TOWARDS A SECURE ADDRESS SPACE SEPARATION FOR LOW POWER SENSOR NODES

Oliver Stecklina, Peter Langendörfer
IHP, Im Technologiepark 25, 15236 Frankfurt (Oder), Germany

Hannes Menzel
Distributed Systems/Operating Systems Group, Brandenburg University of Technology, Cottbus, Germany

Keywords: Wireless sensor node, Microcontroller, Virtualization, Memory protection, Security, Isolation, Operating system.

Abstract: Wireless sensor networks are becoming more and more considered for application in real world systems such as automation control, critical infrastructure protection and the like. By going wireless these systems are no longer to be protected by fences and walls but need to take into account security of all their components. In this paper we discuss two alternatives for implementing isolation on a Micro Controller Unit (MCU). The first one is a pure software solution, i.e. a Hypervisor which comes with a reasonable performance penalty when applied for 16-bit RISC processor cores such as the TI MSP430. Since it is a pure software solution it can be applied to existing MCUs without any hardware modification. Our second approach is to use a Memory Protection Unit (MPU) realized in hardware, which is placed between the processing core and the resources of the sensor node. The MPU especially supports fine-grained isolation of the sensor node software and further reduces the performance penalty compared to the pure software solution.

1 INTRODUCTION

Wireless Sensor Network (WSN)s are becoming rapidly adapted for real life application scenarios e.g. in the area of automation control or telemedicine. This type of application scenarios require a high level of dependability and safety. Both features cannot be guaranteed if the security of the WSN cannot be ensured. While in the last years there was significant research effort on network security for WSNs only very few approaches have been focusing on improving the security of individual nodes. Worm attacks like Stuxnet have impressivly demonstrated that automation control systems are a potential target of malicious attacks. In addition the work published in (Francillon and Castelluccia, 2008) has shown that sensor nodes can be compromised by harmful code such as worms.

In the single address space of a low power MCU, a simple code modification can completely compromise the system's security. We argue that the enforcement of high security levels is only possible by introducing a secure isolation of components. The code size of a single component has to be as small as possible to reduce the impact of a vulnerability. We present two separation concepts for low power MCUs, which allow the execution of software in isolated compartments. Beyond this we present a performance assessment that clearly indicates that both concepts are feasible from a performance point of view. Our approach inserts an additional layer between the software and the resources of a sensor node. As a pure software solution the isolation is enforced by a Hypervisor, which checks any memory access generated by the software during its operation. Similar memory checks are done by a MPU in our hardware-base approach. Furthermore the MPU supports a fine-grained separation of components to guarantee a high security level.

The rest of this paper is structured as follows. Section 2 introduces vertical and horizontal isolation techniques. These well known techniques stem form resource rich computer systems, but in section 3 we explain how these techniques can be used to improve the security of sensor nodes. In Section 4 we review some related work. The selected papers address a secure and dependable isolation of software entities

Stecklina O., Langendörfer P. and Menzel H. (2011).
TOWARDS A SECURE ADDRESS SPACE SEPARATION FOR LOW POWER SENSOR NODES.
In *Proceedings of the 1st International Conference on Pervasive and Embedded Computing and Communication Systems*, pages 512-517
DOI: 10.5220/0003364505120517

as well as state of the art virtualization concepts for WSNs. Finally the paper concludes with a short summary.

2 ISOLATION CONCEPTS

Isolation in the context of operating systems denotes the property that a certain software entity can access only those resources, which are assigned to itself. By the term software entity we mean a task, may be a scheduled computation (Cha et al., 2007), or an event-driven function (Levis et al., 2004), produced by a hardware interrupt. To guarantee security isolation has to ensure that the separation is still effective in case that a software entity is running malicious code. In this section we will introduce classic isolation concepts. They are mainly realized and used on resource rich systems, but known to be very effective.

2.1 Horizontal Isolation

To enforce a secure separation of processes distinguished capabilities are required. Commodity operating systems realize a horizontal separation by using a supervisor and an user mode. While in the supervisor mode all system resources are accessible, in the user mode access is limited to resources assigned to a certain process. Resources assigned to a process running in the user mode are managed by activities running in the supervisor mode. User mode activities cannot extend or modify their own access and execution rights for any of the system resources. This separation of capabilities is known as a type of horizontal isolation.

A trend in modern systems is running multiple operating systems on a single computer system by using a Hypervisor. A Hypervisor is a layer between the software and the resources of a modern computer. So, it decouples the operating system from the hardware. Hence, operating systems are running in a virtualized environment. Hypervisor techniques can be differentiated in full and para-virtualization. A full virtualization does not require any modification to a guest system (Bellard, 2005), but without specialized hardware features its implementation is difficult and often inefficient. I contrast to this para-virtualization uses few code modifications to prepare a guest system for virtualization. The XEN system uses this approach for running guest systems in a secure and efficient manner (Barham et al., 2003).

The virtualization concept supports a secure separation of software components and is already used in many application scenarios to setup high secure, dependable and robust systems.

2.2 Vertical Isolation

Vertical isolation enforces that a process is capable to operate unrestricted on its resources and in addition that foreign resources are invisible to it or protected form it. In a commodity operating systems this type of isolation is realized by protection keys, paging or segmentation.

The protection key scheme was introduced with the IBM360 system (IBM, 1964). It partitions the system memory in regions with a fixed size and assigns a single key to every region. If a process generates a memory access the key currently stored in the CPU status register is compared with the key assigned to the memory region. If the two keys are equal the access is granted otherwise a protection exception is triggered. It is a quite simple and effective scheme, but it does not support shared regions and does not check memory accesses generated by an instruction fetch. A more elaborated approach for address separation is the use of translation schemes as done when paging or segmentation are employed. These schemes translate a virtual address into a physical one by using translation maps. The concepts support shared memory and checks all memory accesses. But they require memory for storing the translation maps and a privileged mode for modifying these maps. Thus, these approaches cannot be used for on resource constraint sensor nodes.

The effectivity of a vertical isolation depends on the size of the isolated software entity. There are a number of approaches for supporting programmers to separate their applications. An approach to partition an application into an untrusted and a trusted section is proposed in (Kilpatrick, 2003). The execution of privileged operations and private data are isolated from the untrusted part of the application. For achieving more acceptance (Brumley and Song, 2004) proposes an automatic incorporation of such a vertical isolation into source code. We are convinced that these approaches are also suitable for improving the security of a sensor node.

3 ADDRESS SPACE SEPARATION ON SENSOR NODES

The address space of low power MCUs is shared among all software entities and does not support any isolation concepts. For safety aspects a secure isolation can be realizable by using a type-safe programming language or by employing type-checks at runtime. But in the context of security, where malicious code will not follow such a protection scheme, the enforcement has to be more strict and effective. In this section we will introduce two alternative horizontal isolation concepts i.e. a *MSP430 Virtualization Environment for the MSP430 MCU* and a *Memory Protection Unit (MPU)* - to enforce security on low power sensor nodes.

3.1 MSP430 Virtualization Environment

Virtualization is a pure software solution and decouples the software from the physical hardware. Thus, it enables a secure isolation of software entities. Due to the simple architecture of low power MCUs we have to emulate all guest operations in a full virtualization environment. This is done by the Hypervisor that has an exclusive access to the node's hardware. The virtualized software gets only a logical view onto the physical system resources.

The memory footprint and the performance of the Hypervisor is defined by the number and the complexity of the emulated instruction set. We use a 16-bit MSP430 from Texas Instruments (TI) as basic architecture. The MSP430 is widely used in various sensor nodes and has a simple RISC architecture. Its instruction set consists of only 27 basic operations and supports only two instruction formats. In contrast to this Atmel's ATmega has about 120 operations and the SPARC V8, as a well-known RISC architecture, supports 72 basic operations. Furthermore the simple von-Neumann architecture of the MSP430 simplifies relocating of a guest system. The MSP430 has a static memory, in which resources are mapped to fixed places into the 16-bit address space. During emulation the addressed resources can be efficiently identified by substracting the relocation offset.

We use a MSP430X CPU (CPUX) for running the Hypervisor. The CPUX is fully compatible to the 16-bit version, but has an extended address space of 20-bits. As shown in figure 1, we store the virtualized 16-bit guest systems in the upper memory. The lower 16-bit address space is used by the Hypervisor. The available memory is shared among the Hypervisor and the guest systems. A guest system gets a dedicated memory section, where its stack and heap as well as its virtualized CPU and peripheral registers are stored. Any access to the memory and to peripheral registers generated by the guest system is controlled by the Hypervisor. IO resources are also emulated by the Hypervisor. A guest access is passed via the peripheral registers inside the guest memory.

The instruction emulation of our virtualization environment has five execution stages: instructions fetch (IF), instruction decode (ID), memory access

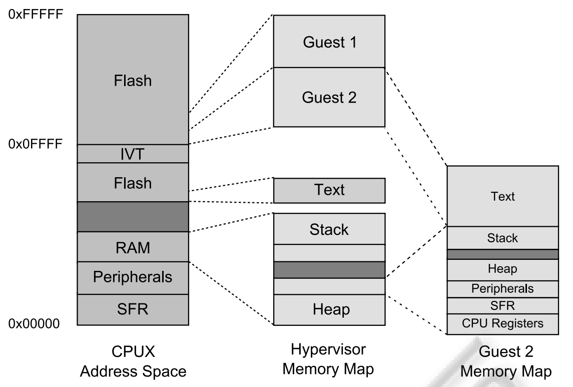

Figure 1: Address space of the MSP430 virtualization environment on a CPUX as host system. Guest systems are stored in the upper memory and are mapped into lower 16-bit address space by the Hypervisor at runtime.

(MEM), execution (EX) and write back (WB). The memory is accessed in stages IF, MEM and WB. Before executing a stage the used addresses are checked by the Hypervisor to ensure that they are located in the guest's memory region. In case of an access violation the guest state has to be rolled back to the operation's start. Therefore all data will be stored in shadow registers and writing into CPU registers in the guest's memory is delayed until the address used in the WB stage is validated by the Hypervisor. The ID and EX stages can be executed by the Hypervisor without any restriction. In case of an access violation a guest interrupt is triggered. We use a non-maskable interrupt, so that the guest system has to handle it or it will be reseted.

Due to the emulation of all operations the performance of the virtualized system is much slower than the one of the native system. At the current development state of our Hypervisor we can only provide a performance estimation. We are convinced that our emulation needs between two or six operations maximum per execution stage. The MSP430 needs up to six clock cycles for executing a complex operation. That means that in a worst case a emulation of a complete operation takes up to 180 clock cycles. But we are sure that in the average case the overhead will be much smaller, in particular when the number of simple instruction is a significant part of the emulated code. Furthermore the execution speed can be increased by using a higher clock rate for the host system. Nevertheless on a CPUX the energy consumption will be slightly higher.

Beyond optimization and higher clock rates the performance can be significantly improved by using para-virtualization. Similar to QEMU small pieces of guest code will be translated and stored in cache (Bellard, 2005). While emulating the same code again the cached block can be used. Due to the scarce memory resources of a MCU the cache size is quite limited. But this limitation can be neglected for sensor node applications with good locality or by using external memory as an additional code cache.

3.2 Hardware-based Fine-grained Isolation of Software Entities

A virtualization environment makes primarily sense for the execution of commodity operating systems. But this coarse-grained separation is unpractically for many sensor node applications and causes an inappropriate overhead. A fine-grained separation with an equal security level becomes feasible by extending the MCU with a hardware MPU. By taking the characteristics of sensor node applications into account the design of such a MPU can be simplified.

We propose an MPU that is tailor made for sensor nodes. It will support up to 16 EIDs and partitions the memory in up to 128 segments with a variable size with a minimal size of 2^1 and a maximal size of and 2^{13}. An Entity IDentifier (EID) is similar to a process identifier and is used to address the corresponding software entity. In contrast to a fixed segment size a variable one increases the complexity of the corresponding MPU, but allows for fine-tuned isolation of resources.

The maximal allowed number of EIDs and the maximal allowed number of memory segments is a trade-off between MPU complexity and system flexibility. The memory segments are described by segment descriptors, which are stored inside the MPU. A descriptor contains the base address, the size and the permission rights for its corresponding memory segment. The permission rights are described by the 4-bit EID and the 3-bit access type (read, write, execute). Furthermore the descriptor has a shared memory indication bit. If this bit is set the EID points into an array of group descriptors. A group descriptor defines a group of software entities sharing a piece of memory.

The MPU, see figure 2, is designed as an external component and is placed between the processing core and the node's resources. Any memory access generated by a software entity is passed by the Memory Address Bus (MAB) to the MPU. Whenever the memory is accessed the MPU determines the addressed memory segment and compares the access type and the EID with access rights stored for this segment. If they are equal the data is transfered via the Memory Data Bus (MDB) to the processing core. In case of a mismatch the access is denied and an access violation interrupt is signaled to the processing core.

Our approach does not require a privileged level

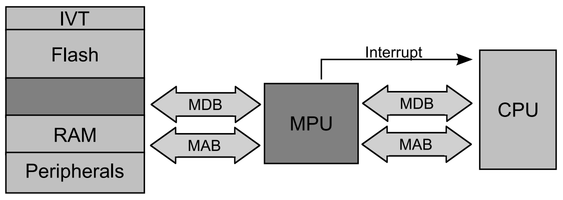

Figure 2: Integration of a hardware MPU as a component between the sensor node resources and the processing core. Any memory access is validated by monitoring the MAB. Access violations are signaled via an interrupt to the processing core.

like a supervisor mode. All software entities operate on the same privilege level. The current EID is stored in a MPU register and can be modified via a memory mapped interface. Writing a new EID into the register causes a context switch. The software has to ensure that in the next instruction a jump into a memory segment of the selected entity follows. Otherwise the following instruction causes an access violation. In addition to this we are supporting an unprivileged two-step segment initialization process. In the first step, at compile time, the resources are assigned to the entities and in a second step, at boot time, the segment descriptors are passed to the MPU via a memory mapped configuration interface. To protect the system against an unintended or malicious reconfiguration this interface should be write protected for all software entities after completing the boot process. A looser configuration is possible, but will reduce the system's security.

We realized the MPU in our co-simulation framework. The framework consists of the MSPsim (Eriksson et al., 2008) plus a SystemC environment. The MPU is implemented in SystemC and the MSPsim operates as a stimuli generator. Our co-simulation simplifies the developing process, but operates on a higher abstraction level. We can use it to get a behavioral description of the system to validate our approach. Furthermore we can estimate the required resources and get a rough performance estimation. The simulation results have shown that the most expensive component of the MPU is the segment lookup logic. Currently we analyze two different lookup strategies: a time consuming sequential search and a power consuming parallel search. We have already prooved that using a cache a segment lookup can be done in a single clock cycle, so that the processing core has not to be stalled.

Our MPU is designed to support small pieces of memory to make a fine-grained separation of software possible. But such a setup requires the integration of various context switches among the isolated compartments. We support this by a C-to-C source code translation process. We propose to implement a translation scheme similar to the approach presented in (Brumley and Song, 2004). In that scheme a programmer has to annotate security contexts by using C-macros and the switching code and the Inter Process Communication (IPC) is automatically generated by an afterburner. Furthermore we propose here a concept of an afterburner that can separate a well-defined software without any user annotation. The isolation boundaries would be aligned at the object file interface. A well-defined software object is offering only a minimized public interface and wraps the private data. As shown in figure 3, the afterburner isolates every object as a software entity and replaces a function call by a context switch and an IPC. This makes a secure and fine-grained separation of a complex software system feasible.

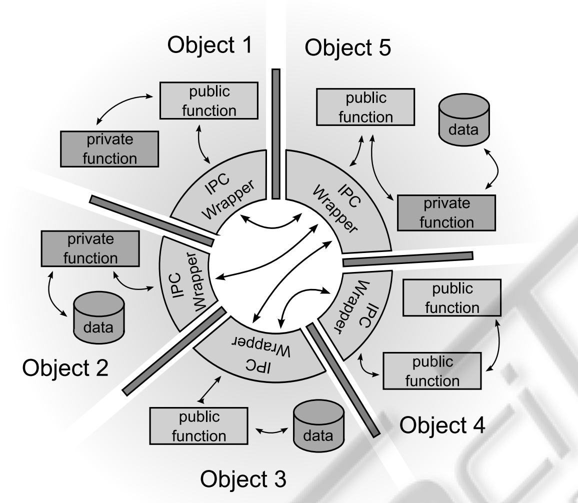

Figure 3: Fine-grained separation of software entities by encapsulating objects in different address spaces and replacing a cross-object function call by a context switch and an IPC.

4 RELATED WORK

In this section we will discuss related work. Most sensor node attacks exploit the simple node architecture with its single address space. Due to the unrestricted access to resources any successful attack compromises the whole system. The lack of restrictions can be used to construct malicious code by itself. The authors of (Francillon and Castelluccia, 2008) demonstrates how a return oriented programming attack can be used to annul the isolation of the data and the code section of the Harvard architecture. They use existing code and tricky reordering to execute an uncontrolled update process.

The authors of (Francillon et al., 2009) present a approach to block any kind of stack smashing attacks by a small performance penalty. But the solution requires a hardware extension and does not check any memory accesses. In (Lopriore, 2008) a pure hardware approach is presented that checks memory accesses generated by a software entity. The proposed solution partitions the node's memory in blocks with a fixed size and is mainly aimed at confining the consequences of programming errors. The authors of (Kumar et al., 2007) present a hardware extension for an AVR MCU that modifies the instruction set to check memory accesses during store operations and to integrate return stack protection. The solution is also focused on software dependability. Security aspects are not addressed.

Virtualization for achieving a higher level of robustness, dependability and safety is used by various approaches. In (Stilkerich et al., 2010) a JVM-based solution is presented, in which the node's software is written in Java. KESO, an ahead-of-time compiler, is used to generate the C code (Thomm et al., 2010). It provides a software-based memory protection by separating software entities in isolated JVMs. A virtual machine for TinyOS is presented in (Levis and Culler, 2002). It provides the execution of portable pieces of code. The VM has a very small instruction set that ensures a small memory footprint. But the solution is focused on portability, security aspects are not addressed. A similar approach is presented in (Müller et al., 2007). It uses an extensible byte-code language that is Turing complete. Security aspects are also out of the scope of that paper.

5 CONCLUSIONS

In this paper we have investigated the idea of using horizontal isolation between software entities and resources of resource restricted devices such as embedded systems and sensor nodes. We have presented two concepts for realizing such a separation. The first concept we investigated is the realization of a Hypervisor for a MCU, which is implemented as an intermediate layer between the processing core and the resources of the device. We have done a high level performance assessment which clearly indicates the feasibility of this approach. i.e. the performance penalty is about factor four in the worst case i.e. for the most complex instructions. This approach is especially interesting since it requires no modifications on the existing hardware and is fully transparent to the programmer. The second concept requires an extension of a MCU, by a Memory Protection Unit (MPU). This MPU is intercepting all memory accesses and checks whether or not the currently executed task may read/write from/to the specified address. We have partly implemented our MPU in a co-simulation environment. There we could prove that the memory access verification can be done within a single clock cycle, i.e. that this protection scheme can be applied without performance penalty.

Beside the isolation of software entities their virtualization can improve the robustness of the whole system against denial of service attacks. The Hypervisor is fairly scheduling the virtualized systems so that the affected instance can capture only a piece of the available computing time. The attack is limited on the affected virtualized system and in case of an distributed and well-defined system design critical services can be still available.

Our approach is still work in progress so that we are not able to completely validate our concept and providing sophisticated measurement results. But our performance assessments clearly indicate that horizontal isolation is a valid concept to improve the security and by that reliability of MCUs.

ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Community's Seventh Framework Program (FP7/2007-2013) under grant agreement No. 258754.

REFERENCES

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA. ACM.

Bellard, F. (2005). Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA. USENIX Association.

Brumley, D. and Song, D. X. (2004). Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, pages 57–72.

Cha, H., Choi, S., Jung, I., Kim, H., Shin, H., Yoo, J., and Yoon, C. (2007). RETOS: resilient, expandable, and threaded operating system for wireless sensor networks. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 148–157, New York, NY, USA. ACM.

Eriksson, J., Dunkels, A., Finne, N., Österlind, F., Voigt, T., and Tsiftes, N. (2008). Demo abstract: MSPsim - an extensible simulator for MSP430-equipped sensor boards. In *Proceedings of the 5th European Conference on Wireless Sensor Networks (EWSN 2008)*, Bologna, Italy.

Francillon, A. and Castelluccia, C. (2008). Code injection attacks on harvard-architecture devices. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 15–26, New York, NY, USA. ACM.

Francillon, A., Perito, D., and Castelluccia, C. (2009). Defending embedded systems against control flow attacks. In *SecuCode '09: Proceedings of the first ACM workshop on Secure execution of untrusted code*, pages 19–26, New York, NY, USA. ACM.

IBM (1964). *IBM system/360 principles of operation*. IBM Press.

Kilpatrick, D. (2003). Privman: A Library for Partitioning Applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 273–284.

Kumar, R., Singhania, A., Castner, A., Kohler, E., and Srivastava, M. (2007). A system for coarse grained memory protection in tiny embedded processors. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 218–223, New York, NY, USA. ACM.

Levis, P. and Culler, D. (2002). Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95, New York, NY, USA. ACM.

Levis, P., Madden, S., Polastre, J., Szewczyk, R., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., and Culler, D. (2004). TinyOS: An operating system for sensor networks. In *in Ambient Intelligence*. Springer Verlag.

Lopriore, L. (2008). Hardware/Compiler Memory Protection in Sensor Nodes. *International Journal of Communications, Network and System Sciences*, 1(3):235–240.

Müller, R., Alonso, G., and Kossmann, D. (2007). A virtual machine for sensor networks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 145–158, New York, NY, USA. ACM.

Stilkerich, M., Lohmann, D., and Schröder-Preikschat, W. (2010). Memory protection at option. In *CARS '10: Proceedings of the 1st Workshop on Critical Automotive applications*, pages 17–20, New York, NY, USA. ACM.

Thomm, I., Stilkerich, M., Wawersich, C., and Schröder-Preikschat, W. (2010). Keso: an open-source multi-jvm for deeply embedded systems. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 109–119, New York, NY, USA. ACM.