Software-Based Memory Protection In Sensor Nodes

Ram Kumar, Eddie Kohler and Mani Srivastava, UCLA
{*ram, mbs*}*@ee.ucla.edu, kohler@cs.ucla.edu*

ABSTRACT

Typical sensor nodes are resource constrained micro-controllers containing user level applications, operating system components, and device drivers in a single address space, with no form of memory protection. A programming error in an application can easily corrupt the state of the operating system and other software components on the node. In this paper, we propose a memory protection scheme that prevents the corruption of operating system state by buggy applications. We use sandboxing to restrict application memory accesses within the address space. Severe resource constraints on the sensor node present interesting challenges in designing a sandbox for user applications. We have implemented and tested our scheme on the SOS operating system. Our experiments were able to detect a memory corruption bug in an application module that had been in use for several months.

1 INTRODUCTION

Sensor networks are permeating many industrial, commercial, and medical applications. For example, Code-blue [1] is a prototype medical sensor network platform that is used for expediting the triage process during disaster response. A network of 4000 sensors deployed by Intel [2] in a semiconductor fabrication plant performs predictive maintenance of machinery in service. The Zigbee consortium [3] is seeking to equip lighting and HVAC controllers with wireless radios to enable intelligent building automation and security services. These current and upcoming sensor network deployments require a high availability infrastructure with the ability to support multiple users. Unexpected failures could cause anything from financial losses to life-critical damages. However, current software technology is grossly inadequate for such long term deployments. Bugs in any part of the software can easily bring down the entire network. In particular, corruption of memory due to the lack of protection from buggy applications can crash or freeze the node, or corrupt sensed data.We argue that memory protection is a vital enabling technology for creating reliable and long-lasting sensor network software systems.

Most sensor nodes have a very simple architecture, and lack features, such as memory management units (MMU) and privileged-mode execution, used in desktop/server class systems to isolate or protect the data and code of one program from another. The micro-controllers used in sensor nodes typically have separate memories for program and data storage. The entire data memory of the sensor node is accessible to all the programs running on the node via a single address space.

However, sensor software is quite complex. This complexity arises mainly from the need to support diverse sensors, multiple distributed middleware services, dynamic code updates, and concurrent applications.Implementing the software components presents a tough challenge. Programmers have to deal with severe resource constraints and concurrency issues. Furthermore, there is very limited debugging support on the sensor node hardware. Therefore, programming errors are quite common. The impact of these errors can be severe.

Virtual machines for sensor nodes, such as ASVM [4], ensure that programming errors in high-level scripts cause no harm. However, individual script instructions are executed as native code which could be buggy.

In this paper, we present and evaluate a technique for providing software-based memory protection in resource constrained embedded sensor nodes. Our approach partitions the software components installed on a sensor node into kernel and user modules. The primary goal of our scheme is to protect the memory of the kernel modules from being corrupted by the user modules. We achieve memory protection by re-writing the source code of user modules to enforce restrictions on memory accesses. This class of technique was first proposed by Wahbe et. all [5] and is known as "software-based fault isolation" (SFI) or "sandboxing".

We investigate the challenges in implementing SFI on resource constrained embedded sensor nodes. The limited address space on the motes precludes the partitioning of the address space into contiguous kernel and user domains. Scarce memory resources require the scheme to have a very small memory footprint. Limited computational capabilities also require a reasonably small CPU overhead. In our approach, we maintain a fine-grained memory map of the address space containing the access permissions of the memory regions. A run-time checker intercepts all write operations made by a user module and restricts accesses based upon the permissions encoded in the memory map.

We have implemented and evaluated the software-based memory protection mechanism on the SOS operating system [6] although our work is also applicable other OSes. The implementation on an embedded platform is non-trivial. However, the benefits became immediately apparent when our scheme was able to detect memory corruption in a data collection (Surge) application module that had been in use for several months. 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, data was written to an incorrect memory location, causing some nodes to crash. Our scheme was successfully able to prevent the corruption and signal the invalid access.

In the rest of the paper, we will describe the design of the memory protection scheme in detail. Section 2 provides a brief background on the SOS operating system. The protection architecture is described in Section 3. The overhead of our scheme is evaluated in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2 Background: SOS

Unique resource tradeoffs in sensor nodes, the distributed and collaborative nature of applications and, remote unattended operation of sensor networks motivated a new class of operating systems and run-times. TinyOS [7], the most popular operating system for sensor networks, uses reusable software components [8] to implement common services, but each node runs a statically linked system image. ASVM [4], an application specific virtual machine on top of TinyOS, provides limited flexibility to re-task a deployed network using high-level scripts; script instructions are implemented by machine code. SOS [6] has a more traditional architecture, including a kernel installed on all nodes. The rest of the system and application level functionality is implemented by a set of dynamically loadable binary modules. Our memory protection mechanism is implemented for SOS specifically, although it could be applied to other run-times as well (Section 6). This brief background is useful to fully understand the design and implementation of our scheme.

The SOS kernel is well tested and assumed to be free of programming errors. Modules are position independent binaries that implement a specific task or function. Modules operate on their own state, which is dynamically allocated at run-time. An application in SOS is composed of one or more modules, such as routing protocols, sensor drivers and applications, interacting via asynchronous messages or function calls.

The SOS kernel supports dynamic memory allocation. Dynamic memory is used to store module state and messages.Memory is allocated using a block-based first-fit scheme to minimize the overhead of the allocation process. The dynamic memory region is shared by the SOS kernel and the modules. The SOS kernel tracks ownership of each block of memory. Ownership can also be transferred, enabling easy movement of buffers through the system.

3 Protection Architecture

3.1 User Fault Domain

We focus on memory corruption faults caused by programming errors in the user modules. A wild write made by a user module can easily corrupt operating system state and trigger a severe failure condition. Our fault model for memory protection is to prevent corruption of kernel state caused by illegal write operations made by a user module. We create and enforce a *user fault domain* within the data memory address space of the sensor node. The user fault domain refers to a fragmented but logically distinct portion of the overall data memory address space (Figure 1). The state belonging to the user modules resides entirely within the fault domain. No assumption is made about the layout of the state belonging to the individual applications within the fault domain. The kernel state resides entirely outside the user fault domain. Modules are restricted from writing to the memory outside the user fault domain through run-time checks.

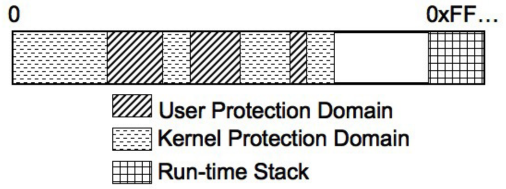

Figure 1—Kernel and User Protection Domains

The protection model based on a user fault domain does not address all possible memory corruption faults in the system; user modules can still corrupt each other's memory.This form of corruption, though undesirable, is less serious than kernel corruption.A stable kernel can always ensure a clean re-start of user modules when corruption is detected. On the other hand, a corrupted kernel has unpredictable behavior, leaving complete system reboot through a watchdog or grenade timer as the only possible means of recovery [9].

Creating and enforcing a user fault domain is a challenging task on embedded platforms. The total available address space is only 4 KB. Limited memory prohibits prevents contiguous partitioning of the address space into kernel and user domains, since this would severely limit memory available to applications. We designed the user fault domain with three requirements: low CPU overhead, small memory footprint, and no design constraints for user applications. A memory map data structure satisfies these requirements.

3.2 Memory Map Manager

A memory map specifies the permissions value for every block of the address space, where a block is a small, contiguous, chunk of memory. The main operation of the memory map is to to find the access permissions for a given address. Its design goal is to balance the lookup efficiency with the extra storage required for the table. The memory map specifies two pieces of information. First, it contains the ownership (user or kernel) for every block of memory. Second, it encodes information about the memory layout such as the start of a segment.The actual encoded information and its meaning is specified in Table 1.

Code	Meaning
00	Free or Start of Kernel Allocated Segment
01	Later portion of Kernel Allocated Segment
10	Start of User Allocated Segment
11	Later portion of User Allocated Segment

Table 1—Encoded information in the memory map table

The mapping from address to memory map ownership information is shown in Figure 2. Assuming a block size of 8 bytes[1], the last three bits of the address are the offset into a given block. The remaining bits represent the block number in the data memory. Block permissions are packed into a byte; each byte contains information for four contiguous memory blocks. Therefore, the last two bits of the block number represent the bit offset of the permission, and the remaining bits represent the index into the memory map table. This particular design of the memory map table was chosen to minimize memory footprint.

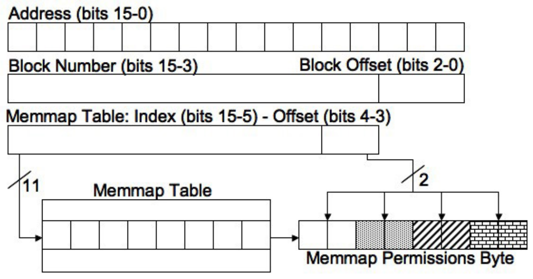

Figure 2—How an address indexes the memory map

The memory map manager tracks permission information for every block in the address space.The memory map is initialized such that all the statically allocated kernel memory blocks (containing kernel globals) are marked as owned by the kernel, since user modules never read or write to them. The remaining portion of the address space is partitioned into a heap and a stack. The heap is divided into blocks; the dynamic memory allocation API allocates segments , or contiguous block ranges, from the heap. The smallest allocated segment can be a single block. The memory map for the heap is initially marked as “Free”. The stack frames are not guaranteed to lie on block boundaries and therefore there is no memory map for the stack. The memory map itself is protected from wild writes as it is a part of the kernel state.

[1]Our implementation on AVR uses a block size of 8 bytes

The memory map manager works closely with the dynamic memory manager in the SOS operating system. Any request for dynamic memory is passed to the kernel’s memory map manager which, sets the correct permissions for the set of allocated blocks. During the free operation, the memory map manager automatically clears the permissions. The dynamic memory manager in SOS permits ownership to change for dynamically allocated memory blocks. The memory map manager tracks any changes to the permissions that are caused due to the ownership transfer of a set of memory blocks.

3.3 Run-time Checker

A run-time checker restricts the memory access of user modules to permissible regions based on the memory map. The policy used for access control can vary; our current policy prevents user modules from writing to memory regions that are owned by the kernel. The modules are instrumented to introduce checks before every write operation that needs protection. The pseudo-code for performing the write access checks in shown in Figure 3.

```
WRITE_ACCESS_CHECK(addr_t addr){
  if (addr < STACK_PTR){

   // Retrieve permissions byte
   uint16_t  mmap_index = (addr >> 5);
   uint8_t   perms = MEM_MAP_PERMS_TBL[mmap_index];

   // Generate bit mask
   uint8_t   mmap_offset = (addr & 0x1f);
   // uint8_t perms_bm = (BLOCK_TYPE_BM << ((mmap_offset >> 3) << 1));
   uint8_t   perms_bm = MEM_MAP_BM_LUT[mmap_offset];

   // Check validity
   if  !(perms & perms_bm) mem_access_violation();
  }
}
```

Figure 3—Pseudo-code for run-time checker

The write access checker performs three operations. First, it retrieves the ownership permissions byte for a given address from the memory map table. Second, it generates a bit mask from the address offset to derive the actual permission. Third, it checks the permission and signals a memory access violation for invalid operations. The generation of the bit mask requires bit shift operations. These operations took 32 clock cycles on the Atmega128L, which have no instruction level support for arbitrary bit shifts. Therefore, a lookup table is used; this is stored in flash memory to minimize data memory utilization. The organization of the lookup table and its operation is described in Figure 4. The lookup table takes only 8 clock cycles. Write accesses to the stack are not subject to any checks. Stack range is determined by reading the stack pointer register.

During module unloading, the kernel uses ownership information to free all the memory owned by that module. The ownership information and the size of the seg-

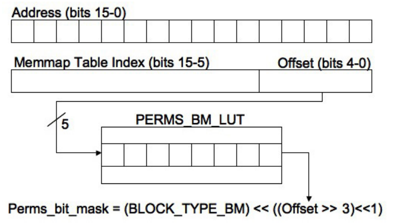
Address (bits 15-0)
Memmap Table Index (bits 15-5)
Offset (bits 4-0)
5
PERMS_BM_LUT
Perms_bit_mask = (BLOCK_TYPE_BM) << ((Offset >> 3)<<1)

Figure 4—Lookup table optimization to simulate bit-shift operations

ment are stored as meta-data within the first memory block of a segment, as shown in Figure 5. This organization enables the kernel to efficiently map a memory location to its owner. However, the user modules can overwrite the meta-data as it lies within the block boundary. Therefore, the run-time checker is modified such that the meta-data in the first block of any segment is protected from writes. Note that the memory map table maintains information about the starting block of any segment. The additional checks are also implemented using a look-up table for improved efficiency.

```
typedef struct _Block
{
  uint16_t segmentSize; // Blocks in current segment
  uint8_t owner; // Identity of the segment owner
  union
  {
    uint8_t userPart[BLOCK_SIZE - sizeof(uint16_t)];
    struct
    {
      struct _Block *prev; // Doubly linked free-list
      struct _Block *next; // Doubly linked free-list
    };
  } ;
} Block;
```

Figure 5—Block implementation in SOS kernel

All the checks are currently introduced by modifying the source code of the user modules. The CIL (C Intermediate Language) [10] framework catches all the writes made by the user module and inserts the appropriate write access check. In future, a binary rewrite tool will analyze the code to introduce fewer checks. The binary rewrites would be performed at load time of modules into the sensor network.

4 Evaluation

The main objective of our evaluation was to determine the overhead introduced by the protection mechanism. The experiment setup was a network of 5 Mica2 motes, arranged in a linear topology to form a two hop network to the basestation. All the nodes were installed with an image of the SOS kernel with the protection features enabled. A data collection application comprising two modules was installed on all the nodes in the network. First, a tree building and maintenance module [11] was distributed to all the entire network. Next, the Surge module was installed that periodically samples light sensor data and sends it to the basestation via the collection tree. We first present micro-benchmarks that measure the CPU overhead introduced by the protection mechanism. The computation overhead of the micro-benchmarks was measured using Avrora [12], a cycle accurate node and network simulator for the Mica family of sensor nodes. The measurements were averaged over multiple iterations spanning different nodes in the network.

Function Name	Normal	Protected
ker_malloc	343	661
ker_free	138	467
ker_change_own	55	285

Table 2—Overhead (CPU cycles) of memory allocation routines

Table 2 compares the overhead of memory allocation routines in the presence and absence of the protection mechanism. The overhead is mainly due to the setting of the appropriate permissions fields in the memory map table. Furthermore, all the calls were enhanced with extra checks. For example, `ker_change_own` does not permit a user module to take ownership of a memory block owned by the kernel. The run-time memory access checking routine has an overhead of 66 clock cycles. The increased code and data memory due to the protection mechanism are shown in Table 3. The difference in data memory is due to the memory map table, which occupies 86 bytes of RAM.

Memory Section	Normal	Protected
FLASH	38470 B	39526 B
RAM	2827 B	2741 B

Table 3—Code and Memory Size Overhead for Mica2 platform

The impact of these overheads on the complete application was measured by profiling the active time of the CPU in the Avrora simulator. A total of 40 memory write operations were instrumented in the two modules. The CPU active time was observed to be 6.48% and 6.64% over a duration of 30 minutes for the normal and protected mode operation respectively. The absolute difference in the CPU utilization is only 0.16% when the check operation was invoked 4410 times. The difference in the overheads can be further reduced by using standard compiler optimizations to invoke fewer checks. The increased overhead is a small price to pay for the improved reliability provided by the software-based memory protection.

The protection mechanism was able to detect a programming error in the Surge module. Surge module invokes a dynamic function call to the Tree Routing module to determine the size of the routing header. Dynamic function calls are linked at run-time and they fail with an error code if the function provider is not present. The error code returned by the function was not being checked in the Surge module implementation. Nodes where the Surge module was installed before the Tree Routing module would use an arbitrary value for the size of the routing header and thereby corrupt the memory.

The run-time checker detected the occurrence of the illegal write and prevented memory corruption.

5 Related Work

As sensor networks are being envisioned for long-term deployments, there is an emerging interest to address reliability as primary design concern [13, 9]. Sympathy, a debugging framework, has focussed on developing network-level protocols to diagnose/localize problems [13] At present, high dependability is generally simulated by node reboot [9]. Our approach aims to provide memory protection as an enabling technology for building high availability sensor networks. In this aspect, our work relates primarily to efforts that isolate independent software components from corrupting each other's state, due to program bugs.

Type-safe languages such as JAVA and ML can flag illegal accesses at compile time. However, their run-time support incurs high overhead. Therefore, most of the code on embedded platforms is written in unsafe languages such as C or assembly. Languages such as NesC [8] contain minimal extensions to C (such as the `atomic` keyword) to remove race conditions, but do not address memory corruption.

Run-time techniques such as Software Fault Isolation (SFI) [5] have been suggested for desktop/server systems. SFI enforces a static partitioning on an application's address space to enable safe sharing of the address space by multiple cooperative modules. Protecting modules from each other would be a natural extension of our work, requiring different space/time tradeoffs. Our work is also related to numerous OS extensions to address isolation among distrusted modules. Nooks [14], for example, separates modules into lightweight protection domains by managing separate page-tables for each module.

Hardware-assisted protection is vastly popular on standard computing platforms [15]. Mondrian Memory Protection (MMP) [15] inspects memory accesses at the instruction level from within the processor pipeline to provide word-level protection. It uses fairly complex and expensive hardware extensions to reduce overhead of monitoring all accesses. Sensor-class nodes, however, lack such hardware, soliciting simpler, software-based techniques.

6 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 to other run-times such as TinyOS and ASVM. TinyOS applications, particularly those using dynamic memory [16], might benefit from this infrastructure by partitioning the components into user and system at compile time. ASVM envisions building new instructions as needed. Our work could ensure that these new instructions do not destabilize the system. Therefore, memory protection is a general enabling technology for long term sensor network deployments. In future, we plan to evaluate our scheme on real world applications to better understand the trade-offs involved.

References

[1] Konrad Lorincz et. al. Sensor networks for emergency response: Challenges and opportunities. In *In IEEE Pervasive Computing, Special Issue on Pervasive Computing for First Response*, Oct-Dec 2004.

[2] Robert Adler et. al. Design and deployment of industrial sensor networks: Experiences from the north sea and a semiconductor plant. In *Third ACM Conference on Embedded Networked Sensor Systems (SenSys)*, November 2-4, 2005.

[3] Zigbee consortioum.

[4] Phil Levis, David Gay, and David Culler. Active sensor networks. In *Proceedings of the second international conference on Networked Systems Design and Implementation (NSDI)*, 2005.

[5] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, New York, NY, USA, 1993. ACM Press.

[6] Chih Cheh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. Sos: A dynamic operating system for sensor networks. In *Third International Conference on Mobile Systems, Applications, And Services (Mobisys)*, 2005.

[7] Phil Levis et. al. T2: A second generation os for embedded sensor networks. Technical report, University of California, Berkeley, 2005.

[8] David Gay, Philip Levis, Robert von Behren, and Matt Welsh. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation*, 2003.

[9] Prabal Dutta, Mike Grimmer, Anish Arora, Steve Bibyk, and David Culler. Design of a wireless sensor network platform for detecting rare, random, and ephemeral events. In *The Fourth International Conference on Information Processing in Sensor Networks (IPSN)*, 2005.

[10] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, 2002.

[11] Alec Woo, Terence Tong, and David Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *First ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2003.

[12] Ben L. Titzer, Daniel Lee, and Jens Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks (ISPN)*, 2005.

[13] Nithya Ramanathan, Kevin Chang, Rahul Kapur, Lewis Girod, Eddie Kohler, and Deborah Estrin. Sympathy for the sensor network debugger. In *Third ACM Conference on Embedded Networked Sensor Systems (SenSys)*, November 2-4, 2005.

[14] Michael Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *ACM Transactions on Computer Systems*, volume 23, 2005.

[15] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[16] Ben Greenstein, Eddie Kohler, and Deborah Estrin. A sensor network application construction kit (snack). In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 69–80, New York, NY, USA, 2004. ACM Press.