Harbor: Software-based Memory Protection For Sensor Nodes

Ram Kumar, Eddie Kohler, Mani Srivastava
University of California at Los Angeles
420 Westwood Plaza, Los Angeles, CA, USA
{ram,mbs}@ee.ucla.edu, kohler@cs.ucla.edu

ABSTRACT

Many sensor nodes contain resource constrained microcontrollers where user level applications, operating system components, and device drivers share a single address space with no form of hardware memory protection. Programming errors in one application can easily corrupt the state of the operating system or other applications. In this paper, we propose *Harbor*, a memory protection system that prevents many forms of memory corruption. We use software based fault isolation ("sandboxing") to restrict application memory accesses and control flow to protection domains within the address space. A flexible and efficient *memory map* data structure records ownership and layout information for memory regions; writes are validated using the memory map. Control flow integrity is preserved by maintaining a *safe stack* that stores return addresses in a protected memory region. Run-time checks validate computed control flow instructions. Cross domain calls perform low-overhead control transfers between domains. Checks are introduced by rewriting an application's compiled binary. The sandboxed result is verified on the sensor node before it is admitted for execution. Harbor's fault isolation properties depend only on the correctness of this verifier and the Harbor runtime. We have implemented and tested Harbor on the SOS operating system. Harbor detected and prevented memory corruption caused by programming errors in application modules that had been in use for several months. Harbor's overhead, though high, is less than that of application-specific virtual machines, and reasonable for typical sensor workloads.

Categories and Subject Descriptors: C.3 [Special-Purpose and Application-Based Systems]: Real-time and embedded systems

General Terms: Performance, Design, Reliability

Keywords: Software Fault Isolation, Memory Protection

1. INTRODUCTION

Sensor networks have promise for many industrial, commercial, and medical applications. For example, CodeBlue [13] is a prototype medical sensor network platform for expediting triage during disaster response. A network of 4000 sensors deployed by Intel in a semiconductor fabrication plant performs predictive maintenance of machinery in service [10]. The Zigbee consortium [24] seeks to equip lighting and HVAC controllers with wireless radios, enabling intelligent building automation and security services. These current and upcoming sensor network deployments require high availability infrastructure with the ability to support multiple users. Unexpected system failures could cause problems ranging from financial impacts to loss of life. Current software technology is grossly inadequate to run such long term deployments. Bugs in any part of the software can easily bring down an entire network. In particular, memory corruption due to buggy applications can crash or freeze sensor nodes or corrupt sensed data. We argue that memory protection is a vital enabling technology for creating reliable and long-lasting sensor network software systems.

Sensor software is quite complex, supporting many sensor types, multiple distributed middleware services, dynamic code updates, and concurrent applications. Programmers must deal with severe resource constraints and concurrency issues on hardware with very limited debugging support. Therefore, programming errors are quite common and can impact the network.

Mote-class sensor nodes have a very simple architecture. All primary memory is accessible to all programs running on a node via a single address space. Common mote-class architectures do not have features such as memory management units (MMUs) and privileged-mode execution used in desktop/server class systems to isolate program data and code. Embedded microcontroller designers face extreme pressure to minimize chip cost and area. Sometimes even 32-bit ARM processor cores omit an MMU to minimize system cost and power [1]. We expect MMU designs will continue to be absent from low-cost low-power microcontrollers. If sensor node applications are to be made robust, it must happen through software.

Software-based approaches for memory protection have emerged to compensate for the architectural limitations of embedded microcontrollers. Domain specific interpreters such as Maté [11] provide a safe environment to execute high-level application scripts. Type-safe languages such as Virgil [20] provide fine-grained protection of individual memory objects. However, these approaches have their limitations. For example, Maté instructions are implemented in non type-safe language and could be buggy. Type-safe languages require unsafe extensions to interface to the low-level hardware, though these extensions could be used sparingly. An ideal system for memory protection might combine two or more software-based approaches.

In this paper, we present *Harbor*, a system for providing software-based coarse-grained memory protection in resource-constrained embedded sensor nodes. Harbor can be used as a building block with other approaches to create more effective protection mechanisms. For instance, Harbor can be used to implement memory safe Maté instructions. Harbor partitions a sensor node's memory into multiple *domains*. Memory belonging to one domain is protected

This material is based in part upon work supported by the National Science Foundation under Grant Nos. 0306408 and 0435497, the Office of Naval Research under Grant No. N00014-06-1-0253, and the UCLA Center for Embedded Networked Sensing (CENS). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

from corruption by code running in other domains. We achieve memory protection by rewriting machine instructions to enforce restrictions on memory accesses. This technique, first proposed by Wahbe et al. [22], is known as software-based fault isolation, SFI, or "sandboxing".

We investigate the challenges in implementing SFI on resource-constrained embedded sensor nodes. Motes' limited address space precludes static address space partitioning: there is not enough memory available to assign each software module a single contiguous range of addresses. Scarce memory resources require Harbor to have a very small memory footprint. Limited computational capabilities also encouraged us to limit Harbor's CPU overhead. The contribution of our work has been to design techniques that make sandboxing feasible on embedded sensor nodes. First, a *memory map* data structure efficiently maintains fine-grained ownership and layout information for the entire address space. The memory map can be tuned to match available resources and protection requirements of a system. Second, a *safe stack* in protected memory preserves control flow integrity within a domain by storing function return addresses. The conventional run-time stack, which stores local data, function parameters, and so forth and is is shared by all the domains, is protected from corruption via *stack bounds*. The alternative of maintaining a separate stack per domain is not possible due to address space limitations. Third, *cross domain calls* implement low-overhead context switches between domains. The overhead of copying call arguments is eliminated as the domains share a common run-time stack. Cross domain calls and returns track the system's currently active domain. Fourth, *run-time checks* ensure that control flow in and out of a domain occur as expected even on computed transfers, and similarly that memory is accessed only as expected. To minimize the module code size, the run-time checks are not inlined. Modules invoke the run-time checks by calling or jumping into the appropriate routines located in the trusted domain, and all potentially unsafe operations are replaced by calls to corresponding checks. Calls to the checks are introduced by a *binary rewriter* and verified independently by a *verifier* running on every sensor node. Harbor's correctness depends only upon the correctness of the verifier and the Harbor runtime, and not on the rewriter. The design of the verifier affects the system's performance (Section 8). So far, we have only evaluated a simple verifier that maintains no additional state. Exploring the design space of verifiers and evaluating their impact on performance is a challenge that remains to be addressed.

These techniques are easily incorporated into existing systems. We have implemented and evaluated Harbor's protection mechanisms on the SOS operating system [8], although the ideas should apply elsewhere. During experimentation, Harbor detected memory corruption in a data collection application module that had been in use for several months. A common programming mistake in SOS is to forget to check the error code returned by a cross-domain function call. In the Surge module, under certain conditions, the invalid result of a failed function call was being used to determine an offset into a buffer. Subsequently, the data was being written to an incorrect memory location which would cause some of the nodes in the network to crash. Harbor was successfully able to prevent the corruption and signal the invalid access.

In the rest of the paper, we will describe the design and evaluation of Harbor in detail. After presenting related work and a brief overview in Section 3, we describe Harbor's run-time components, the memory map manager (Section 4) and control flow manager (Section 5). The binary rewriter and verifier are described in Section 6. Section 7 presents evaluation results. The design alternatives offered by the Harbor mechanism and our current operating point are discussed in Section 8.

2. RELATED WORK

Several recent systems address reliability as a primary design concern for long-term sensor network deployments [7, 18, 5]. *t-kernel* [7], a runtime for Mica motes, also rewrites binaries to make them safe for execution. Harbor and *t-kernel* represent different points in the design space of software-based protection mechanisms. *t-kernel* enforces a strong isolation boundary between the application and the kernel. Through a process called *naturalization*, the application binary is rewritten on the sensor node to guarantee that the *t-kernel* can always safely regain control of the processor, even, for example, in the presence of application infinite loops that could otherwise hang the system. The rewritten application binary contains an entire TinyOS operating system image, in contrast to Harbor, where a sensor node can protect multiple modules from one another. *t-kernel* also implements software-based differentiated virtual memory, which translates the addresses for all memory accesses made by a program into the heap segment. The overhead of virtual memory is unpredictable and can be very high in the event of a swap from external flash. Harbor does not implement virtual memory, but does enforce isolation at a finer granularity than *t-kernel*. In particular, Harbor can protect application modules from one another. Harbor does not address control flow isolation, except as required to enforce memory isolation; in particular, it cannot force a buggy module stuck in an infinite loop to relinquish control of the CPU. *t-kernel* requires external flash memory, whereas Harbor makes use of on-chip flash memory only.

Safe TinyOS [18] uses CCured [16] and static analysis techniques to provide memory safety to TinyOS applications. CCured performs complex pointer analysis to mark pointers as safe or unsafe. Much driver code performs arbitrary typecasts that can cause CCured to fail or conservatively mark pointers as unsafe, which introduces a performance penalty as the CCured run-time performs bounds checks on unsafe pointers during code execution. CCured provides memory safety at a much finer granularity than Harbor. The UTOS framework allows untrusted extensions to safely interface with Safe TinyOS components. UTOS extensions are made type-safe and memory-safe using CCured and a backend service that copies buffers when they are exchanged between the extension and the Safe TinyOS core. Extensions are not allowed to interact with one another. Harbor allows safe buffer transfers without copying, and allows extensions to interact, but its current simple runtime check infrastructure introduces overhead that CCured can sometimes avoid. The UTOS backend service also mediates resource requests and prevents any extension from starving other extensions in the system. Harbor does not make any guarantees on fair resource allocation.

Type-safe languages such as Virgil [20] can flag illegal accesses at compile or run time and provide fine-grained memory protection of individual objects. However, most software developed for embedded systems is currently written in unsafe languages such as C, or even assembly. Virtual machines can also provide memory protection by allowing sensor network users to program in a higher-level, safe language [11]; we discuss tradeoffs between the virtual machine and sandboxing approaches in Section 7.

Software fault isolation (SFI) was initially developed for desktop and server-class systems [22]. SFI allows multiple modules to safely share an address space by partitioning that space into contiguous ranges, one (or a small number) per module. This memory organization, common to SFI and later optimized variants, avoids the need for Harbor's memory map data structures. It is reasonable on hardware that already supports virtual memory, but not on an embedded sensor node, whose limited address space precludes such a static partitioning. Nevertheless, the SFI optimizations in PittSFIeld [14] and similar systems would apply to Harbor. The two-stack execution model used by Harbor to ensure module control flow integrity was motivated by XFI [6], a high-performance variant of SFI. XFI's scoped stack holds data accessible only in the static scope of each function, including return addresses and most local variables. A separate allocation stack stores data that may be shared within the functions in a module. Several other efforts in the desktop/server space isolate kernel modules such as device drivers either using hardware support [19] or through type-safe languages [3] or software fault isolation [9].

3. SYSTEM OVERVIEW

The goal of our work was to provide memory protection for mote-class sensor nodes running the SOS operating system. This brief introduction to SOS is useful to fully understand the implementation of our scheme.

3.1 SOS Operating System

TinyOS [12], the most popular operating system for sensor networks, uses reusable software components to implement common services, but each node runs a statically linked system image. SOS has a more traditional architecture: a kernel is installed on all nodes, and application level functionality is implemented by a set of dynamically loadable binary modules [8]. The kernel is relatively well tested; we assume it is free of programming errors. Modules are position independent binaries that implement a specific task or function, and are less well tested by comparison. Modules operate on their own state, which is dynamically allocated at run-time. An application in SOS is composed of one or more modules interacting via asynchronous messages or function calls. Examples of modules are routing protocols, sensor drivers, application programs, and so forth.

The SOS kernel supports dynamic memory allocation. Dynamic memory is used to store module state and to create messages to be dispatched to other modules. Memory is allocated using a block-based first-fit scheme to minimize the overhead of the allocation process. Limited memory forces SOS kernel and user modules to share a common allocation heap; any static partitioning would be too conservative. The kernel therefore tracks ownership of memory blocks. A block's ownership can also be transferred, allowing buffers to pass easily through various modules in the system.

3.2 Protection Domains

User modules' wild writes can easily corrupt operating system state and trigger severe failure conditions. Harbor aims to prevent this systemwide corruption by preventing user modules from corrupting memory in different *protection domains*. These protection domains are distinct subsets of a sensor node's overall data memory address space (Figure 1), created and enforced by Harbor. Each module resides in exactly one domain; SOS kernel state resides in its own, separate domain. The kernel can read and write any domain, but each module can only write into its own domain. (This simple design precludes writable shared memory regions, but SOS did not previously support such regions anyway.) The number of protection domains is subject to a tradeoff between space efficiency and fault protection. Harbor's space overhead is minimized when there are two domains, one for the kernel and one for all user modules. This protects kernel state from user wild writes, but allows any module to corrupt any other module's state. Alternatively, each module might store its state in a separate protection domain. No assumptions are made about layout of state within a domain. Run-time checks restrict user modules from writing to memory outside their domains. Checks are added to each memory write, and to jumps and other control flow transfers. The latter checks prevent applications from avoiding the former checks.

The domain protection model does not address all possible memory corruption faults; for example, modules can still corrupt their own state. This form of corruption, though undesirable, is less serious than corruption across domains, and stable kernel can always ensure a clean re-start of user modules when corruption is detected. On other hand, a corrupted kernel has truly unpredictable behavior, leaving complete system reboot through a watchdog or grenade timer as the only possible means of recovery [5].

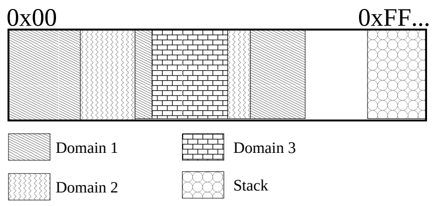

Figure 1: Protection Domains

3.3 System Components

Harbor's four components are shown in Figure 2. The system's input consists of raw user module binaries generated by a cross-compiler toolchain. The *binary rewriter* is a desktop application that statically analyzes these binaries for potentially unsafe operations and inserts run-time checks to sandbox them. The sandboxed binary is then distributed to a network of sensor nodes. A *verifier* running on each node verifies that incoming binaries are correctly sandboxed. Verified binaries admitted for execution interact closely with Harbor's run-time components, the *memory map manager* and *control flow manager*.

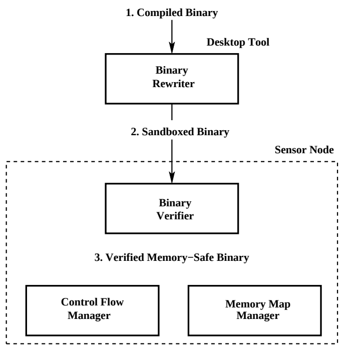

Figure 2: System Overview

4. MEMORY MAP MANAGER

Creating and enforcing protection domains is a challenging task on resource constrained embedded platforms. Initial SFI designs allowed a sandboxed module to access a single contiguous range of memory [22]. Motes' limited physical memory and absence of virtual memory precludes this partitioning, however; such a partition would constrain applications by severely limiting available memory and lead to internal fragmentation, extremely wasteful on severely resource constrained platforms. Harbor's *memory map* abstraction was designed with the following requirements: first, it should have a small and customizable memory footprint; second, it should permit arbitrary layout of state within the data memory; and third, it should be easy to incorporate into existing operating systems. We propose a design that satisfies these requirements.

4.1 Data Structure

We assume a sensor node's address space is partitioned by the operating system into small, contiguous *blocks* of equal size, then allocated to domains in *segments* consisting of sets of contiguous blocks. (On AVR, SOS's block size is 8 bytes.) The allocation of segments to domains could be static (at compile time) or dynamic (through malloc). A domain could be allocated multiple segments that are scattered randomly across entire address space. The Harbor memory map contains *per-block* access permissions for the entire address space. The main operation of the memory map is to store and retrieve access permissions for a given address. Its design goal is to balance lookup efficiency and the extra storage required for the permissions table. The memory map contains ownership information (a domain identity) for every block of memory, and encodes information about memory layout, such as the start of a logical allocation segment. The memory map must contain sufficient permission bits per block to encode the total number of domains supported by the system. Supporting two distinct domains (kernel and user) requires just one domain bit per block, four domains require two domain bits per block, and so forth. Table 1 shows an example of Harbor's memory map encoding in a system with 8 domains.

Code	Meaning
1111	Free, or start of kernel allocated segment
1110	Later portion of kernel allocated segment
xxx1	Start of user allocated segment
xxx0	Later portion of user allocated segment

Table 1: Memory map information encoding for 8-domain protection.

Figure 3 shows how an address is looked up in the memory map. Assuming a block size of 8 bytes, the last three bits of address are an offset into a given block. The remaining bits of the address represent a block number in data memory. Access permissions are packed into a byte. If encoded information is stored in four bits (for 8-domain protection), then each byte would contain information for two contiguous memory blocks. Therefore, the last bit of block number selects a memory map record from within an access permissions byte. The remaining bits of the block number form an index into the memory map table. This design was chosen to minimize memory footprint. The memory map is a configurable data structure; tradeoffs between the inter-module protection and memory map size are discussed in Section 8. The memory map data structure and address translation operations are encapsulated in an object accessible through the API in Table 2.

4.2 Using the Memory Map for Protection

Information stored in the memory map can be used for a variety of protection models; our protection model restricts programs from writing to memory outside their domain.

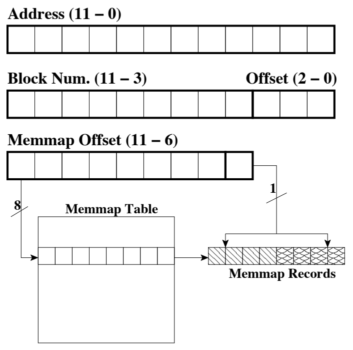

Figure 3: Address to memory map translation (8-domain mode)

Systems using a memory map need to ensure the following four conditions. First, the memory map should accurately reflect the current ownership and layout of memory. In any real system, memory is constantly allocated, freed, and/or transferred from one module to another. The memory map should be immediately updated when any of these events occur; thus, SOS's `malloc`, `free`, and `change_own` system calls were modified to update the memory map data structure. Second, only the block owner should be permitted to free or change its ownership. This condition is necessary as one module may accidentally (due to programming errors) attempt to free memory being used by other modules. To enforce this condition, the system needs to track the currently active domain (Section 5). Third, direct access to the memory map API (described in Table 2) should be restricted to trusted domains, such as the kernel. In addition, the blocks storing memory map data structures should be owned by a trusted domain, preventing accidental corruption of the memory map data structure.

A memory map can be easily incorporated into software systems. As an example, we describe how SOS's memory map provides multi-domain protection. The memory map is initialized such that all statically allocated kernel memory blocks are marked as owned by kernel. The remaining portion of the address space is partitioned into a heap, a safe stack (further described in Section 5.3), and a run-time stack. The heap is divided into blocks, so the minimum granularity of memory allocation is a block. The heap's memory map is initially marked as free. The safe stack is marked as belonging to the kernel domain. The run-time stack has no memory map; we discuss run-time stack protection in Section 5.2. Our implementation modified 150 lines of code, about 1% of the 12720-line SOS kernel; the change was mostly localized to dynamic memory management routines.

4.3 Memory Map Checker

Harbor's run-time checks validate memory accesses, in particular writes. These accesses are validated using a protection model. Our memory map checker enforces the protection model described earlier: each user module can write only into its own domain. The memory map checker belongs to the trusted domain. Pseudocode for a write access check in a system with 8-domain protection is shown in Figure 4. The write access checker performs three operations. First, it performs address translation to retrieve the byte containing ownership information from the memory map table for a given address. Second, it locates the appropriate record within that byte and determines the domain of the block's owner. Third, it compares this domain ID and the current executing user module's domain ID. A store is allowed only if these domains match. As

Prototype	Description
int8_t memmap_set(uint8_t blkID, uint8_t nBlks, uint8_t domID)	Set owner of segment [BlkID, BlkID + nBlks) to domID
uint8_t memmap_get(uint8_t blkID)	Get owner and layout of block number BlkID

Table 2: Memory map API

```
write_access_check(addr_t addr, data_t data) {
   // Check is for writes outside stack region
   if (addr < STACK_PTR) {
      // Address translation: Get table index
      uint16_t blk_num = (addr >> log2_blk_size);
      uint16_t mmap_index = (blk_num >> log2_rec_per_byte);
      // Retreive memory map byte
      uint8_t mmap_byte = MEM_MAP_PERMS_TBL[mmap_index];

      // Get the appropriate record in byte
      if (blk_num & SWAP_MASK) swap(mmap_byte);
      uint8_t mmap_owner = mmap_byte & OWNER_MASK;
      uint8_t first_blk_in_segment = mmap_byte & 1;

      // Validate access
      if (mmap_owner != curr_dom_id
          || (first_blk_in_segment && addr points to block metadata))
         mem_access_exception();

      // Perform store
      st addr, data;
   } else {
      // Check for writes to stack
      stack_access_check(addr, data);
   }
}
```

Figure 4: Pseudocode for Memory Map Checker (8-domain protection)

mentioned previously, the memory map manager does not maintain permissions for run-time stack; write accesses to the run-time stack are subject to a different check described in Section 5.2.

An implementation detail involves the protection of heap metadata, such as the owner and/or size an allocated segment. SOS stores this metadata in the segment itself. This complicates the write access check, since heap metadata is effectively kernel-domain information and must be protected from wild writes. Since the SOS kernel stores this metadata in a segment's first memory block, the memory map checker protects the metadata in the first block of any segment from user writes. The memory map table's layout information supports this check by identifying the starting block of any segment.

5. CONTROL FLOW MANAGER

Programming errors can cause a module to corrupt its own state, even with protection domains. Unfortunately, a user module's control flow might be affected by internal memory corruption. For example, function pointers (commonly used to implement callbacks) are stored in RAM. Return addresses to function call sites are stored in stack. Corruption of these values might cause the processor to execute arbitrary code, including the memory map API, violating one of the requirements of using the memory map for protection. The *control flow manager* ensures that control can never flow out of a domain except via calls to functions exported by the kernel or modules in other domains, and via the corresponding returns. Conversely, control flow can enter a domain only through an exported function or through the return site of a call that was made to a function exported by some other domain. The control flow manager also tracks the identity of the currently executing domain. This information is required by the memory map checker to validate write accesses. Harbor's *cross domain call* mechanism is used to transfer control safely from a caller to a callee domain. A corresponding *cross domain return* mechanism restores control back to a caller domain. Control flow integrity within a domain is preserved through a *safe stack* that stores return addresses.

5.1 Cross Domain Call

A cross domain call performs four operations. First, it verifies the call's target address, which should match the address of a function officially *exported* by some module or the kernel. Second, it saves the caller's domain identity and return address. Third, it sets up a *stack bound*, which prevents the callee from modifying portions of the stack belonging to the caller. Finally, it jumps to the callee. The cross domain call mechanism tries to optimize the performance of these operations.

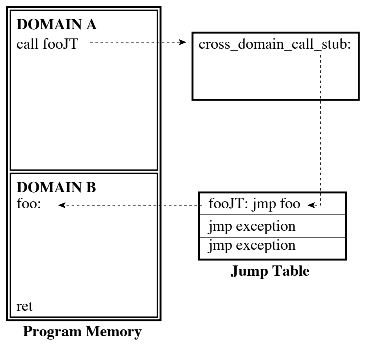

Figure 5: Cross Domain Call

Call address verification is accomplished with the help of a *jump table*, an extra level of indirection in cross-domain function calls. Modules in a domain are linked with modules in other domains at load-time. A linker running on the sensor node parses the module's exported functions and writes them to the jump table. The jump table, which is stored in flash memory, is similar in design to an interrupt vector table. Each jump table entry is an instruction to jump to an exported function; the function's address is encoded within the instruction stream. Each domain has its own jump table, containing all the functions exported by modules in that domain (or, for the kernel domain, all functions exported by the kernel). Since modules cannot directly write to flash memory, they cannot corrupt the jump table. Modules that subscribe to functions exported by a particular module are redirected through the corresponding domain's jump table. This is illustrated in Figure 5. The jump table mechanism is independent of the process used for dynamic linking (exporting and subscribing to functions), which might use several other techniques [4].

Each domain is currently allocated one page of internal flash memory for storing its jump table. In the AVR architecture, this imposes a limit of 64 exported functions per domain. SOS limits each module to exporting 12 functions, allowing at least 5 modules to share a domain. Empty entries in the jump table are filled with a jump instruction to an exception routine. All domains' jump table pages are stored contiguously in flash memory, reducing the overhead of verifying a call's target address and domain.

All function calls made across modules need to pass through a *cross domain call stub*. This stub, a part of the Harbor runtime, is located in a trusted region of program memory. In SOS, cross module calls use a macro `SOS_CALL`; we modified its implementation to force a call into the cross domain call stub. This is implemented as an assembly routine within the SOS kernel, with pseudocode shown in Figure 6.

```
cross_domain_call(addr_t addr) {
   // Store current return addr in safe stack
   push_ss ret_addr

   // Check if target address is valid
   if (addr > JMP_TBL_BASE) {
      // Store current state in safe stack
      push_ss curr_domain_id;
      push_ss curr_stack_bound;

      // Compute new domain ID
      curr_domain_id = MSB((addr - JMP_TBL_BASE) << 1);
      if (curr_domain_id > MAX_DOMAIN_ID)
         control_flow_exception();

      // Compute new stack bound (For run-time stack protection)
      curr_statck_bound = STACK_PTR;

      // Push the return address of cross domain return
      push_ss cross_domain_return

      // Call into jump table
      call addr;

cross_domain_return:
      // Restore previous state
      pop_ss curr_stack_bound
      pop_ss curr_domain_id

      // Return to caller domain
      ret
  } else
    control_flow_exception();
}
```

Figure 6: Pseudocode of Cross Domain Call Stub

The stub first stores the return address in the safe stack (Section 5.3). All valid cross-module calls must have target addresses that reside in the jump table, since modules subscribe to jump table locations corresponding to the functions exported by other modules. A call into the jump table is checked by a simple compare operation to the base address of jump table. The stub then stores the current domain identifier and the stack bound in the safe stack. A store into the stack is required because cross domain calls can be chained: domain A calls domain B, which in turn calls domain C. Next, the identity of the callee domain is computed by determining the jump table page in which the target address falls. If the target domain identifier exceeds the maximum number of domains in the system, then the target address is greater than the upper bound of jump table; an exception is generated. The callee domain may equal the current domain when a module transfers control to another module in the same domain. The cross domain return address is pushed to the safe stack, ensuring that control flow will return to the stub. Finally, a call is made into the jump table, which redirects it to the actual entry point in the target domain. During cross domain return, the previous domain identifier and stack bound are restored and the control is transferred back to caller domain.

As all domains share a common run-time stack, the cross domain call stub does not need to copy call arguments. Further, no modifications are made to any data frames set up in the run-time stack during function calls. Therefore, a single cross domain call and return stub sufficies for all cross domain calls, unlike the per-function stub required by the original SFI [22]. The implementation of the cross domain call stub only uses the caller-saved registers described by the `avr-gcc` ABI.

Harbor currently disallows all computed branches except for cross module calls. This ensures that applications cannot avoid memory and/or control flow checks, but also prevents certain implementations of control flow structures like `switch`.

5.2 Run-Time Stack Protection

Harbor shares a common run-time stack across all domains. The design alternative, allocating a private stack per domain, would require too much memory, since stack memory must be allocated conservatively (the stack can grow significantly during execution). Harbor implements a *stack bound* to prevent one domain from corrupting another domain's local variables and other stack information. The cross domain call stub sets up a stack bound before transferring control, and the cross domain return stub restores the previous stack bound. As shown in Figure 4, the stack access checker is invoked for writes to the run-time stack, a statically allocated region of memory at the high end of the address space. Harbor disallows writes to memory addresses greater than the current stack bound.

The stack bound does prevent cross domain data sharing through the stack, but we have never encountered an instance of such sharing in the SOS system.

5.3 Safe Stack

Correct fault isolation requires that Harbor limit control flow *within* modules, as well as across modules. A module must not be able to jump into its code arbitarily, since this might allow it to avoid a run-time check. The Harbor runtime therefore uses an additional *safe stack* to preserve the integrity of control flow within and across modules. The safe stack resides in the trusted domain, preventing any corruption by application wild writes. Harbor stores function calls' return addresses on the safe stack, protecting them from wild writes by applications in any domain. The cross domain call stub also uses the safe stack to store the current domain ID and run-time stack bound. The safe stack pointer is maintained as a global variable, and manipulated by sequences of push and pop operations. A function entry stub, `func_entry_stub`, copies the return address from the run-time stack onto the safe stack. Similarly, `func_exit_stub` pops the return address from the safe stack and restores the run-time stack. The entry and exit points of every local function within a domain are rewritten to invoke these stub routines. We do not modify the run-time stack in any manner as this would corrupt data frames setup by functions for storing local data and function arguments.

The safe stack can be placed anywhere in data memory as long as it is protected from accidental writes and overflow. We usually place the safe stack at the end of all global data in the system and make it grow upwards. The run-time stack and safe stack thus approach one another.

6. BINARY REWRITER AND VERIFIER

As described above, memory protection is established through run-time checks that are introduced by rewriting the binaries produced by a cross-compiler toolchain. The routines that implement the checks are located in the trusted domain. The rewriter introduces calls and jumps that invoke the runtime checks from the sandboxed binary. A verifier inspects the sandboxed binary to ensure that sufficient checks have been introduced to prevent any possible protection violation. The verifier and rewriter are completely independent, and the node that executes a sandboxed binary needs to trust only the verifier. The verifier could be a desktop application that, for example, cryptographically signed binaries, or a trusted component running on the sensor node. Our verifier is a trusted component running on every sensor node that verifies sandboxed binaries locally prior to admitting them for execution.[1] An attractive feature of this architecture is that sensor nodes do not need to

[1]Sandboxed binaries can be seen as an example of proof-carrying code (PCC) [15], even though they do not include any logical proofs.

trust any external component. We discuss the tradeoffs involved in the design of the verifier in Section 8.

6.1 Sandboxed Operations

Five types of operations are protected. First, all forms of store instructions are sandboxed by a call to the memory map checker. Second, all function call entry points are instrumented with calls to the function entry stub, which uses the safe stack. Third, all return instructions are redirected to the function exit stub. Fourth, all computed calls are redirected to a stub that checks if the destination is in the jump table. Fifth, the beginning of all the basic blocks are marked by a special `NOP` symbol. An example of the sequence of instructions introduced by the rewriter for the AVR architecture is shown in Figure 7. The sequence is re-entrant; it can be preempted by interrupts. The rewriter performs a basic block analysis of the binary and preserves the program's original control flow by updating static jump, call, and branch targets.

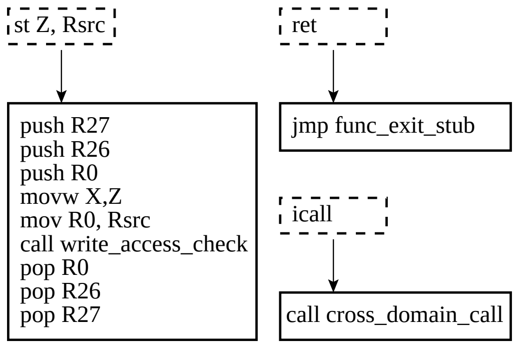

Figure 7: Inline Checks

The rewriter reads ELF object files. It uses symbol table information to distinguish binary data representing code and constants. Upon sandboxing, the rewriter outputs a new ELF object file. The symbol table and the relocation records in the output ELF file are suitably modified to reflect their updated positions within the sandboxed code. Since linkers such as `avr-ld` link object files in ELF format, the binary rewriter can be used to sandbox only portions of the complete binary. For example, only the device drivers in the final image of an operating system can be sandboxed before installing them on a sensor node.

6.2 Verifier

The verifier is a very simple program that maintains no state. In Harbor, all potentially unsafe operations, such as store to memory, returns, and computed calls, are performed within the run-time checkers. Therefore, the verifier performs a single pass over the entire instruction sequence and raises an exception if it encounters any of the potentially unsafe operations. It checks all static jump, call, and branch targets to ensure that they are within domain boundaries. It also ensures that these targets point to valid instructions. This check is necessary because AVR ISA has multi-word instructions. If control flow jumped into the middle of a multi-word instruction, run-time checks might be circumvented. Therefore, the rewriter marks the beginning of basic blocks with a special `NOP` symbol. The verifier checks that all static jump, call, and branch targets point to a `NOP`, and checks that the `NOP` symbol never appears in the middle of a multi-word instruction. Function entry points (determined from call instruction targets) are checked to ensure that they store the return addresses on the safe stack. Finally, the verifier does not permit store instructions that write to program memory. The verifier calls an exception handler if any of these properties are violated. Its total line count is only 211 lines.

7. EVALUATION

In this section, we analyze the protection benefits and overheads introduced by Harbor's protection mechanisms.

7.1 Overhead Microbenchmarks

We first present microbenchmarks that measure Harbor's CPU overhead. Overhead was measured using Avrora [21], a cycle accurate node and network simulator for the Mica family of sensor nodes. Measurements were averaged across multiple application scenarios.

Function Name	Cost
Write access check	65 cyc
Cross domain call	65 cyc
Cross domain return	28 cyc
Function entry stub	38 cyc
Function exit stub	38 cyc

Table 3: CPU Overhead of Memory Protection Routines

Table 3 summarizes the results for all of Harbor's protection primitives. All these routines are implemented in assembly for optimizing performance. The registers used in these routines are saved to the run-time stack. Many CPU cycles are spent in push and pop operations. This overhead could be significantly reduced by dedicating one or more registers for Harbor's exclusive use; the cross-compiler would be directed to ignore these registers entirely. However, `avr-gcc` does not have stable support for dedicated registers. Overhead is also introduced during dynamic memory allocation, deallocation, and transfer, since the memory map must be updated. Table 4 compares the overhead of memory allocation routines in the presence and absence of the protection mechanism. The overhead depends upon the size of memory block that is being allocated, freed or transferred. The average size of the memory block used by all the three operations in our experiments was 16 bytes. The numbers in Table 4 are an average measurement of the execution time obtained from a long running simulation of the Surge application [23]. The relatively higher overhead of `ker_change_own` and `ker_free` calls is due to additional checks introduced to prevent illegal ownership transfer or freeing of memory blocks by non-owners.

Function Name	Normal	Protected
`ker_malloc`	343 cyc	610 cyc
`ker_free`	138 cyc	425 cyc
`ker_change_own`	55 cyc	365 cyc

Table 4: CPU Overhead for Dynamic Memory Calls

7.2 Resource Utilization Microbenchmarks

Increases in code and data memory utilization due to Harbor's protection mechanisms are shown in Table 5. Code memory usage increases by about 15% in a protected kernel relative to an unprotected kernel. This increase is mainly due to the memory map and cross domain call jump table mechanisms. There is no significant change in program memory usage going from two protection domains to multiple protection domains. Data memory usage increases relative to an unprotected kernel by at most 5% and 9.5% in

2-domain and 8-domain systems, respectively. The main culprit is the memory map, which takes 128 and 256 bytes in 2- and 8-domain systems. (There are 28 additional bytes of constant overhead.) This is the maximum possible overhead, as this memory map configuration stores layout and ownership information for the entire address space. By modifying data layout, the portion of address space that requires a memory map can be reduced. For example, in SOS, memory map is needed only for the heap and the safe stack; by abutting these data structures, the memory map can be reduced to 70 or 140 bytes for 2- and 8-domain protection, respectively.

Memory	Raw	Δ (2 domains)		Δ (8 domains)	
Flash	41796 B	+6146 B	+14.7%	+6228 B	+14.9%
RAM (Max)	2892 B	+148 B	+5.1%	+276 B	+9.5%
RAM (Min)	2892 B	+98 B	+3.4%	+168 B	+5.8%

Table 5: Code and Memory Overhead for Blank SOS Kernel for Mica2

Module	Raw	Δ (2 or 8 domains)	
Blink	150 B	+48 B	+32%
Tree Routing	2820 B	+1658 B	+59%
Surge	542 B	+350 B	+65%
Outlier Det.	1312 B	+738 B	+56%
DVM	13072 B	+6652 B	+51%
FFT	3016 B	+894 B	+30%

Table 6: Code Size Increase of Mica2 SOS Modules

Finally, we evaluate the relative increase in size of modules due to introduction of checks by the binary rewriter. As noted in Table 6, there is a significant increase in the relative code size of sandboxed binaries as compared to raw binaries. This is mainly caused due rewriting all store instructions to a long sequence of instructions that call the memory map checker (Figure 7). This overhead could be significantly reduced by using dedicated registers to eliminate all push/pop instructions in the sequence, and/or by adding static analysis to eliminate some redundant checks.

7.3 Relative Application Performance

In this subsection, we measure Harbor's performance impact. Many sensor network applications are heavily duty-cycled and therefore not CPU intensive. However, some are not, and for our first benchmark, we choose a CPU intensive Fast Fourier Transform (FFT). This should present a realistic idea of Harbor's costs for challenging applications. The FFT module receives a buffer of samples represented as 16-bit fixed-point integers (we do not measure the cycles required to obtain the samples). It transforms the samples in place and outputs results to the same buffer. As shown in Table 7, FFT takes 3.6 ms to execute in normal mode and 17.3 ms in the protected mode. This gives Harbor a slowdown factor of 4.8.

The next experiment is another challenging application, an outlier detector. The outlier detector samples a set of sensor values and stores them in a buffer. Once the buffer is filled, it computes the distance between all pairs of samples in the buffer and stores the result in a matrix. Using a distance threshold, the algorithm marks the distance measurements in the matrix that are greater than the threshold. If the majority of the distance measurements for a sensor readings are marked, then the sensor reading is classified as an outlier. This application is memory write intensive and the matrix operations are easily prone to buffer overflow errors.

We consider two systems that can protect against such errors, Harbor and the Dynamic Virtual Machine (DVM) [2], an extensi-

Module	Time (ms)	Slowdown
FFT	3.6	—
FFT-Harbor	17.3	4.8
Outlier detector	0.18	—
Outlier-Harbor	1.47	7.9
Outlier-DVM	102.4	554.5
Outlier-DVM-Harbor	268.3	1453.0
Sort with Maté VM script [11]		115

Table 7: Relative Performance of Applications

ble domain-specific interpreter that performs a bounds check on every write. DVM's bounds checks are in some ways more stringent than Harbor's protection, since they also prevent scripts from corrupting *their own* memory. However, errors in DVM's native code implementation might corrupt any memory on the node. We implemented the outlier detector as a DVM script that is interpreted on sensor sampling timer events. The execution time of the script was measured to be 102.9 ms. However, some of this time is also spent within the kernel to actually sample the sensor data. The sampling time was measured to be 0.49 ms, so the script's true execution time is 102.41 ms. This is over 500 times longer than an outlier detector implemented as a raw binary, as Table 7 shows. The high overhead is due to DVM interpretation. The outlier detector script uses many low-level operations. Prior work has shown that this kind of script has high overhead; for example, a Maté script that sorts an array using low-level operations takes 115 times longer than a script that sorts an array with a single "sort" operation [11]. While DVM's overhead could be reduced by adding high-level instructions that perform complex operations in native code, these high-level instructions might themselves contain bugs. Harbor's protection is much less expensive than interpretation overhead. Sandboxing slows down the native code implementation by a factor of 7.9. Perhaps more surprising, sandboxing DVM slows it down only by an additional factor of 2.6. Here, DVM provides fine grained protection (at the level of individual memory objects) to scripts and Harbor provides coarse grained protection (at the level of domains) to the system running DVM.

A final challenging, write-intensive application is a simple buffer writer, which allocates a memory block and completely fills its contents with arbitrary data. This application simulates a very common behavior of sensor network applications, namely copying sampled sensor data into a buffer that can be transmitted into the network. A common programming mistake in such applications is to overflow the buffer. Figure 8 plots execution time of the buffer writer on DVM, Harbor, and native SOS for varying buffer sizes. The average slowdown factor of Harbor relative to native SOS for this application was 13.3. The average slowdown factor of DVM relative to native SOS was about 1200.

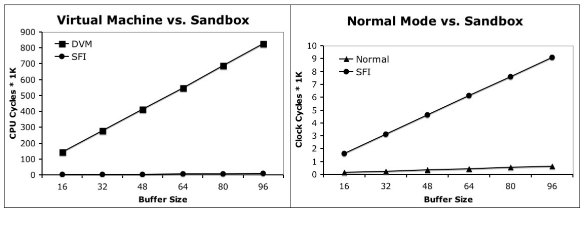

Figure 8: Buffer Writer Performance Overhead

We next consider a typical duty-cycled sensor network application, namely data gathering through a collection tree. Our experiment setup was a simulated network of 5 Mica2 motes arranged in a linear topology to form a two-hop network to a base station. All nodes were installed with an SOS kernel with 8-domain Harbor protection. When the network was up, the two-module data collection application was installed on all nodes. First, a tree building and maintenance module [23] was distributed; then the Surge module, which periodically samples light sensor data and sends it to the base station via the collection tree, was installed. The two modules were installed in different protection domains. Harbor's impact on the complete application was measured by profiling CPU active time in the Avrora simulator. CPU active time was observed to be 8.41% and 8.56% over a duration of 30 minutes for normal and protected mode operation, respectively. The relative increase in CPU utilization for a protected system is thus only 1.85% compared to an unprotected system. In most sensor network applications, absolute CPU utilization is even lower [7]. The increased overhead is a small price to pay for the improved reliability provided by the software-based memory protection.

7.4 Experience

Harbor has been in use in SOS for several months, and has discovered two memory corruption faults in application modules that had been in active use for several months previously. The first error was discovered while executing the data collection application on a SOS kernel with 2-domain Harbor protection. A programming error in the Surge module triggered an invalid memory access exception. Figure 9 demonstrates the error.

```
// Size of routing header
hdr_size = SOS_CALL(s->get_hdr_size, proto);
// Using return value without checking
s->smsg = (SurgeMsg*)(pkt + hdr_size);
// Memory Corruption
s->smsg->type = SURGE_TYPE_SENSORREADING;
```

Figure 9: Programming error in Surge module

The Surge module invokes a dynamic function call [8] to the tree routing module to determine the size of the routing header. Dynamic function calls are linked at run time, and fail with an error code if the function provider is absent. The code above fails to check whether `hdr_size` is an error code. Nodes where the Surge module was installed before the tree routing module would use an negative value for the routing header size and thereby corrupt memory—specifically, heap metadata, which effectively resides in the kernel domain. Harbor detected this error and killed the offending application.

A second programming error was discovered in the tree routing module with 8-domain protection. There were three active domains in the system: the kernel, the tree routing module, and the buffer writer module. The SOS kernel tracks the ownership of a message payload as it is passed from a source module to its destination module. If the source module wishes to relinquish ownership, it sets a release flag while posting the message. A destination module that wishes to write into the message payload is *required* to gain ownership through a `sys_msg_take_data` system call. If the source module set the release flag, this system call is effectively a no-op. If the source module did not set the flag, however, then the system call makes a private copy of the payload owned by the destination module. Failure to call `sys_msg_take_data` could corrupt the source module's memory. The buffer writer module was not releasing the buffer, but the tree routing module was not calling `sys_msg_take_data`. Harbor discovered this error when the tree routing module tried to overwrite the message payload.

The errors described in this section occur under rare conditions and are hard to detect during software testing. However, the impact of these errors could be severe. A system that can guarantee memory protection is indispensable for building robust embedded software.

8. DESIGN ALTERNATIVES

Harbor provides a number of design knobs that allow systems to trade off protection and resource utilization. First, the memory map data structure is configured by changing the number of bits stored per block to match the number of domains required by a system. For example, four bits per block support up to eight protection domains. We have found eight domains to be sufficient for most sensor network systems using SOS. Two bits per block can create a two-domain system (a user/kernel model). Increasing the number of bits stored per block increases the size of the memory map.

Second, there is a tradeoff between the size of the memory map and the amount of memory fragmentation. Larger block size leads to increased internal fragmentation, but reduces the number of blocks and thereby the size of the memory map. For example, a block size of 256 bytes is suitable for the large image and matrix objects passed around in the Cyclops imager [17]. Mica2 based modules use a block size of 8 bytes. Harbor and SOS currently support a single fixed block size for the entire heap; an extension might permit using different block granularities in different memory regions.

Third, the memory map can be configured to track ownership and layout in only a subset of the entire address space, reducing its space overhead. For example, in SOS the memory map tracks only the heap and safe stack. In general, the size of the memory map can be reduced by decreasing the fraction of memory reserved for the heap.

The Harbor design also allows trading off execution overhead, and code size increase, against the complexity of the verifier. Performance and code size increase are directly proportional to the number of operations that are sandboxed. The current implementation sandboxes *all* unsafe operations. This severely penalizes performance and code size but reduces the complexity of the verifier, which requires only a single pass over the entire binary and maintains no additional state. Static analysis on the binary can reduce the number of operations sandboxed by adding single checks that safely protect a series of potentially unsafe operations. However, the safety of such a check is harder to verify using a simple verifier. We are currently exploring this design space to develop a rewriter–verifier combination that consumes limited resources but improves performance, and reduces code size, of sandboxed binaries.

9. CONCLUSION

We have explored the challenges in providing software based memory protection through sandboxing in resource constrained embedded sensor nodes. Though we have implemented the protection technology in the SOS operating system, our general approach is applicable elsewhere. As discussed in Section 8, there is a large space of protection architectures that can be designed using Harbor components. A detailed exploration of the design space with an evaluation of the various overheads is necessary to pick the most appropriate operating point for a given system. We believe that a complete system for memory protection would require combination of two or more software based approaches. Of particular interest is a system composed of ASVM with memory safe extensions. Finally, we are also exploring low-cost extensions to micro-controller architecture to provide memory protection for embedded software. We envision that the application of these techniques will create robust software that would enable long term sensor network deployments.

ACKNOWLEDGEMENTS

We are grateful to Simon Han, Roy Shea, and the other members of the SOS team at UCLA and elsewhere for discussions and bug reports. We also thank the anonymous reviewers for their perceptive comments.

REFERENCES

[1] ARM7TDMI Technical Reference Manual. http://www.arm.com/pdfs/DDI0210C_7tdmi_r4p1_trm.pdf.

[2] R. Balani, C.-C. Han, R. Kumar Rengaswamy, I. Tsigkogiannis, and M. Srivastava. Multi-level software reconfiguration for sensor networks. In *EMSOFT '06: Proc. 6th ACM & IEEE International Conference on Embedded Software*, 2006.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *SOSP '95: Proc. 15th ACM Symposium on Operating Systems Principles*, 1995.

[4] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *SenSys '06: Proc. 4th ACM Conference on Embedded Networked Sensor Systems*, 2006.

[5] P. Dutta, M. Grimmer, A. Arora, S. Bibyk, and D. Culler. Design of a wireless sensor network platform for detecting rare, random, and ephemeral events. In *IPSN '05: Proc. 4th International Symposium on Information Processing in Sensor Networks*, 2005.

[6] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *OSDI '06: Proc. 7th USENIX Symposium on Operating System Design and Implementation*, 2006.

[7] L. Gu and J. A. Stankovic. *t-kernel*: Providing reliable OS support to wireless sensor networks. In *SenSys '06: Proc. 4th ACM Conference on Embedded Networked Sensor Systems*, 2006.

[8] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. SOS: A dynamic operating system for sensor networks. In *MobiSys '05: Proc. 3rd International Conference on Mobile Systems, Applications, and Services*, 2005.

[9] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *SOSP '97: Proc. 16th ACM Symposium on Operating Systems Principles*, 1997.

[10] L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, N. Kushalnagar, L. Nachman, and M. Yarvis. Design and deployment of industrial sensor networks: Experiences from the North Sea and a semiconductor plant. In *SenSys '05: Proc. 3rd ACM Conference on Embedded Networked Sensor Systems*, 2005.

[11] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *NSDI '05: Proc. 2nd Symposium on Networked Systems Design and Implementation*, 2005.

[12] P. Levis, D. Gay, V. Handziski, J. H. Hauer, B. Greenstein, M. Turon, J. Hui, K. Klues, C. Sharp, R. Szewczyk, J. Polastre, P. Buonadonna, L. Nachman, G. Tolle, D. Culler, and A. Wolisz. T2: A second generation OS for embedded sensor networks. Technical Report TKN-05-007, Telecommunication Networks Group, Technische Universität Berlin, 2005.

[13] K. Lorincz, D. Malan, T. R. F. Fulford-Jones, A. Nawoj, A. Clavel, V. Shnayder, G. Mainland, S. Moulton, and M. Welsh. Sensor networks for emergency response: Challenges and opportunities. *IEEE Pervasive Computing, Special Issue on Pervasive Computing for First Response*, 3(4):16–23, Oct. 2004.

[14] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proc. 15th USENIX Security Symposium*, Aug. 2006.

[15] G. C. Necula and P. Lee. Proof-carrying code. In *POPL '97: Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997.

[16] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *POPL '02: Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[17] M. Rahimi, D. Estrin, R. Baer, H. Uyeno, and J. Warrior. Cyclops, image sensing and interpretation in wireless networks. In *SenSys '04: Proc. 2nd ACM Conference on Embedded Networked Sensor Systems*, New York, NY, USA, 2004. ACM Press.

[18] J. Regehr, N. Cooprider, W. Archer, and E. Eide. Memory safety and untrusted extensions for TinyOS. Technical Report UUCS-06-007, School of Computing, University of Utah, 2006.

[19] M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *SOSP '03: Proc. 19th ACM Symposium on Operating Systems Principles*, 2003.

[20] B. L. Titzer. Virgil: Objects on the head of a pin. In *OOPSLA '06: Proc. 21st ACM SIGPLAN Conference on Object-Oriented Systems, Languages, and Applications*, 2006.

[21] B. L. Titzer, D. Lee, and J. Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *IPSN '05: Proc. 4th International Symposium on Information Processing in Sensor Networks*, 2005.

[22] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP '93: Proc. 14th ACM Symposium on Operating Systems Principles*, pages 203–216, New York, NY, USA, 1993.

[23] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *SenSys '03: Proc. 1st ACM Conference on Embedded Networked Sensor Systems*, 2003.

[24] Zigbee Consortium. www.zigbee.com.