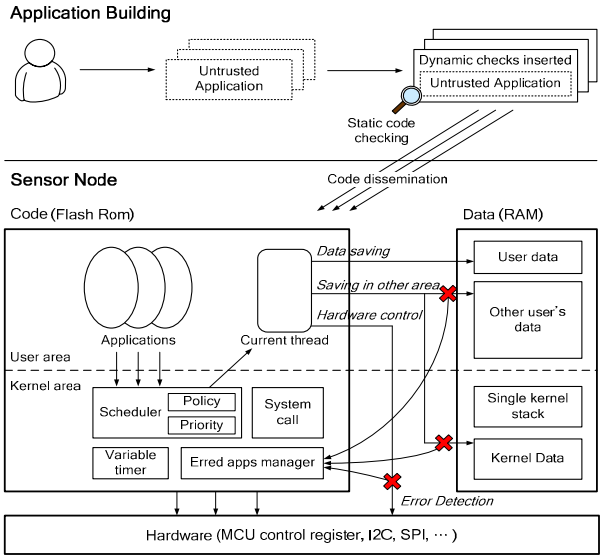

Figure 1. RETOS System Overview

In the RETOS system, applications are separated from the kernel code, and several applications can be loaded and executed dynamically. To exploit runtime application loading in a single address space, the address relocation technique is employed. The method produces an address relocation table during the compile time, and relocates the addresses of the binary to the ROM/RAM address, which is allocated for the application. The memory manager for the data/stack area management of loadable applications is based on first-fit allocation. Dynamic memory allocation is currently available only for the kernel codes.

RETOS is being implemented on the Tmote Sky [13]. The mote is based on the TI MSP430 F1611 (8Mhz, 10Kb internal RAM, 48Kb internal Flash) micro-controller, and the Chipcon 2420 RF module. The MSP430 instruction set consists of core instructions and emulated instructions. There are three types of core instructions: dual-operand, single-operand, and jump. Users program sensor applications with the standard C and the Pthread library.

3. Ensuring System Safety

The proposed mechanism ensures system resilience via dual mode operation and application code checking. Since general microcontrollers such as MSP430 only have a single address space, the kernel and applications exist in the same address space. Dual mode operation logically separates the kernel and the user execution area. Mote devices cannot protect applications' address spaces or keep the hardware resources controlled. Application code checking which consists of both static and dynamic techniques solves the problem. The concept of the safe

operating system mechanism proposed in the paper is summarized in Figure 1. User applications become trusted code through static/dynamic code checking. Applications loaded on the system are allowed to store data and to execute codes in their own resources, but application errors that were not detected at the compile time are reported to the kernel. When the errors are reported, the kernel informs users of the illegal instruction address and safely terminates the program.

3.1. Dual Mode Operation

Dual mode separates the kernel from the user execution area to perform application code checking in the target operating system, RETOS. Since the static/dynamic code checking evaluates if the application modifies data or issues codes in its allocated area, preemption in the system which executes kernel and user code in the same stack would invoke problems. For example, a thread which has access rights to other threads in a common application group would destroy the stored kernel data, such as a return address and hardware contexts, in the blocked thread stacks.

In the proposed mechanism, dual mode is operated by stack switching. Applications in the user mode use the user stack, and the stack is changed to the kernel stack for system calls and interrupts handling. Figure 2 shows the dual mode operation for system call handling on the proposed system. System calls are implemented by a function call on the TI MSP430 microcontroller, so the return address remains in the user stack, thereby leaving it to be modified by other threads. Upon system call, the current stack pointer indicating the user stack and the return address are stored in the PCB, and the runtime stack is changed to the kernel stack. Therefore, the return address validation is necessary before returning to the user mode. The case of interrupt handling is similar to the one illustrated in Figure 2. When an interrupt is invoked, MSP430 pushes the program counter in the current stack and jumps to the corresponding interrupt handler. The handler function switches to the kernel stack, if the system was in the user mode, and checks the return address after processing.

Dual mode may incur memory overhead on resource-constraint sensor nodes due to the per-thread kernel stack. To save memory usage, the proposed mechanism maintains a single kernel stack in the system. Kernel stack sharing means that the system cannot arbitrarily interleave execution flow, including thread preemption, while they are in the kernel mode. Instead, the kernel provides the deferred invocation. System calls, such as radio communication, register their long-running tasks to be executed later and return as soon as possible. Thread switching is performed right before returning to user mode, that is, the time when all work pushed on the kernel stack is finished. Although the single kernel stack is unable to preempt threads in the kernel mode, it enables the memory efficient implementation of dual mode operation in the soft real-time system, where the preemptive kernel is not strongly required. In addition to memory overhead, mode switching overhead is found in interrupts and system calls handling. Section 4 evaluates such overhead.

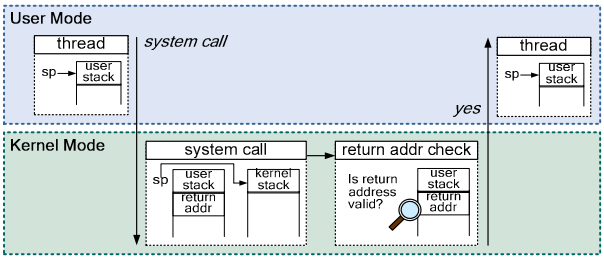

Figure 2. Dual Mode Operation

3.2. Static/Dynamic Code Checking

Static/dynamic code checking sets restrictions on an application for using data and the code area within the application itself, and restricts direct hardware resource manipulation. The proposed technique inspects the destination field of machine instructions. The destination address evaluation prevents the application from writing or jumping to an address outside its logically separated portion. The destination field is observed if it is a hardware control register, such as the MCU status control register. The source field of instructions can also be examined to prevent the application from reading kernel or other applications data. We, however, do not adopt the mechanism because of the security issues as well as computation overhead. The code checking technique considers non-operand instructions such as *ret* and *eint*/*dint*. This technique evaluates a return address of function for *ret*, and looks up *eint*/*dint* to disallow immediate hardware control.

The technique consists of static and dynamic code checking. Direct/immediate addressing instructions, pc-relative jumps and *eint*/*dint* are verified during the compile time. As we assume the library codes are safe, the libraries are not checked in our implementation. Indirect addressing instructions are verified in runtime due to unpredictable destination addresses. Runtime checking is required of the *ret* instruction, because the return address can be affected by buffer overrun. Figure 3 shows the application building sequence, including static/dynamic code checking. Every source code of the application is compiled to assembly code; then checking code is inserted to the place where the dynamic code checking is required. After dynamic code insertion, a

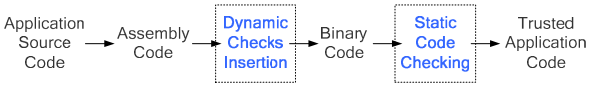

Figure 3. Application Building Sequence

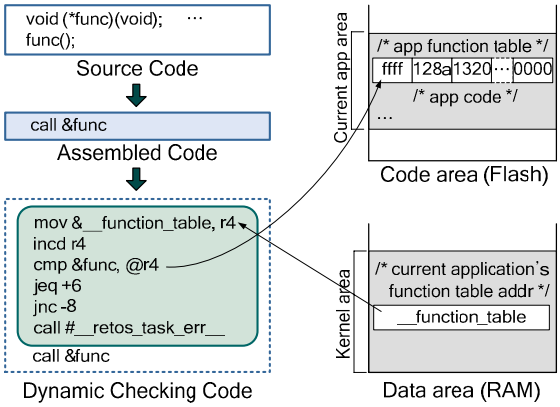

Figure 4. Dynamic Code Checking

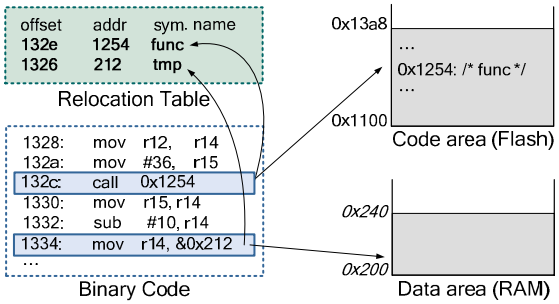

Figure 5. Static Code Checking

binary image is created via compiling and linking, and the static code checking is then performed on the binary.

Figure 4 is the example of dynamic code checking for TI MSP430. Since the technique for indirect calls inspects the destination address using the application's function table, it makes the instruction unable to branch directly to a hazardous instruction by passing the inserted checking code or to the midst of the instruction. Here, an application binary should include its function table in the header, and the kernel should provide a variable to save the address of the current application's function table and update it at the thread scheduling. The technique for call/push instructions does not consider stack overflow because the stack usage verification is conducted at the function prologue by way of the stack depth count in the function. The indirect store instruction is examined similarly as shown in Figure 4. The difference is to check every dynamically allocated RAM area for threads in the application using the linked list. Note that the r4 register shown in the Figure 4 checking code is configured to not be used by the MSP430-GCC.

Figure 5 shows an example of static code checking. This technique identifies the existence of the destination address in the relocation table, which has all the symbol information in the source code, and the offsets of those instructions referring to the symbols. If the address is not in the relocation table, the instruction will be confirmed as incorrect. For instance, the destination address of the instruction call 0x1254 in Figure 5 exists in the relocation table, and its offset is identical to the one in the table. In addition, the technique estimates the destination address within a dedicated area because data and code size is limited in the mote system. The address 0x1254 in the figure is between the application start address 0x1100 and the end address 0x13a8. Hence, the instruction call 0x1254 is proved correct. The *mov* instruction in the figure is also checked to be correct by the same method.

4. Evaluation

This section describes the experiment results of the proposed resilient operating system mechanism, and analyzes its performance characteristics. Both the mechanism and RETOS have been implemented for the TI MSP430 F1611 (8Mhz, 10Kb RAM, 48Kb Flash) based Tmote Sky hardware platform.

4.1. Functionality Test

To adequately evaluate the error management of the proposed mechanism, we classify the safety domain of applications into four parts: stack, data, code and hardware. Stack safety means the prevention of stack overflow due to function calls and local variable handling. Data safety implies the protection of the kernel and other applications data against illegal access, and code safety restricts the execution of the kernel and other applications code. Hardware safety implies whether the system can be protected from immediate hardware control. Table 1 shows the examples of the codes for each safety domain. A recursive call verifies the stack safety of the proposed mechanism. Modifying data outside the application's area, by using a directly addressed pointer and deviated array indexing, is done to analyze data safety. A directly addressed function call and a corrupted return address due to buffer overflow are used to test code safety. An interrupt disabling code and a Flash ROM writing code are used for the hardware safety check.

Following the functionality tests, the proposed mechanism is proved to ensure system resilience against hazardous application codes. An application, including a recursive call, an illegal array indexing, and a buffer overflow, is reported to the kernel by dynamic code checking and is terminated safely. The static check detects storing at and calling the address outside the

Table 1. Example Codes for Functionality Test

	Test Set
Stack safety	- General/Mutual recursive call void foo() { foo(); }

Data safety	- Directly addressed pointer
	int *tmp = 0x400; *tmp = 1;
	- Array indexing
	int array[10]; /* array in heap area */
	for(i = 10; I > 0; i--) array[i-100] = i;

Code safety	- Directly addressed function pointer
	void (*func)(void) = 0x1000; func();
	- Buffer overrun (damaging return address)
	void func() { int array[5], i;
	for(i = 0; i < 10; i++) array[i] = 0; }

Hardware safety	- Disable interrupt asm volatile ("dint"); - Flash rom writing (memory mapped regs.) FCTL1 = FWKEY+WRT;

Table 2. Dual Mode Overhead (cycle)

	Single mode	Dual mode
system call (led toggle)	264	302
system call (radio packet send)	352	384
timer interrupt (invoked in kernel)	728	728
timer interrupt (invoked in user)		760

application, which are caused by directly addressed pointers, interrupt disabling and Flash ROM writing. In order to compare the existing sensor network operating systems, that provide no safety mechanism, TinyOS v1.1.13 is selected to execute the application code in Table 1. The application using the directly addressed pointer and deviated array indexing does not perform well. The codes for recursive call, directly addressed function pointer, buffer overflow, interrupt disabling and the Flash ROM writing crash the system. The watchdog timer reboots the system sometimes, but the system is not restored most of the time.

4.2. Overhead Analysis

The proposed mechanism may have some overhead due to dual mode operation and dynamic code checking. The first set of experiments aims to analyze the performance of dual mode operation. We have implemented two versions of RETOS, dual mode and single mode, to measure performance degradation from mode switching and the return address check. Table 2 shows the results.

The experiment data shown in Table 2 denotes approximately 32~38 cycles of computational overhead for system calls, toggle a led and send a radio packet, and dual mode operation. At the timer interrupt handling, however, operation time differs from the interrupt that occurred in kernel mode and user mode. As the stack is not changed and the return address checking is omitted in kernel mode, the result of handling the timer interrupt invoked in kernel mode on the dual mode system is identical with the result on single mode. The overhead of the timer interrupt invoked in user mode on the dual mode system is 32 cycles, which is similar to the case of system calls.

Table 3. Dynamic Code Checking Overhead (cycle)

	No check	Dynamic check	Overhead(%)
MPT_backbone	40326	40508	0.5
MPT_mobile	386566	394624	2.1
R_send	24815	26176	5.5
R_recv	4704	5010	6.5
Sensing	1056	1096	3.8
Pingpong	1243	1347	8.4
Surge	58723	62688	6.7

Table 4. Application Code Size Comparison (bytes)

	No check	Dynamic check	Overhead(%)
MPT_backbone	614	682	11.1
MPT_mobile	8726	10046	15.1
R_send	946	1014	7.2
R_recv	902	1004	11.3
Sensing	774	835	7.9
Pingpong	478	510	6.7
Surge	1436	1610	12.1

The second set of experiments was conducted to observe the execution time overhead of dynamic code checking. We compare the codes with dynamic checks to original applications running RETOS by calculating average instruction cycles per second during 10 minutes. To measure the overhead of inserted codes, we consider the execution time running in user mode. Seven sensor network applications are used for the test. *MPT_mobile* and *MPT_backbone* are decentralized multiple object tracking programs [14]. When *MPT_mobile* node moves around, it sends both an ultrasound signal and a beacon messages every 300ms to nearby *MPT_backbone* nodes. *MPT_backbone* nodes report their distance to the mobile node, and *MPT_mobile* computes its location using trilateration. *R_send* and *R_recv* are programs to send and receive radio packets with reliability. *Sensing* samples the data and forwards it to the neighbor node. *Pingpong* makes two nodes blink in turns by means of the counter exchange. *Surge* is a multihop data collecting application which manages a neighbor table and routes the packet.

Table 3 shows that applications using dynamic code checking have 0.5~8.4% performance degradation. *MPT_mobile*, which requires the longest processing time, generated 2% more overhead when the protection mechanism was used; the amount of calculation time caused by non-hardware multiplier is much larger than dynamic checking. Whereas, *R_recv, Pingpong,* and *Surge,* all of which require more memory access than complex arithmetic calculations, shows larger overhead. Since the dynamic code checking requires code insertion, the technique increases the application code size when compared to the original one. Table 4 shows the code size comparison. The increases are 4~12%. However, the application is inherently small, being separated from

kernel, and hence code size increases are not considerable for the internal Flash ROM of TI MSP430 or AVR ATmega128L, the common microcontrollers used for sensor node devices.

The performance of dual mode operation, the computational overhead and code size increment of dynamic code checking are analyzed. Since the system stays idle most of the time due to sensor network application characteristics, energy consumption for the proposed mechanism may be almost the same as existing systems. The overhead of dynamic code checking, however, heavily depends upon application types and the programmers' coding style. Generally, frequent usages of local variables, function calls, and pointers cause more overhead.

5. Conclusion

In this paper, we presented a resilient operating system mechanism for wireless sensor networks, based on dual mode operation and static/dynamic code checking. The proposed mechanism guarantees stack, data, code and hardware safety on MMU-less hardware without restriction of the standard C language. Dynamic code checking with dual mode operation is reported in approximately 8% of execution time overhead on the TI MSP430 processor.

The experiments were conducted under the assumption that the libraries always operate safely, and the code checking techniques do not inspect library codes. In real situations, however, if a user passes an invalid address to *memcpy()* then the function would destroy memory contents. For a solution, we considered recompiling standard libraries with our techniques or making wrapper functions that check address parameters. Also, our mechanism cannot handle the case where a user intentionally skips the code checking sequences or modifies a program binary. To prevent malicious usage, user authentication on code updating would be required.

RETOS, safety mechanism applied operating system, is currently being developed by our research group. Although this paper shows that RETOS protects the system from errant applications, supplement for libraries and system calls is required in order to program applications easily. We are presently developing a network stack for energy efficient radio communication on RETOS, as well as implementing device drivers for diverse sensors and porting to other processors.

Acknowledgements

This work was supported in part by the National Research Laboratory(NRL) program of the Korea Science and Engineering Foundation (2005-01352), and the ITRC programs(MMRC) of IITA, Korea.

References

[1] D. Culler, P. Dutta, C. T. Eee, R. Fonseca, J. Hui, P. Levis, J. Polastre, S. Shenker, I. Stoica, G. Tolle, and J. Zhao, "Towards a Sensor Network Architecture: Lowering the Waistline," *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, Santa Fe, NM, USA, June 2005.

[2] V. Handziski, J. Polastre, J. Hauer, C. Sharp, A. Wolisz and D. Culler, "Flexible Hardware Abstraction for Wireless Sensor Networks", *Proceedings of the 2nd European Workshop on WirelessSensor Networks (EWSN'05)*, Istanbul, Turkey, January 2005.

[3] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for network sensors," *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, Cambridge, MA, USA, November 2000.

[4] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, "MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms," *ACM/Kluwer Mobile Networks & Applications (MONET), Special Issue on Wireless Sensor Networks*, vol. 10, no. 4, August 2005.

[5] C. Han, R. K. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava, "SOS: A dynamic operating system for sensor networks," *Proceedings of the 3rd International Conference on Mobile Systems, Applications, And Services (MobiSys'05)*, Seattle, WA, USA, June 2005.

[6] P. Levis and D. Culler, "Maté: A Tiny Virtual Machine for Sensor Networks," *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, San Jose, CA, USA, October 2002.

[7] Deluge: TinyOS Network Programming, http://www.cs.berkeley.edu/~jwhui/research/deluge/

[8] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Software-based fault isolation," *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP'93)*, Asheville, NC, USA, December 1993.

[9] G. C. Necula, "Proof-carrying code," *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, Paris, France, January 1997.

[10] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," *Proceedings of the 2002 USENIX Annual Technical Conference*, Monterey, CA, USA, June 2002.

[11] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. "Memory safety without runtime checks or garbage collection," *Proceedings of Languages, Compilers and Tools for Embedded Systems (LCTES'03)*, San Diego, CA, June 2003.

[12] R. West, and G. T. Wong, "Cuckoo: a Language for Implementing Memory and Thread-safe System Service", *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC'05)*, Las Vegas, NV, USA, June 2005.

[13] Moteiv, Inc., http://www.moteiv.com.

[14] W. Jung, S. Shin, S. Choi, and H. Cha, "Reducing Congestion in Real-Time Multi-Party Tracking Sensor Network Application," *Proceedings of the 1st International Workshop on RFID and Ubiquitous Sensor Networks*, Nagasaki, Japan, December 2005.