Building Hardware Components for Memory Protection of Applications on a Tiny Processor

Hyunyoung Oh, Yongje Lee, Junmo Park, Myonghoon Yang and Yunheung Paek
Seoul National University
{hyoh,yjlee,jmpark,mhyang}@sor.snu.ac.kr,ypaek@snu.ac.kr

ABSTRACT

As we move towards the IoT era, it is clear that many tiny processors will surround us, constantly communicating and transferring valuable information with others. Although it is obvious that such processors will become appealing targets to attackers, they often lack sophisticated hardware memory protection mechanism, such as virtual memory. To enhance memory protection of tiny processors, several studies have tried to design new hardware protection mechanisms at low cost. Those mechanisms, however, demand invasive modifications to the CPU internal architecture, and these modifications require significant cost and time for CPU redesign and verification. In this paper, we present the design of hardware components for realizing an isolation of critical information on a tiny processor. Unlike other previous work, in our approach, several hardware components are integrated together to build a system with enhanced memory protection. To check the feasibility of our idea, we implement an early prototype where a RISC-V processor is connected with the proposed hardware components. Empirical results show that ours achieves the goal with virtually no performance and low area overhead.

1 INTRODUCTION

As we move to the *Internet of Things* (IoT) era, more and more extremely small embedded devices like implanted medical devices are expected to use tiny embedded processors. Moreover, these devices mostly would likely be connected to the IoT network and carry sensitive user information like user profiles or credentials, and accordingly they are becoming attractive targets for cybercriminals. Therefore, for the proliferation of small devices in the IoT era, it seems clear that we must properly address the security challenge that is to protect security-critical information on these devices from various attacks. To enforce the protection of the critical information, a conventional method is to implement a memory protection mechanism that can isolate trusted applications processing critical information from other untrusted ones.

To efficiently support memory protection, the hardware typically provides a special module for memory management, called the *memory management unit* (MMU). A main task of MMU is performing virtual-to-physical mappings to realize virtual memory so that the OS can exercise access control over every page in its memory system by specifying security properties for memory protection (i.e., memory access permissions and memory region attributes) in memory-mapped tables known as *translation tables* [2]. From the perspective of security, a key benefit of virtualizing address space is that it empowers the OS to protect sensitive data from unauthorized code by selectively binding the data with the code in the virtual address space. That is, in the virtual memory system, if any application wants to access data at runtime, its code and the data must be mapped (or bound) to the same virtual address space. Therefore, in order to authorize only a trusted application to access sensitive data, the OS simply configures translation tables in a way to map the physical address of the data onto the virtual address space of the authorized application code.

Despite its powerful security capability through code-data binding, virtual memory is rarely adopted by tiny embedded processors today in the market [3], but instead, deployed mostly in more capable processors targeting desktop PCs or other computing devices with ample resources. One major reason is that maintaining virtual memory inherently requires a considerable amount of computation which is more than what a resource-stringent tiny processor could afford. To satisfy the resource constraints, many vendors of such low-end processors have implemented in hardware memory protection mechanisms without virtualization. For this, these processors usually have a hardware memory management module, called the *memory protection unit* (MPU), that assists the OS to enforce basic access rules for memory protection on the device without virtual memory support. MPU assumes that the system has a single physical memory space regardless of the number of processes running on the device. The memory space is divided into a number of *regions*, and each process is assigned a set of regions associated with specific access permissions given by the OS. At runtime, MPU enforces the access rules by constantly checking if every process makes data accesses complying with their permissions.

Being free from the virtualization overhead, MPU can control data accesses of a process on memory regions with more efficiency. To maximize the efficiency, the access control mechanism of MPU in existing tiny processors has been simplified in a way that each region in the main memory is given the identical access permission for every process trying to access it. However, from the perspective of security, such efficiency exposes the system to a potential security breach that even when sensitive data must be processed by a trusted application, the region containing the data can be accessed not only by the trusted code but also by others (possibly from adversaries). For this reason, when users want to set different access permissions individually for each process, the OS should set different MPU configurations whenever the context switch occurs. However, this OS-dependent approach is reportedly no longer secure because

CARRV'17, Oct 2017, Boston, MA USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

many recent cases have reported that even an OS kernel can be compromised by the high-level attack such as rootkits. In those cases, the memory isolation based on MPU is no longer guaranteed.

To overcome the limitation of the OS-dependent memory protection, recent studies [4, 7] endeavor to extend the protection features by providing a hardware-enforced isolation of software modules targeting tiny embedded processors. In SMART [4], they presented a new processor architecture which includes an *attestation Read-Only Memory (ROM)* and a special storage in which the sensitive data are stored. Also, its MCU access control logic is changed to ensure that only the specified code in the attestation ROM can access the sensitive data in the special storage. In [7], they tried to enhance the protection mechanism of the original MPU architecture by augmenting hardware logic to link code regions to data regions, thus making the permission check execution-aware. Although they have successfully enhanced the security of tiny processors, all these previous solutions cannot be applied directly to the devices with existing processors because they require intrusive modifications to the internal architecture of an existing processor including the pipeline registers and datapaths.

A line of research [6, 8] has focused on implemental issues such as how security hardware logics can be implemented with keeping the underlying CPU intact. Both studies commonly have used the hardware debugging interface of the processor to extract the branch target addresses for the *control flow integrity* enforcement. By verifying branch target addresses at runtime, they have successfully detected the illegal control flow with incurring virtually no overhead on the host CPU. Motivated by their work, in this paper, we present the design of hardware components which enable the enhanced memory protection features for a tiny embedded processor. Our solution is basically exerting the strategy similar to that of earlier work [7]. However, while their approach requires a permanent design change of the host processor, our study conforms to the *modular design approach*, that several hardware components can be assembled together to offer the enhanced memory protection. As our first hardware component, we design the *memory region protector* (MRP) whose main role is to enable a tiny processor to have the code-data binding ability that has been provided by MMU with virtual memory. MRP is able to isolate code regions individually from each other according to their permissions, and thereby to bind data regions to a specific code region based on the pre-defined access rules between code and data regions. In order to define individual access rules between code and data regions, MRP maintains the register files for storing the low- and high addresses of code and data regions which are configured at the boot time. Also, We build the configurable *access permission matrix* which defines the access permission of a code region for each data region. With the access permission matrix, MRP determines the legitimacy of each data transfer by checking the defined access permission between the instruction address generated at the CPU's instruction fetch stage and the data transfer address generated at the execution stage. However, as our MRP is located outside the processor, those necessary addresses issued by the internal pipeline are not directly visible to our MRP. Thus, we must somehow devise a special mechanism to extract the processor internal information outwardly in a timely manner. For this purpose, we also build a generic *security interface* inside the processor to extract the control flow and the data access patterns of the host. To check the feasibility of our idea, we make a prototype where the slightly modified RISC-V Rocket is integrated with other hardware components.

The rest of the paper is organized as follows. Section 2 describes our assumption and the threat model. After Section 3 shows the overall system architecture for the memory protection, Section 4 explains the detailed implementation of our hardware modules. Then, Section 5 discusses the experimental setup and results. In Section 6, we cover the previous studies related to ours.

2 ASSUMPTION AND THREAT MODEL

In this work, we target embedded systems employing tiny processors available in today's market, which cannot afford MMUs for sophisticated virtual memory managements. We assume that the target system comes in the form of an SoC and therefore any external hardware can be attached during implementation.

As for our adversaries, we assume that untrusted software modules can be installed in the target devices. These untrusted modules can lead to compromising an entire program when (1) the modules have vulnerabilities which can be exploited by an attacker or (2) the modules themselves are malicious. This is due to the fact that, without address protection between modules, a software module would be able to corrupt the data used by another module. Additionally, we assume that the adversary has full control over the communication interface with the target system and can deliberately manipulate or eavesdrop on the messages sent over the interface. We also assume that the target system is exposed to physical attacks such as on-chip bus probing or reverse engineering but these are out of the scope of this paper.

3 OVERALL SYSTEM ARCHITECTURE

3.1 SoC Prototype Overview

As clearly stated in Section 1, the ultimate goal of our research is to develop the hardware components which can be used to build a SoC with an external memory protection capability. To this end, we first present the SoC prototype including our proposed hardware components. Figure 1 shows the overall architecture of our SoC platform. In the prototype, the host CPU is the RISC-V Rocket CPU in which the security interface is installed. The host CPU and our MRP module are connected via the *system bus*, which is the standard AMBA3 AXI interconnect [10]. MRP supports the security function that is to establish the software isolation environment by observing the host internal information (i.e., currently executing instruction address or memory access patterns of programs running on the host). In order to obtain the information, MRP has an additional connection to the security interface embedded in the host. Also, MRP has a connection to the access permission matrix that contains access permissions of a code region for each data region. By referring the access permission matrix, MRP determines if there is an illegal memory access. The main memory is a storage which keeps the code and data necessary for the host's operations. The host CPU and the main memory are connected through the *system bus* in a way that the main memory can be accessed by the host. Our MRP hardware is also attached to the system bus so that the host can configure our MRP at will. In Section 4, more details of the hardware modules will be explained.

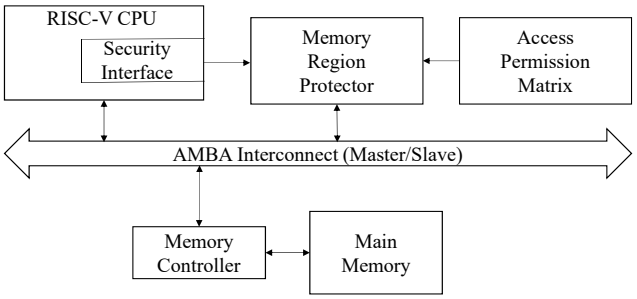

Figure 1: The overall architecture of our prototype

3.2 Memory Protection Process

Figure 2 illustrates the exemplary case for our SoC prototype with the high-level software and hardware architecture. Inside the SoC boundary, the RISC-V CPU is integrated with a *programmable read-only memory* (PROM). PROM stores the *boot loader* code to define the code and data region and initialize the access permission matrix and for each task during boot. Since the tiny devices that our security solution for the memory protection targets include limited tasks, PROM stores the code in advance. Once the code and data region are defined and the access permission matrix is initialized, they are controlled so that they can not be modified. The host CPU is also connected with the main memory and peripheral IPs such as a timer or high-speed communication interfaces. Our MRP monitors all memory accesses issued by the CPU based on the information fed by the security interface. These memory accesses include general memory access as well as access to memory-mapped IO devices (here, they are peripherals in Figure 2).

In the example given in Figure 2, each task has its own address space to access. During the software execution, our MRP checks whether or not any task running on the host CPU tries to access the wrong address space. During the boot-up sequence, the host CPU first access the *boot loader* code stored in PROM. Then, the boot loader programs our MRP to define the accessible code and data regions for each task.

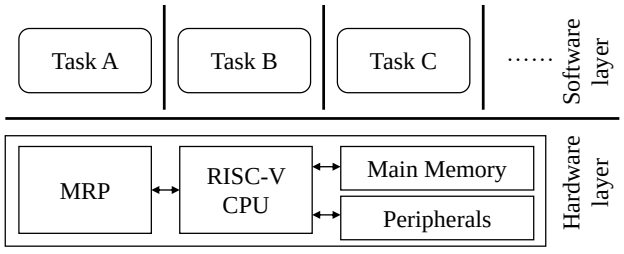

Figure 2: Memory protection process using our prototype

4 IMPLEMENTATION DETAILS

4.1 Memory Region Protector

Figure 3 shows the overall architecture of our MRP hardware. The MRP hardware is divided into four submodule. The *code region selector* (CRS) and *data region selector* (DRS) respectively finds the code region which the data transfer instruction belongs to and the data region which the instruction's data address lands. Another submodule is the *decision unit* that makes the final decision about the occurrence of illegal memory accesses when a data transfer occurs. To make the decision if there are illegal memory accesses, MRP refers to the *access permission matrix*. And the other is the *MRP controller*. During the boot-up sequence, through the *AHB slave interface* in the MRP controller the code stored in PROM configures the registers in CRS and DRS to define protected code and data regions, respectively and the registers in the access permission matrix to assign the allowed permissions. To define the address range of each code or data region, two registers are provided to set the low and high addresses of the region.

MRP needs the program execution information inside the processor to perform the aforementioned functions. The essential information is categorized into three groups: (1) the instruction address of currently executing code (PC) and (2) the data address and (3) the data transfer type of each data transfer instruction. MRP takes as input the information through the security interface. The information for groups (1), (2) and (3) is fed respectively by the security interface via the following pins:

(1) `data_transfer_inst_addr`
(2) `data_transfer_data_addr`
(3) `data_transfer_type`

In addition, the flag indicating the execution of data transfers is sent over signal line, `data_transfer_en`.

The `data_transfer_data_addr` and `data_transfer_inst_addr` are the data and instruction addresses of the data transfer instruction, respectively. The instruction address of transmitted data means the address of the currently executing code. The `data_transfer_type` indicates whether the data transfer instruction is stored or loaded. (i.e., either memory write or read). The `data_transfer_en` indicates a signal to enable the preceding signals. When a data transfer instruction is executed by the processor, the `data_transfer_en` signal is set on. At the same cycle, `data_transfer_inst_addr`, `data_transfer_type` and `data_transfer_data_addr` become valid. Then, the instruction and data address are respectively sent to CRS and DRS by the MRP controller. To find out which code region the executed data transfer instruction belongs to, CRS searches through its registers (`CodeRegion0-7`). Then, the code region number is output via `code_region_num`. Similarly, using the data address from the MRP controller, DRS looks up the matched code region in its registers (`DataRegion0-7`) and sends the region number outwards through `data_region_num`. When the incoming data address belongs to one of defined code region, the region number is conveyed via the `code_region_num_t` signal. Then, the selected region numbers are sent to the access permission matrix. Finally, the access permission matrix searches the allowed permission corresponding to the region, and sends back the permission to the decision unit to decide whether or not the executed data transfer is legitimate. When the currently executed data transfer instruction is found to violate the received access rule, the decision unit sets the `illegal_access` signal on.

Up to now, we have explained how a code region for a trusted software module can be defined by the set of memory regions and

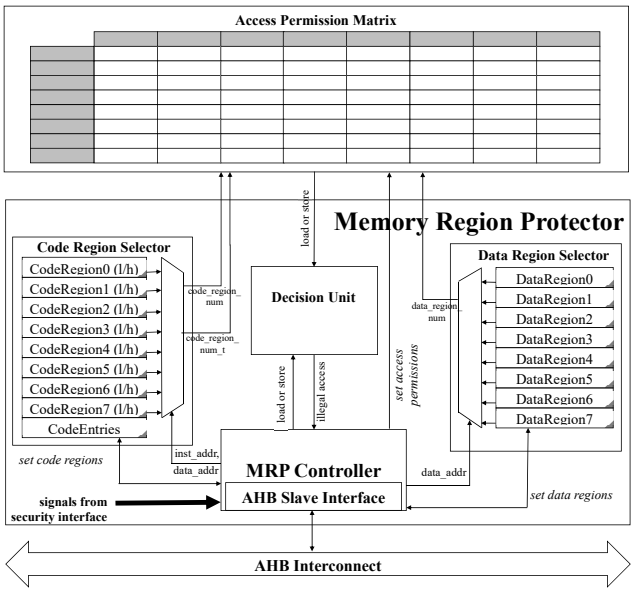

Figure 3: Memory Region Protector architecture

associated access permissions. In addition to this, we have to consider the case when the trusted software module is invoked by another software regardless of whether the latter is trusted one or not. The most important thing for a safe communication to the trusted software is to enforce that the trusted software is only invoked at a known address. For example, under *code reuse attacks* [9], the attackers might want to reuse the code snippets at an arbitrary address in the trusted software to access security-critical data. To provide the users with an interface to safely call trusted code, MRP has an optional registers called `CodeEntries` which contain the instruction addresses of the overall code regions set by MRP that can be executed by other software modules. When an untrusted software tries to invoke one of the trusted software modules, MRP checks if the value in PC after change of the control flow(i.e., target address of branch instruction) can be found in `CodeEntries`.

4.2 Access Permission Matrix

The access permission matrix has the access permission for code and data regions, and the number of each region is eight. The access permission matrix is depicted in Figure 4. The access permission matrix maintains two types of relationship: code vs. code region and code vs. data regions. There are three types of access permissions: Readable, Writable, Executable. For each code and data region, The data in the access permission matrix is represented by three bits to cover the every case of permission allocation.

During the boot-up, the code stored in PROM initializes the access permission matrix. When the code stored in PROM is executed, the processor launches a command. Over the AHB interconnect, the command is encoded in accordance with the bus protocol. On receiving the command, the *AHB slave interface* in the *MRP controller* module decodes the command and sends the decoded information to the access permission matrix to update its registers.

OBJECT / SUBJECT	Code Region0	Code Region1	Code Region2	Data Region0	Data Region1	Data Region2
Code Region0	RX	-	R	RW	-	RW
Code Region1	-	RX	-	-	R	-
Code Region2	-	R	RX	RW	R	RW

Access Permissions

R : Readable, W : Writable, X : eXecutable
- : No access is permitted

Figure 4: Access Permission Matrix

The access permission matrix reads and sends the permission after the receiving the code/data region number from MRP. The procedure is as follows. First, let $AP(Ci, Dj)$ be the access permission between code region i and data region j. Similarly, $AP(Ci, Cj)$ means the access rule between code region i and j. On receiving the code region number and data region number, the access permission matrix reads the permission $AP(Ci, Dj)$. When the code region i and j are valid, the access permission matrix reads the permission $AP(Ci, Cj)$. Then the access permission matrix sends the allowed permission to the decision unit.

4.3 Security Interface

The security interface is interacting with the RISC-V processor so that the interface can transfer the processor's internal information to MRP such as instruction address or data access patterns. To extract these pieces of information, we analyzed the pipeline architecture and internal components such as *stage control registers* or the *program counter* at each stage. The RISC-V processor has a six-stage pipeline composed of pcgen, fetch, decode, execute, memory and write back. Among these pipeline stages, the main part we are paying attention to is between the execute and memory stages where instruction/data addresses are calculated and stored for extracting the signals we need. However, there may often be a case that the sequence conducted by the core is executed differently from the input instruction sequence due to the execution optimization or interrupt. In this sense, to recognize the context of the instruction being executed, we designed the interface to observe control registers with the signals regarding the execution status. Consequently, our security interface have to determine which information is accessed to the memory by keeping track of the signals generated by the core.

For this purpose, the security interface is composed of three parts: enable flag generation, classification and instruction/data address extraction. First, an enable flag generation is a part which confirms that the data transfer actually takes place and triggers the data transfer to MRP. If it turns out that data transfer to the data array inside the data cache occurred, the security interface is ready to export instruction/data address to MRP. Second, since MRP creates each area table depending on the load or store operation, there should be a classification part that can send the signals to MRP after distinguishing what an operation is taking place in the core. Lastly, based on memory access type perceived by the classification part, the instruction/data address extraction part obtains the corresponding instruction/data address transferred from the

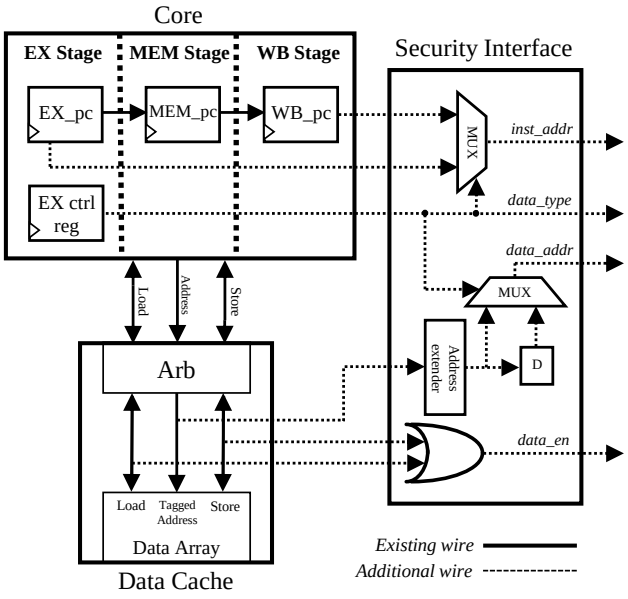

Figure 5: Information extraction through Security Interface

core and sends them to an output port. These parts are shown in Figure 5.

First of all, to recognize if the memory access operation occurs, the security interface checks the ready/valid signal between the data array and the arbiter that controls the input signals of the data array. At this time, since each ready/valid signal exist in the store/load operation, the state when the ready/valid signal corresponding to each operation are both high can be used as an enable flag. From the enable flag, the security interface gets ready to transfer the signals to MRP. Additionally, when the ready/valid signal become both high for the load or store operation, the accessed address is also passed from the arbiter to the data array. However, this address is a short address based on the data address sent from the core. Therefore, the simplified address sends to the output port after being re-extended to the data address received from the core through the *address extender*.

A classification part is used to classify whether the currently executed instruction is the load or store operation. Since MRP creates different tables depending on the load or store operation, it is necessary to recognize that this instruction has what type of memory access. To determine the memory access type, the security interface refers to control registers that contain information about the instruction being executed and controls the operation of the execute stage. In summary so far, if the enable flag occurs by the ready/valid signal inside the data cache and the interface obtains the corresponding data address, the operation which makes the enable flag high is determined which memory access type has.

Lastly, the most important factor is, when the enable flag occurs, which instruction address should be fetched from the program counter at which stage. To accomplish this work, the instruction/data address extraction part are used to export appropriate addresses depending on the input instruction sequence. In general, it is executed in the core in the same order as the instruction entered. However, we observe some cases which the processing order may be different from the order of the input instructions depending on the execution status. In this case, we allow the security interface to align elements within a data set by using control registers from an execute stage. In other words, the security interface comprehensively determines the current situation and selects the corresponding instruction/data address executing the load or store operation through the multiplexers. Finally, the selected address will be passed to the output port of the security interface.

5 EXPERIMENTAL RESULTS

5.1 Hardware Area Overhead

To evaluate the area overhead of our approach, we have implemented the SoC prototype including the hardware components as described in Section 3. In our prototype, we use the Xilinx Zynq-7000 board and use a version 1.7 of RISC-V Rocket core parameterized by FPGA configuration DefaultFPGASmallConfig, as the host processor. In our implemented MRP, the number of code and data region are both configured to be eight. The bus compliant with AMBA AHB2 protocol [1] is used to interconnect all the modules in our prototype system. As the OS kernel, Linux zynq 3.15.0 is used.

Based on the parameters for the prototype as described above, we synthesized our overall SoC Design onto the prototyping board with a Xilinx XC7Z020CLG484-1 FPGA and 64MB external SDRAM. Table 1 provides the design statistics of our hardware prototype. We quantified the resources necessary for our hardware components in terms of lookup tables for logic (LUTs) and FFs. The design statistics show that, compared to the baseline Rocket core, our components incur the resource overhead of 12.81% and 21.48% for FFs and LUTs, respectively. We also estimated the gate count of our hardware components using Synopsys Design Compiler. With a commercial 45nm process library, the total gate-count of the proposed modules is 16,586(1,712 gates for the security interface, 12,828 gates for MRP and 2,046 gates for the access permission matrix).

Category	Components	LUTs	FFs
Baseline System	Rocket Core	9229	6894
Our Hardware Components	Security Interface	80	195
	Memory Region Predictor	1066	1082
	Access Permission Matrix	36	204
	Total	**1182**	**1481**
	% over Baseline System	**12.81%**	**21.48%**

Table 1: Synthesis result of our hardware components

5.2 Performance Overhead

The security interface incurs zero performance overhead because it extracts the internal information without changing the critical path of the host CPU. Using the interface, our MRP runs in parallel with the functional execution of the host. Hence, the access permission check of MRP also does not impact the performance of the target system. To initialize the protected code/data regions and the access permissions in MRP, the overhead of setting registers is additionally

imposed. For each code or data region, three register writes are required: two for the high- and low-address of the region and one for the access permission. However, considering that tiny processors mainly target small devices where most of their applications are already fixed and thus can be statically allocated, this register-setting overhead has little impact on the whole system performance.

5.3 Security Considerations

Firstly, the main goal of our MRP is to establish the isolation environment for the critical code and data. For this purpose, our MRP refers to the access control matrix which ensures that a data region storing security-critical data is only accessed by an authorized application. In addition, the MRP prevents untrusted software from indiscriminately accessing (or invoking) the tasks in code regions by restricting its access to predefined code entries in MRP registers.

As another application, our MRP would be helpful in establishing the root of trust for remote attestation. Remote attestation is the process of securely verifying the state of a remote hardware platform to ensure that the known trusted software is correctly running on the platform [4]. To ensure this code integrity, the usual strategy of attestation is to compute a hash value of the trust code periodically by reading the code itself and comparing the hash value to a golden value which is pre-computed when the trusted software is initially installed. To support this scenario, our MRP is also capable of providing a code region with an exclusive access permission to read another code region to be attested and of providing the secret key necessary for computing the hash value.

6 RELATED WORK

As memory protection (e.g., process isolation) is considered to be a fundamental primitive of most security enhanced systems, much research effort has been devoted to developing, in hardware, efficient isolation mechanism available to various platforms. For high-end devices (e.g., desktop PCs or even smartphones), the problems of dealing with software isolation are relatively well-understood and a considerable amount of knowledge has been obtained from researches conducted for the past decades. In fact, most existing high-end processors deploy an MMU hardware to protect the system by preventing one process from interfering with the critical code or data of another process. On the other hand, only a few research studies have been interested in developing enhanced memory protection features for low-end resource-impoverished devices.

In SMART [4], one of the hardware solutions to enhance memory protection for low-end devices, they proposed a technique to establish a root of trust for performing remote attestation. In their work, a special storage in the memory space is added to the processor architecture to store the secret key which is necessary to perform hash functions for attestation. They also implemented a custom access logic which ensures that only the code residing in a specified region in ROM can access the key in the special storage. This way, the secret key is only accessible by the verifier code in ROM, and users can get the assurance that the hash value is computed by the authorized verifier code using the correct secret key. TrustLite [7] is another noticeable hardware-assisted technique aiming to enhance the memory protection of tiny processors with MPU. Basically, TrustLite extended the memory access control model proposed by SMART. Instead of implementing a special storage for the secret key and the access control logic, TrustLite realized a programmable access control logic, called the *execution-aware memory protection unit (EA-MPU)*, to link code regions to data regions. The EA-MPU hardware has an *access rule table* which maintains the access permissions from code regions to data regions. To find out which code is currently running on the host, they modified the pipeline logic of a processor to extract the *instruction pointer* (IP) which points to the address of the executed instructions. Also the read/write addresses of data transfer instructions are also extracted to monitor the data access pattern. The EA-MPU logic checks the IP and data transfer addresses against the access rule table to decide the existence of invalid memory accesses. These hardware studies for the memory protection problem empirically suggest that capitalizing on hardware techniques should be an excellent way to overcome the lack of sophisticated memory protection features of low-end devices (or processors). Unlike our approach, however, their techniques have made permanent changes to the internal architecture of the existing processor to improve memory protection. In our work, we present a new design approach to building a security system with enhanced memory protection by integrating several hardware components.

As a hardware component that plays a role similar to the MRP designed in this paper, there are MPU [3] of ARM Cortex series processors and PMP [5] which can be optionally enabled in RISC-V processors recently. For MPU and PMP, the size of a protected region can be configured as a power-of-two multiple of 4KB. Compared to these components, MRP has the advantage that the upper- and lower-bounds of a region can be configured to any address. Consequently, MRP provides more flexibility in setting up the protected regions. Moreover, it is meaningful that the security interface provides the system developers with the capability of installing various security hardware components which operate based on the CPU internal information.

7 CONCLUSION

We present in this paper a hardware-based memory protection solution to enhance the memory protection features of a tiny embedded processor. To achieve this goal, we propose several hardware components. The core component of the our hardware is MRP that plays the role of deciding the existence of invalid memory accesses by observing the source and destination addresses of data transfers inside the host. To this end, MRP refers to the access permission matrix that provides the access permission for each code and data region. Moreover, to establish a communication channel between the our hardware components and the host processor, we design the generic security interface by modifying the internal architecture of the RISC-V processor. Being connected to the RISC-V CPU via the security interface, MRP can receive the CPU internal information at runtime. The experimental results showed that our hardware based solution successfully isolated code regions individually from each other and provides enhanced memory protection with low area and virtually no performance overhead.

REFERENCES

[1] ARM 1999. *AMBA Specification.* ARM.
[2] ARM 2007. *ARM Architecture Reference Manual (ARMv7-A and ARMv7-R edition.* ARM.

[3] ARM 2014. *ARM Cortex-M7 Processor.* ARM.
[4] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. 2012. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust.. In *NDSS*, Vol. 12. 1–15.
[5] Electrical Engineering and Computer Sciences University of California 2016. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture.* Electrical Engineering and Computer Sciences University of California.
[6] Zonglin Guo, Ram Bhakta, and Ian G Harris. 2014. Control-flow checking for intrusion detection via a real-time debug interface. In *Smart Computing Workshops (SMARTCOMP Workshops), 2014 International Conference on.* IEEE, 87–92.
[7] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. 2014. TrustLite: a security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems.* ACM, 10.
[8] Yongje Lee, Jinyong Lee, Ingoo Heo, Dongil Hwang, and Yunheung Paek. 2017. Using CoreSight PTM to Integrate CRA Monitoring IPs in an ARM-Based SoC. *ACM Trans. Des. Autom. Electron. Syst.* 22, 3, Article 52 (April 2017), 25 pages. https://doi.org/10.1145/3035965
[9] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security.* ACM, 552–561.
[10] Xilinx 2011. *AXI Reference Guide.* Xilinx.