SCM: Secure Code Memory Architecture

Ruan de Clercq†, Ronald De Keulenaer⋆, Pieter Maene†,
Bart Preneel†, Bjorn De Sutter⋆, Ingrid Verbauwhede†,
†KU Leuven, Belgium - ESAT/COSIC and iMinds
⋆Ghent University, Belgium - Computer Systems Lab

ABSTRACT

An increasing number of applications implemented on a SoC (System-on-chip) require security features. This work addresses the issue of protecting the integrity of code and read-only data that is stored in memory. To this end, we propose a new architecture called SCM, which works as a standalone IP core in a SoC. To the best of our knowledge, there exists no architectural elements similar to SCM that offer the same strict security guarantees while, at the same time, not requiring any modifications to other IP cores in its SoC design. In addition, SCM has the flexibility to select the parts of the software to be protected, which eases the integration of our solution with existing software. The evaluation of SCM was done on the Zynq platform which features an ARM processor and an FPGA. The design was evaluated by executing a number of different benchmarks from memory protected by SCM, and we found that it introduces minimal overhead to the system.

Keywords

Software Integrity; SoC; Security; Hardware

1. INTRODUCTION

Systems-on-Chip (SoC) vendors typically integrate a number of third-party Intellectual Property (IP) cores and a number of custom IP cores into their SoCs. Usually a SoC's IP cores communicate via standard interfaces. This is critical, as it allows for much shorter design cycles, faster time-to-market, reduced vendor lock-in, reduced non-recurrent engineering costs, flexibility towards the creation of product ranges by interchanging individual IP cores, etc.

For long-term software and data storage, modern SoCs often use external memories such as Dynamic Random-Access Memory (DRAM), FLASH, or Read-Only Memory (ROM).

Meeting the security requirement of code integrity is challenging in scenarios where attackers can control the external SoC interfaces and components, or can tamper with software before it is installed on a device. Attackers can try to alter software when it is transferred from vendors to users, or when it has already reached the users, or when it is transferred and stored to, from, and in the external memories. It is in particular challenging to provide strong security guarantees without abandoning the aforementioned benefits of using interchangeable IP cores via standard interfaces.

Our proposed solution, called SCM, is an IP core that verifies the integrity of code as it is fetched from external memory into the caches of the SoC's processor. This IP core can be added to existing SoC designs without requiring any changes to the used IP cores, memories, or interfaces; and without requiring any changes to existing SoC design flows. In other words, none of the already used components needs to feature any security support to provide code integrity. Moreover, our design requires only minor adaptations to the software build process and offers flexibility in supporting a range of reaction mechanisms. This includes the guarantee that not a single tampered instruction can be executed. Furthermore, our solution fits into many schemes to sign and distribute software. Finally, as we will demonstrate, the performance overhead of SCM is limited.

Many existing works [8, 17, 19, 20, 5, 9, 21, 10] provide code integrity guarantees. However, it appears that all these works require modification to the processor, memory hierarchy, or existing IP cores. Modifying existing IP is difficult, as it requires modification by the IP vendor (which could be expensive), or requires the IP vendor to release it's HDL to the client (which could be even more expensive). SCM provides integrity guarantees without making any modification to the existing IP on a SoC. The only requirement is that our IP core needs to be connected to the bus of a SoC, which is a relatively simple task. In this regard, SCM has some similarities to SecBus [4]. However SCM is more lightweight, and SCM's overhead was evaluated with a prototype hardware implementation. No such evaluation has been published for SecBus.

Our contributions. We present a lightweight IP component to ensure code integrity for SoC designs. It easily composes with existing IP cores using existing SoC design flows. We present a prototype implementation, security analysis, and performance evaluation of the component.

2. PROBLEM STATEMENT

2.1 Threat Model

The goal of the attacker is to execute tampered code on the system. We focus on static, native code, and exclude just-in-time compiled code or self-modifying code. On

ASIA CCS '17, April 4–6, 2017, Abu Dhabi, United Arab Emirates.

DOI: http://dx.doi.org/10.1145/3052973.3053044

many instruction set architectures, instructions alone do not express the semantics of a program efficiently. Instead, instructions are complemented by read-only data. In the remainder of the paper, we use the term "code" as shorthand for instructions and read-only data.

We consider attackers with three powerful capabilities. First, they can tamper with code after it has been built and before it is installed on a device. Second, they are in control of all addressable off-chip memory. Third, they can perform fault attacks on off-chip memory. This includes physical fault attacks, and software-based fault attacks, such as Rowhammer [15, 18].

We use the Dolev-Yao [7] model, which assumes attackers can not break crypto primitives, but can perform protocol-level attacks. We furthermore assume the attacker controls the SoC's digital inputs, such as the General-Purpose Input/Output (GPIO), DDR , and networking signals. While we assume the attacker can perform physical attacks, such as fault attacks and side-channel analysis, on off-chip memory, we assume he cannot perform physical attacks on the SoC itself.

2.2 System Goal

The goal of SCM is to verify the integrity of code stored in off-chip memory. One might argue that mutable data needs to be protected as well, seeing as the initial values of global, statically allocated and initialized arrays also part of the program semantics. Protecting mutable memory is out of the scope. However, it is simple to let compilers generate code such that all static initialization values are stored in read-only data sections, not in mutable data sections.

To enable fluent integration into a SoC, SCM is implemented as a standalone IP core that connects to the bus. By doing so, SCM provides the SoC with security guarantees without requiring modification of other IP cores in the SoC. Furthermore, we aim for minimal disruption of the traditional SoC design cycle, by only building on pre-existing interfaces and composition schemes. This is important, as it enables rapid integration of SCM into SoCs, and improves prototyping, development, and production costs.

SCM should provide protection from the following types of attacks (1) *spoofing*: bits are illegitimately modified, (2) *splicing*: bits are illegitimately relocated, and (3) *replay*: fresh bits (e.g., from an updated program version) are partially substituted with stale bits (e.g., from an outdated program version or another user's program version). SCM works on the principle of verifying the integrity of bits that have been fetched from memory, and more specifically from the code sections and the read-only data sections of binaries as they have been allocated in memory.

In the remainder of this paper we will use the term *memory* to refer to any off-chip memory.

3. SCM DESIGN

3.1 Conceptual Overview

Modern processors issue memory requests to the memory hierarchy to fetch code to be executed. To achieve the system goal, SCM verifies the integrity of code bytes that have been read from an external memory before they are processed by the processor.

The flow of instructions through the system is shown in Fig. 1. Using SCM, memory can be requested from either the

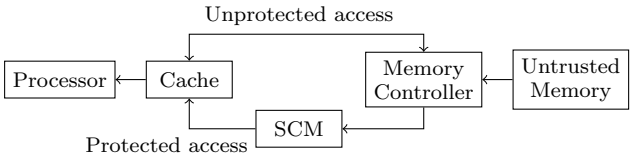

Figure 1: Flow of code and data through system.

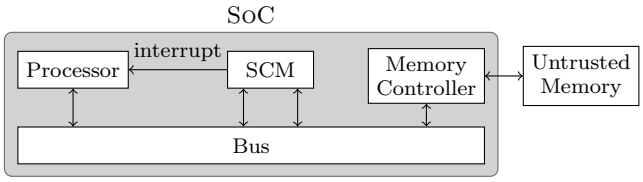

Figure 2: Architectural overview of the system

protected SCM memory region, or from untrusted memory. *Unprotected code* is requested directly from untrusted memory, where it will be stored in caches before being used by the processor. The unprotected parts include the mutable memory regions, but can also include parts of a program that do not require protection. The flexibility to select parts of the software to be protected eases the integration of our solution with existing software.

To access and execute a *protected program*, a group of instructions and integrity information are first fetched from the untrusted memory by the memory controller. Next, those bytes are sent to SCM, which verifies their integrity. If the integrity verification succeeds, the group of bytes is passed on to the caches and the processor. If an integrity check fails, SCM will not forward the tampered instructions to the processor. In addition, the processor is notified by means of an interrupt that a security exception has occurred. The processor can then take appropriate action, depending on the specific use-case for the hardware containing SCM.

3.2 Architecture

The architecture shown in Fig. 2 consists of a SoC and untrusted off-chip memory. The SoC consists of a number of different IP cores, including SCM, a memory controller, and a processor that includes a number of caches. Each of these IP cores are able to communicate via the SoC's main bus. The memory controller acts as an interface between the SoC and the untrusted off-chip memory. SCM is responsible for delivering integrity checked code to the processor.

3.2.1 SCM Memory

SCM has a read-only memory region associated with it, which we call *SCM memory*. Each address in the SCM memory region maps to a physical memory address, as described in more detail below. The physical address can be assigned to any untrusted memory, including ROM, DRAM or flash memory. A bus *transaction* is the sequence of bus actions that are needed to perform a read or write. Whenever a bus transaction requests a read from the SCM memory region, SCM needs to respond by delivering integrity-verified bytes. This non-trivial procedure requires the following steps. First, upon receiving the read request from the processor, SCM needs to fetch the bytes from the matching physical memory address by placing a read request on the

bus. Second, after the requested bytes have been received from the bus, an integrity verification is performed. Third, if the integrity check succeeds, the requested bytes are delivered to the bus. If the integrity verification fails, dummy values are delivered to the bus instead.

Since integrity computations can introduce a large overhead, the integrity checking algorithm and security parameters need to be chosen carefully to ensure a low overhead.

With the addition of the SCM memory region, we effectively split the address range of a program's main memory into a secured and an unsecured region.

3.2.2 Two port interface

SCM needs to perform two simultaneous bus transactions. One transaction is needed for the processor's request to SCM memory, while the second is needed to fetch the code fragment and integrity information from unprotected physical memory. We propose to solve this by using two ports to interface with the bus. Components connected to a bus follow the master/slave communication model. Therefore, we use a slave port to receive read transactions in the SCM memory region, and a master port to perform read transactions from physical memory.

3.2.3 Integrity verification

We propose to use a Message Authentication Code (MAC) algorithm to verify the integrity and authenticity of code. An m-word MAC is *precomputed* on each group of n code words. The MACs are stored interleaved with code in untrusted memory. We use the term *memory block* to refer to the group of n code words and m precomputed MAC words. At *run time*, each memory block's integrity is verified before delivering its code words to the bus.

The MAC algorithm uses a secret key that is deeply embedded in the hardware and is only accessible by the MAC algorithm. In addition, the key is only known by the software provider. Since the MAC key is not known to the attacker, he cannot forge a MAC without being detected.

The system needs to protect against spoofing, splicing, and replay attacks (see Section 2.2). Computing a MAC over the code words ($\text{MAC}(inst_1 \parallel \cdots \parallel inst_n)$), leaves the system vulnerable to a splicing attack, as relocated memory blocks would not be detected. This issue can be exploited by an attacker by rearranging existing memory blocks in order to craft malicious code that cannot be detected by SCM. To prevent this attack, we could use $\text{MAC}(addr \parallel inst_1 \parallel \cdots \parallel inst_n)$, where *addr* is the *physical address* of the memory block. This allows the system to detect any changes to the location of a memory block. However, a replay attack is still possible. Consider the scenario where multiple different programs were transformed under the same key. An attacker can then copy a memory block from one program at $addr_1$ to another program at $addr_1$ without detection. To solve this, we propose to use $\text{MAC}(addr \parallel \omega \parallel inst_1 \parallel \cdots \parallel inst_n)$, where a nonce ω is unique across different programs and different program versions.

3.2.4 Memory map

Each group of n words in the SCM memory region maps to $n+m$ words in physical memory, as shown in Fig. 3. When a group of instructions is fetched from SCM, the m MAC words are stripped out and only the requested instructions are sent to the processor. This has the advantage that the processor works with a continuous address range that does not contain MAC words.

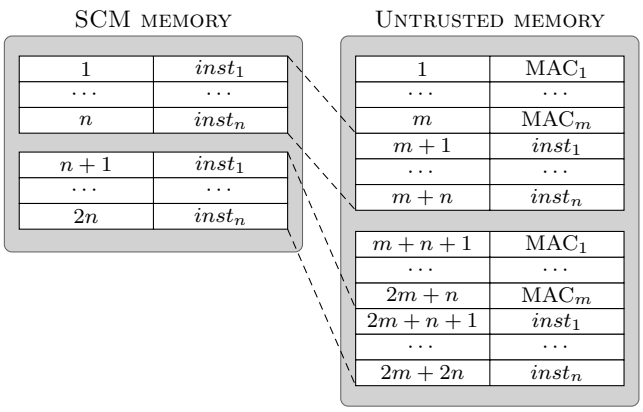

Figure 3: Memory mapping between the SCM memory range and untrusted memory.

As shown in Fig. 1, code typically reaches the processor via caches. To exploit this, parameter n is chosen to match the cache line size. This ensures that each cache line read from SCM memory can be handled by verifying and fetching exactly one memory block.

3.2.5 Software support

The software transformation process is done as follows. First, a customized linker script forces immutable sections in protected segments (in the SCM memory region), while mutable sections are placed in unprotected segments (in unprotected memory). This ensures that the program can execute from the SCM memory region.

Afterwards, the MAC precomputation is done. First, the protected segments of the compiled binary is disassembled. Next, a script calculates a MAC on each group of n opcodes. The MAC is stored interleaved with the opcodes. Finally, the transformed segments as well as the unprotected segments are compiled with a linker script that places both segments in unprotected memory.

Finally, the binary is copied to untrusted memory, and is executed from the SCM memory region.

3.2.6 Integrity failures

If an integrity check fails, SCM needs to initiate an appropriate response to recover from the exception. What is appropriate depends on the use case, and hence will differ for each piece of software. While the development of a recovery mechanism for a specific use case is out of scope for this work, several recovery options are supported by SCM.

One approach is to reset the processor upon detection of an integrity exception. However, this cannot be tolerated by some systems, including safety-critical and real-time systems. Another approach is to reload and restart the program. A more complex option is to increment a counter every time a program is restarted due to an integrity failure, and then rebooting when a threshold value is reached. Some forms of graceful degradation might be useful, or sending notification to online monitoring services.

To provide the necessary flexibility, i.e., to support many forms of reactions in a programmable manner, we designed a generic hardware/software mechanism for SCM to invoke re-

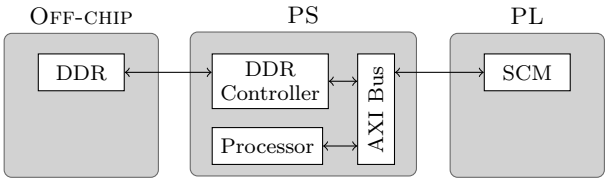

Figure 4: System overview.

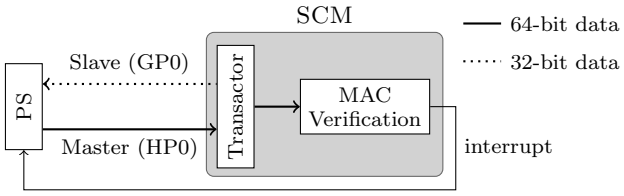

Figure 5: The implemented architecture of SCM.

covery functionality. This mechanism relies on non-maskable interrupts that SCM generates upon integrity failures. When the interrupt occurs, the processor stops executing the current instruction, and transfers control to an interrupt handler. In this handler, any reaction can be programmed, including reloading a program from flash memory to restart a task from a consistent state, or rebooting the processor.

The flexibility of interrupt-based integrity failure handling introduces a potential security problem, as an attacker could also alter the interrupt handler software before an integrity failure occurs. This would prevent the processor from correctly responding to the integrity failure. To address this problem the interrupt handler code can be stored on a small amount of secure memory (e.g., on-chip ROM). Alternatively, a more flexible approach could be to store the interrupt handler in a small SRAM controlled exclusively by SCM. It is then critical that some restrictions be enforced on programming this memory, such as only allowing the memory to be programmed once during boot while the processor is in supervisor mode.

4. PROTOTYPE IMPLEMENTATION

4.1 Target Platform

Zynq SoCs are used to prototype IP cores before fabrication in silicon. Each Zynq SoC contains an FPGA, known as the Programmable Logic (PL), and a non-programmable Processing System (PS), as shown in Fig. 4. The PS features a dual-core ARM Cortex-A9 processor, a memory controller, and ports to communicate with the PL. The AXI4 [2] bus is used for communicating between IP cores located on either the PS or the PL.

AXI4 supports three interface types. The *AXI-Stream* protocol allows two components to communicate without the bus. The *AXI-Full* protocol is used to transfer large amounts of data via the bus, and supports burst mode transfers. The *AXI-Lite* protocol is used for low speed communication, e.g., memory mapped registers, and implements only a subset of the features of the AXI-Full interface.

The PS and PL interface with each other via two different types of AXI-based ports. First, the PS can access PL slave devices via general-purpose (GP) ports. Second, PL master devices can access the PS, which includes off-chip memory, via AXI high-performance (HP) ports. The GP and HP ports both support burst transactions, with data widths of 32-bit and 64-bit, respectively.

As shown in Fig. 5, SCM consists of two subcomponents. The *Transactor* coordinates memory accesses to the PS, while the *MAC verification* component verifies the integrity of memory blocks.

4.2 Transactor

The Transactor handles receives requests from the PS, reads memory blocks from the PS, transfers memory blocks to the MAC verification component, and delivers integrity checked code to the PS.

4.2.1 Interfaces

An AXI-Full slave port allows the PS to read protected code from the SCM memory range. It is configured to allow for 128 MB of SCM memory, which the PS accesses via the 32-bit GP0 port. The 128 MB of SCM memory maps to 160 MB of DRAM, located in the PS.

The Transactor needs a mechanism to fetch a memory block from physical memory after receiving an SCM memory read request. For this mechanism, we evaluated two options. First, using the Xilinx DMA IP core, we measured it takes 60 cycles to receive the first data of a burst read operation from the physical memory. Second, a 64-bit AXI-Full master port connected to the HP0 port requires only 20 cycles to receive the first data of a burst read from the PS. Therefore, the AXI-Full approach was used in our prototype.

After receiving the memory block from the PS, the Transactor passes it to the MAC verification component, which performs the verification before delivering the code to the PS via the slave port. In order to avoid causing a data-abort exception or freezing the PS, it is essential that the slave port responds to read requests with the requested number of memory elements. So upon a verification failure, instead of sending the potentially tampered code bytes, the Transactor simply sends the required number of zero values.

4.2.2 Fetching memory blocks

Each group of n words in SCM memory space maps to a memory block of $n + m$ elements in physical memory (see Section 3.2.4). Since the cache line size of the ARM processor is eight 32-bit processor words, we select $n = 8$. To provide 64-bit security, we use a 64-bit MAC, for which we select $m = 2$. So to serve a read request of 8 words from SCM memory, we need to fetch 10 physical memory words.

With the AXI protocol, the number of words fetched in a burst operation must be a power of 2. We opted to use one 16-word burst of which 6 words are dropped over using an 8-word burst followed by a 2 word burst because every burst involves a 20 cycle delay before the first word arrives, regardless of the burst size. After that initial delay, one word is received every cycle. Furthermore, the extra power consumption of unnecessarily reading six more words is partially compensated by initiating one fewer transaction.

4.2.3 Overlapping read transactions

To allow instructions and read-only data to co-exist inside the SCM memory range, the Transactor's AXI-Full slave port needs to support overlapping read transactions. Such transactions occur when a new read transaction is issued

while the slave is busy processing another transaction. This can happen when a program executing from the SCM memory range executes a load instruction that fetches data from the SCM memory range. This presumably happens when the instruction prefetcher is busy with a speculative fetch from SCM memory, while at the same time a load occurs, causing the memory controller to issue another read request from the SCM memory range.

To support overlapping reads, the Transactor waits for the current read transaction to finish before processing the next. Therefore, registers should be used to store address read channel information, as this could be overwritten when a new transaction arrives.

4.3 MAC Verification

The MAC verification component performs integrity verification of memory blocks. A 64-bit AXI-Stream slave interface is used to receive memory blocks, addresses, and nonces. While data is received from the Transactor, the runtime MAC is calculated. For each memory block, tampering is detected by comparing the runtime MAC to the precomputed MAC. Only untampered code is forwarded to the PS, and detection of tampering fires an interrupt.

For the MAC cryptographic primitive we selected COPA's PMAC1 construction [1]. COPA is an Authenticated Encryption mode of operation for block ciphers, which means it can be used with any symmetric encryption algorithm. However, since we only require authentication, we only use PMAC1, and not the full implementation of COPA.

Although AES is used in COPA's original design, our implementation uses PRINCE [3], which is highly efficient [16]. We placed two pipeline registers inside our PRINCE implementation to allow the MAC verification component to meet the timing constraint of 100 MHz (see Section 5.2). A three cycle implementation of AES will likely have a huge overhead in terms of area and delay, as observed by [16] in a comparison of single cycle implementations. PRINCE's 64-bit block size allows for more effective use of the Zynq's 64-bit HP0 port, since memory blocks received from HP0 can immediately be processed by our MAC primitive.

The nonce is updated by writing to a special SCM register, thereby facilitating context-switches between programs with different nonces.

4.4 Integrity Violations

To handle integrity failures, we configured the PS to allow for fabric interrupts via an IRQ line, and installed a software interrupt handler on the interrupt line. For our prototype, we implemented an interrupt handler that displays a message when such an interrupt occurs.

5. EVALUATION

5.1 Security Evaluation

In SCM, memory tampering and MAC forgery are infeasible, since forged blocks can only be verified online. For an n-bit MAC, an adversary has to perform an average of 2^{n-1} random online MAC verifications before this strategy will succeed [13]. Therefore, a successful forgery of a memory block will require 70,193 years (on average) to succeed on a 100 MHz SCM core.

Table 1: Software benchmarks for SCM

Benchmark	DRAM (cycles)	SCM (cycles)	Reads	Overhead
`qsort v1`	40.35M	40.93M	40.03k	1.43%
`qsort v2`	36.21M	36.20M	18	-0.02%
`sjeng`	22.13G	22.13G	899.59k	-0.02%
`jpeg`	97.08M	97.11M	1462	-0.02%

5.2 Hardware evaluation

We evaluated our design on a ZedBoard, which consists of a Xilinx XC7Z020-CLG484-1 FPGA SoC package. It features a dual-core 667 MHz ARM Cortex A9, 512 MB DDR3, 32 KB L1 cache for each core, and a 512 KB L2 cache. The processor supports prefetching of code and data before they are needed by the processor. The FPGA is comprised of 53,200 LUTs and 106,400 flip flops.

The Xilinx Vivado 2015.2 design suite was used for synthesizing our hardware implementation. It uses an area of 6295 LUTs, and 5880 flip flops. The PS and the FPGA-based PL use a clock frequency of 100 MHz.

5.3 Performance Evaluation

The tool support described in Section 3.2.5 was used to transform the benchmarks. To avoid integrity violations due to the processor's prefetcher issuing reads outside the protected segments, the SCM memory region is expanded by inserting padding data plus correct MACs.

We used the following baremetal benchmarks. First, `qsort v1` [12] performs a Quicksort on 10,000 strings stored in read-only data. Second, `qsort v2` [12] performs a Quicksort on 10,000 strings stored in mutable data. Third, `sjeng` [14] plays a game of chess, and `jpeg` [12] performs jpeg encoding.

Each benchmark was executed from DRAM memory as well as from SCM memory. For the latter, both code (.text) and read-only data (.rodata) were mapped to SCM memory, and measurements were performed in unprotected code. The average overhead is shown in Table 1. The "Reads" column indicate the number of eight word burst reads that was performed by each benchmark. For `qsort v2`, `jpeg`, and `sjeng` we measured an overhead of -0.02%, which is below the noise margin of the performance counters that we used to perform the measurements, as determined by the standard deviations. For `qsort v1` we measured a significantly larger overhead of 1.43%, which is larger than the noise margin of the performance counters. These results show that SCM introduces only a small runtime overhead.

6. RELATED WORK

A large body of work exists on protecting memory or software integrity [19, 20, 5, 9, 21, 10]. However, it appears that most works require modification to the processor, caches, memory, or other IP cores. In contrast, SCM doesn't require modification to existing IP, other than connecting our IP core to the bus. In addition, most works halt the processor after an integrity failure. In contrast, SCM has mechanisms to handle recovery from integrity failures that do not freeze the processor, thereby enabling a wide range of applications, such as real-time systems and systems that cannot handle rebooting the processor. Furthermore, many works were only evaluated in simulation. In contrast, SCM's evaluation was done on an FPGA. This allowed us to provide a detailed

evaluation of the software and hardware overhead in terms of area, and clock frequency.

Domingo et al. [8] detects memory tampering using large encoded trace sequences stored in memory. Elbaz et al. [10] protects memory by encrypting a block of data together with it's address. Intel SGX's [17] Memory Encryption Engine (MEE) [11] uses integrity trees to protect memory. MEE uses expensive components, such as cache, an integrity tree (which requires storage and memory accesses for each transaction), and a TRNG to generate a unique key after each reboot. All of these require extra area, which is not appropriate for small embedded platforms. In contrast, SCM is a light-weight approach that does not make use of such expensive components. SOFIA [6] provides software integrity, software integrity, control flow integrity, tampered code protection, and software copyright protection at runtime. SecBus [4] protects integrity with a component inserted between the memory controller and the bus, which requires modification to the existing architecture. In addition, SecBus can only provide protection to off-chip memory, while SCM can protect any addressable memory region, including on-chip memory.

7. CONCLUSION

In this work we introduced a new hardware-based security architecture called SCM that protects the integrity of code stored in memory. SCM performs MAC-based integrity verifications at runtime and is designed as a standalone IP core that connects to the bus of a System on Chip. We demonstrated the feasibility of using such an architecture by evaluating the design on an FPGA. Our results show a minimal area and performance overhead.

Acknowledgements

We would like to thank Bart Coppens, Koen de Bosschere, and Atul Luykx for their valuable contributions. This work was supported in part by the Research Council KU Leuven: C16/15/058. It is also supported in part by the Flemish Government, FWO G.00130.13N and FWO G.0876.14N, by the Hercules Foundation AKUL/11/19, and by the European Commission through the Horizon 2020 research and innovation programme under contract No H2020-ICT-2014-644371 WITDOM, H2020-ICT-2014-644209 HEAT and Cathedral ERC Advanced Grant 695305. Pieter Maene is supported by a doctoral grant of the Research Foundation - Flanders (FWO).

8. REFERENCES

[1] E. Andreeva, A. Bogdanov, A. Luykx, B. Mennink, E. Tischhauser, and K. Yasuda. AES-COPA v.2. *CAESAR submission*, 2015.

[2] ARM. ARM AMBA AXI and ACE Protocol Specification - AXI3, AXI4, and AXI4-Lite, ACE and ACE-Lite. Technical report, 2011.

[3] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen, and T. Yalcin. Prince: A low-latency block cipher for pervasive computing applications. In *ASIACRYPT'12*, pages 208–225, 2012.

[4] J. Brunel, R. Pacalet, S. Ouaarab, and G. Duc. SecBus, a SW/HW Architecture for Securing External Memories. In *IEEE Mobile Cloud 2014*, pages 277–282.

[5] D. Champagne and R. Lee. Scalable architectural support for trusted software. In *HPCA'10*, pages 1–12.

[6] R. de Clercq, R. De Keulenaer, B. Coppens, B. Yang, P. Maene, K. de Bosschere, B. Preneel, B. De Sutter, and I. Verbauwhede. SOFIA: Software and control flow integrity architecture. In *DATE'16*, pages 1172–1177.

[7] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Trans. on Information Theory*, 29(2):198–208, 1983.

[8] J. Domingo-Ferrer. Software run-time protection: A cryptographic issue. In *EUROCRYPT'90*, pages 474–480.

[9] R. Elbaz, D. Champagne, C. Gebotys, R. Lee, N. Potlapally, and L. Torres. Hw mechanisms for memory authentication: A survey of existing techniques and engines. In *Trans. on Computational Science IV*, pages 1–22. Springer, 2009.

[10] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, M. Bardouillet, and A. Martinez. A parallelized way to provide data encryption and integrity checking on a processor-memory bus. In *DAC'06*, pages 506–509.

[11] S. Gueron. A memory encryption engine suitable for general purpose processors. Cryptology ePrint Archive, Report 2016/204, 2016.

[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. WWC '01, pages 3–14, 2001.

[13] H. Handschuh and B. Preneel. Minding your MAC algorithms. *Information Security Bulletin*, 9(6):213–221, 2004.

[14] J. Henning. SPEC CPU2006 benchmark descriptions.

[15] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory w/o accessing them: An experimental study of DRAM disturbance errors. In *ACM SIGARCH*, volume 42, pages 361–372, 2014.

[16] P. Maene and I. Verbauwhede. Single-Cycle Implementations of Block Ciphers. In *LightSec'15*.

[17] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *HASP'13*, 2013.

[18] R. Qiao and M. Seaborn. A New Approach for Rowhammer Attacks. In *HOST'16*, 2016.

[19] G. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *ACM ICS'03*, pages 160–171, 2003.

[20] P. Williams and R. Boivie. CPU support for secure executables. In *TRUST'11*, pages 172–187, 2011.

[21] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin. Improving cost, performance, and security of memory encryption and authentication. In *ACM SIGARCH*, volume 34, pages 179–190, 2006.