Protecting IoT Devices through a Hardware-driven Memory Verification

Köylü, Troya Çağıl ; Okkerman, Hans ; Reinbrecht, Cezar Rodolfo Wedig; Hamdioui, Said; Taouil, Mottaqiallah

DOI
10.1109/DSD53832.2021.00027

Publication date
2021

Document Version
Accepted author manuscript

Published in
2021 24th Euromicro Conference on Digital System Design (DSD)

Protecting IoT Devices through a Hardware-driven Memory Verification

Troya Çağıl Köylü, Hans Okkerman, Cezar Rodolfo Wedig Reinbrecht, Said Hamdioui, Mottaqiallah Taouil
Computer Engineering, Delft University of Technology
Delft, the Netherlands

Abstract—**Internet of things (IoT) devices are appearing in all aspects of our digital life. As such, they have become prime targets for attackers and hackers. An adequate protection against attacks is only possible when the confidentiality and integrity of the data and applications of these devices are secured. State-of-the-art solutions mostly address software and network attacks, but overlook physical/hardware attacks. Such attacks can still exploit software vulnerabilities or even introduce them. In this paper, we present embedded memory security (EMS); it protects against physical tampering of the memory of IoT devices. As a case study, we have equipped a RISC-V based system-on-chip (SoC) with an EMS module. Our experimental results show that EMS successfully can protect the SoC against hardware tampering attacks, while having a low performance overhead.**

Index Terms—**IoT, external memory protection, confidentiality, integrity, hardware security**

I. Introduction

The number of internet of things (IoT) devices has grown dramatically over the past years [1]. These small autonomous interconnected devices can be found in all areas of the digital world, especially in consumer products [2]. As a consequence, attacks against IoT devices can result in devastating results such as crashes and loss of data [3]. Most IoT attacks look for ways to run malicious code on the target device [3]. This is however not likely to happen through a network access, when the system uses appropriate cryptographic protections, especially during software updates [4]. Software vulnerabilities remain an issue but can be relatively easily patched. However, it is also relatively easy for an attacker to inject code by tampering with the device. There are three common approaches to take control of a device: i) code injection; ii) rogue memory; and iii) replay attack. In code injection, the attacker uses means like fault injection to modify or add malicious code or data into the external memory [5, 6]. In a rogue memory attack, the attacker either changes the content of the whole memory, or swaps the memory chip with a same model containing different contents [7, 8]. In a replay attack, the attacker changes the contents of the memory to an earlier but valid state (such as an earlier firmware with known vulnerabilities) for exploitation [7, 8]. Evidently, protecting the external memories used in IoT devices is of utmost importance.

Throughout the years, several memory protection schemes have been proposed. We can classify them into three groups: trust execution (TE) environments, remote attestation, and usage of cryptographic primitives/functions to validate integrity. An example of a TE environment is ARM TrustZone [9]. A TE uses physical barriers to create two physical zones in a chip: a secure and an insecure zone. Applications running in the secure zone can access all resources (including those residing in the insecure zone), but not vice versa. As such, unauthorized data access by insecure applications are prevented. However, this kind of a countermeasure is simply ineffective against any kind of hardware tampering. In the second group, remote attestation is used to verify the integrity of the nodes by a trusted party. The memory verification can be made with the help of a trusted platform module (TPM), which compares the received data with an already stored and sealed version [10]. This method however requires a significant amount of bandwidth for regular data exchange, which introduces more processing power requirements in the nodes [11]. Furthermore, they are proven to be vulnerable against time-of-check-to-time-of-use (TOCTTOU) attacks, where an attacker modifies the code between the period of code verification and code execution [8]. The third group uses cryptographic functions such as encryption to obfuscate memory contents, as well as secure hash functions to verify the integrity of the data after decryption. For example, in [12] the authors used Advanced Encryption Standard (AES) [13] to encrypt memory contents. Additionally, they used a SHA-1 [14] based hash-tree for verification. A study presented in [15] developed a protection module that encrypts stored memory data and augments it with random tags (that are also stored in the memory), a random key (different per application), and AES. The verification is done by comparing the tag field in the decrypted memory content. Another study presented in [16] used AES in Galois/counter mode for encryption. To provide additional protection, it stores the timestamps of every memory address on chip. These timestamps are used to prevent instruction reuse: a common practice in replay attacks. Such protections indeed offer protection against hardware tampering. However, as these techniques rely on software encryption and decryption, they introduce a large performance overhead. In addition, storing these timestamps on chip for each memory address creates a large overhead. None of the protections above are very suitable for use in IoT devices, which require secure yet lightweight solutions.

In this study we propose a lightweight embedded memory security (EMS) module that effectively protects the external memories of IoT devices. EMS is a standalone hardware component inserted between the processor/cache hierarchy and external memory to check integrity, authenticity and confidentiality of the content from the external memory. Our contributions are as follows:

- Proposal of a hardware protection module that ensures confidentiality, integrity, and authenticity of external memory content.
- Investigation of state-of-the-art lightweight encryption and message authentication code (MAC) functions to implement the proposed solution.
- In-depth evaluation of security, performance, and the hardware overhead of the proposed EMS module, which is applied to a RISC-V processor and implemented on a FPGA platform.

The remainder of this paper is organized as follows. Section II provides a background on lightweight encryption and hash functions. Section III presents the concept, design, and different variants of the proposed EMS module based on lightweight encryption and hashing. Thereafter, Section IV describes the experimental setup and the performed security and performance evaluations. Finally, Section V concludes this paper by discussing the results.

II. Background

This section describes the cryptographic functions considered in this work. First, Subsection II-A describes and compares lightweight block ciphers. Next, Subsection II-B describes and compares lightweight MAC functions.

A. Lightweight Block Ciphers

A block cipher is an encryption/decryption algorithm that processes the input in blocks/rounds. A lightweight block cipher is a cipher that typically requires less resources; hence having a small area, low latency, low power consumption, etc. A typical first choice for encryption in security applications is the Advanced Encryption Standard (AES) [13]. However, having a lightweight hardware implementation was not a design criterion during its development in 1990s. Therefore, the area and power requirements generally do not meet IoT criteria. As a corollary, new block ciphers were developed as lightweight alternatives. The following lightweight ciphers are considered in this work:

- *mCrypton* is a lightweight block cipher based on a substitution-permutation network (SPN). It uses 64, 96, or 128-bit keys to encrypt 64-bit data blocks in 25 rounds [17].
- *Present* is one of the first lightweight block ciphers. It is also based on an SPN, which takes 31 rounds to process 64-bit data blocks using 80-bit or 128-bit keys [18].
- *Piccolo* is based on a generalised Feistel network and aims to create low overhead and energy consumption. It processes 64-bit data blocks in 25/31 rounds for 80/128-bit keys [19].
- *Prince* is designed to provide a high throughput and low latency. It is also based on an SPN, which processes 64-bit data blocks in 12 rounds, using a 128-bit key. However, unlike other ciphers, it processes a data block in a single cycle [20].
- *Rectangle* is a recent cipher based on an SPN. It processes 64-bit data blocks in 25 rounds, using 80 or 128-bit keys. It is specially developed for RFID tags, sensor nodes, and smart cards [21].

TABLE I: Hardware comparison of lightweight ciphers

Cipher	Key	Block	Cycles/block	Throughput	Area (GE)	Efficiency (kbps/KGE)
AES	128	128	226	48	11031	4.35
mCrypton (D)	128	64	13	492.3	4108	119.83
Present	80	64	31	206	2195	93.84
Piccolo (D)	128	64	33	193.8	1362	142.32
Prince	128	64	12	533.3	2953	180.59
Rectangle	128	64	26	246	1787	137.66

Table I compares the lightweight ciphers (and an AES implementation for reference). The data is taken from [22]. The ciphers are presented in the first column. The letter (D) signifies integrated decryption capabilities (i.e., the same component can encrypt and decrypt and hence, no extra component with an inverse operation is needed). The green colored cells show the best values among the ciphers for the criteria and area measures are in gate equivalent (GE). According to the results, Prince is the most efficient cipher for throughput per area. An important point is that only mCrypton and Piccolo have included decryption capabilities in the cipher. On the other hand, Prince can use the same hardware for decryption. As a result, we select Prince as the block cipher in this work.

B. Lightweight Hash/MAC Functions

A hash function is used to map an input to a fixed-length value, which is typically used for integrity checking. A MAC is a hash function that uses a key, and thus, can also verify the authenticity of the data as well. As is the case for block ciphers, lightweight hash and MAC functions require limited resources. A well-known standardized hash function is secure hash algorithm (SHA)-3. It is faster than its predecessors SHA-1 and SHA-2, but has a considerably large hardware overhead. The area optimized variants of SHA-3 still suffer from a large delay. Hence, new lightweight functions were developed to address these issues. The ones that we consider in this work are described next:

- *Armadillo* is a general purpose cryptographic function, which can also be used as a hash. It is especially aimed at RFID tags [23].
- *PHOTON* is a hash function designed for devices with considerable hardware constraints. It is effi-

cient in hashing short messages and comes with low area requirements [24].

- *Spongent* also targets RFID tags. Its construction is based on the Present block cipher (see Subsection II-A) [25]. Among Armadillo and PHOTON, Spongent has the lowest area requirement.
- *GLUON* also targets RFID tags, as well as embedded sensor networks [26]. In most studies, it is compared with PHOTON. Although PHOTON is considered to be more efficient, GLUON is still relevant for practical use.
- *SipHash* is a dedicated MAC function optimized for short inputs, while aiming to be time efficient in software and area efficient in hardware [27]. It was originally created to protect servers against hash collision attacks, but is currently also used in other applications as it is much more efficient than the popular HMAC [28].
- Chaskey is developed as a MAC algorithm [29]. It is designed to provide fast results for software implementations that run on a microcontroller.

TABLE II: Comparison of lightweight hash functions

Name	Tag size (bits)	Block	Cycles/block	Area (GE)
SHA-3 [30]	256	1600	6750	>6500
Armadillo [23]	80	48	44	4030
PHOTON [24]	80	16	132	1168
Spongent [25]	88	8	45	1127
GLUON [26]	128	8	66	2071
SipHash [27]	64	64	12	3700
Chaskey [29]	128	128	NA	NA

Table II compares these hash functions (with SHA-3 as a reference implementation [30]). The data in the table is collected from the different references provided next to the lightweight hash functions. Note that only tag sizes smaller or equal to 128-bit are considered. The green colored cells indicate the optimal selections and NA indicates unavailable data. From the table, PHOTON and Spongent have the smallest area requirements, mainly due to their small block size. GLUON and SipHash both support 64-bit digests (i.e., tags). On the other hand, SipHash has the fastest implementation. As a result, we select SipHash as the MAC function in this work.

III. Methodology

In this section, we explain our embedded memory security (EMS) module for IoT protection. First, Subsection III-A discusses the main concept. Second, Subsection III-B describes the design of the module. Finally, Subsection III-C presents the variations of our module.

A. Concept

To protect the confidentiality and integrity of a resource-constrained IoT node, we propose an EMS module. Figure 1 illustrates the idea. Our module is located between the processor (and caches) and the memory interface to secure data flow. In their perspectives, the module acts as a transparent buffer. It gets the incoming data from the processor/memory, processes them, and relates them to the memory/processor. The module ensures data confidentiality, integrity and authenticity (i.e., prevents running tampered code or code from another device). The module encrypts data and stores MAC tags to guarantee correctness of data, while the processor (and the internal caches) work on unencrypted data, whose integrity was verified by the module.

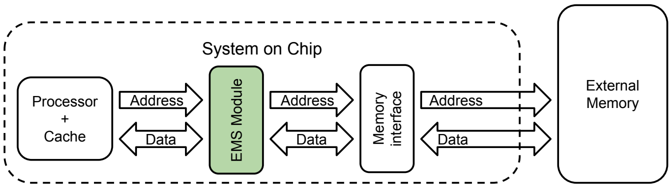


Fig. 1: Our EMS concept, where the location of the proposed module is highlighted with green

B. Design

This subsection provides details on the EMS module. Figure 2 presents an in-depth look into the module and its blocks. To illustrate its functionality, we provide the following example.

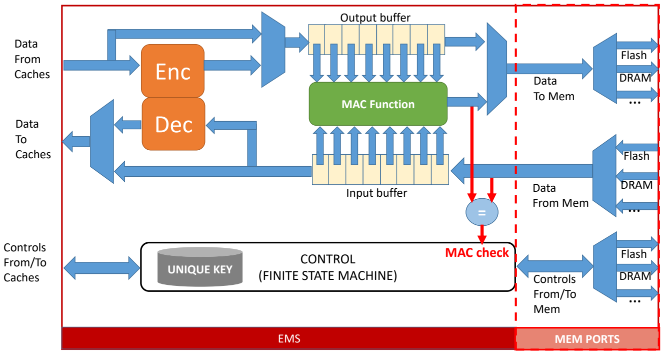


Fig. 2: The EMS module architecture

When the system powers up, the processor starts requesting data from the external memory. In some cases, the first execution code comes from an internal memory, known as the boot memory, that is responsible for providing initialization instructions. When this finishes, the processor is ready to load the operating system from the non-volatile memory (e.g., flash) and store it into the main memory (e.g., DRAM). Any memory request from the processor first goes through the cache hierarchy. After the power-up, this hierarchy is empty, and therefore, any data requests result in a cache miss, and an access to the external memory. In a system with our EMS module, the requests from the last level cache (LLC) go first through the EMS, which then translates these requests to the respective memory controller (e.g, flash or DRAM). When the data from the memory arrives, it is first processed by the EMS module, where the message authentication code (MAC) of this data is calculated. In parallel, EMS requests the MAC value of the data from the memory; the tags are stored on a specific location. When the calculated and received MACs become available, the integrity and authenticity of the data is validated and the data can proceed to the caches/processor.

EMS also provides the option to encrypt and decrypt data. This option should be enabled if the data on the external memories have to be encrypted to provide confidentiality. When enabled, EMS decrypts the incoming data from memory after MAC verification. Likewise, EMS encrypts the data that comes from the processor/LLC, calculates its MAC, and sends these to the external memories. Note that these cryptographic operations (i.e., MAC calculation and encryption/decryption) use a unique key. This key can be created in a number of ways, such as through e-fuses (i.e., the owner burns a unique identification during manufacturing process) or by using the physically unclonable function (PUF) technology [31].

It is important to mention that adding these functionalities (i.e., MAC calculation and encryption/decryption) introduces an impact on the device performance. EMS aims to minimize this impact by considering lightweight approaches. To be precise, it takes only one cycle to encrypt or decrypt 128 bits of data, while it takes about 12 cycles to generate a 64-bit MAC from the same 128-bit data. We also consider that different devices have different security requirements, and hence, we present EMS variants with different features in the following subsection.

C. EMS Variants

We construct six variants of EMS. Each is described next.

- **Unprotected:** This variant is the EMS module without the MAC or encryption/decryption capabilities. Effectively, this variant does not provide data confidentiality and integrity, as the module is effectively reduced to a by-pass module. We use this variant to create a baseline for security and performance comparisons.
- **Cipher:** This variant only ensures the data confidentiality by just including the encryption/decryption operations, although it also offers some integrity protection. As there is no MAC calculation, there are no explicit integrity checks of the data residing on the external memories.
- **MAC-I:** Similarly, this variant ensures data integrity and authenticity by just including MAC verification. As there is no encryption or decryption, the data in the memory can be observed by an unauthorized third party.

  Furthermore, this is a naïve implementation. For each word (memory width) stored in the memory, a MAC is generated and stored in the memory as well at a different location. We assume that the external memory consists of 128 bits per word. As our module (SipHash) generates a 64-bit MAC, it must attach a dummy 64-bit value to it when storing in the memory. As a result, this process requires to double the memory capacity to store the MACs.
- **MAC-II:** This is a memory optimized version of the MAC-I variant; it reduces the memory overhead by 33.3%. Rather than storing one MAC (with padding) per memory location, this variant stores two MACs in the same memory line. Instead of the dummy padding, it places another MAC. Therefore, when it wants to save a MAC calculation, it requires first to read the MAC value of the other word in order to not lose information. As a result, this causes a penalty of an extra read operation when performing a writing operation.
- **Cipher&MAC-I:** This variant ensures data integrity and confidentiality, by including both encryption/decryption and MAC calculations. The MAC operation is done as in MAC-I.
- **Cipher&MAC-II:** This variant uses the memory optimized MAC operation of MAC-II. This is the only difference with Cipher&MAC-I.

IV. Experimental Results

This section describes the experiments we conducted to evaluate the proposed EMS. First, Subsection IV-A describes the experimental setup. Next, Subsection IV-B assesses the security performance of the module. Thereafter, Subsection IV-C evaluates the performance overhead. Finally, Subsection IV-D discusses the hardware overhead that the module creates.

A. Setup

We implemented all variants of EMS in hardware using Verilog hardware description language and integrated them in our system-on-chip (SoC) using the CV32E40P core (formerly RI5CY) [32] as main processor. Our SoC contains UART serial interface, timers and a parametrizable set-associative cache (L1), all interconnected through an AMBA AHB bus. For the experiments, we considered different cache sizes: 4-way set-associative of 2, 4, 8, and 16kB. Additionally, we implemented a main memory using BRAMs with a latency of 100 clock cycles to imitate typical external DRAM behavior [33]. A single address in the main memory stores 128 bits of data, which also corresponds to a single cache line in our platform.

For implementation and simulations, we used Xilinx Vivado 2019.2. We synthesized the designs and emulated them on the PYNQ-Z1 board [34] (Figure 3). Simulations were performed to evaluate the security using three attack cases: i) fault/code/data injection, ii) rogue memory, and iii) replay attack. The emulations were used to measure the performance by running the SoC with our EMS module on the FPGA, while measuring execution times with an internal timer. In these measurements, we used different applications from a public repository of

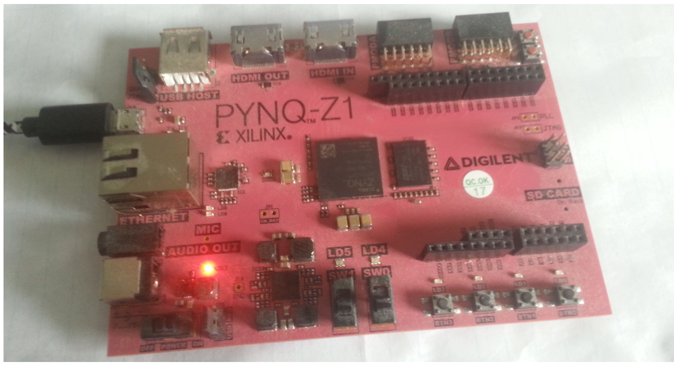


Fig. 3: PYNQ-Z1 Board

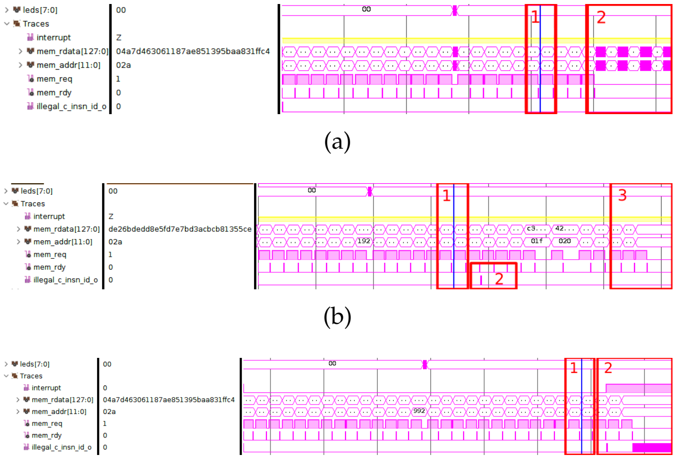
RISC-V benchmarks [35]. They are listed and described below.

- **Median** applies a one dimensional median filter over a 400 element input array. Then it compares the result with another input array for validation. If they do not match, an error is signaled.
- **Multiply** performs an element-wise multiplication between two arrays with a size of 100 elements. Each multiplication is implemented with a shift-and-add algorithm. Like Median, it compares the result with a provided input result array and returns an error if the arrays do not match.
- **Qsort** implements the quicksort algorithm [36] on a 2048 element input array where the sorting is performed in ascending order. Finally, it compares the result with a validation array, and returns the result thereof. This benchmark features the most computationally intensive operation over the ones that we consider.
- **Towers** is a computationally intensive algorithm without inputs. It calculates the moves required to solve the Towers of Hanoi puzzle [37] with 10 rings.
- **Vvadd** is similar to Multiply, except it accomplishes element-wise addition over a 300-element input array. As such, it is not a computationally intensive benchmark.

B. Security Evaluation

The security evaluation tests the effectiveness of the EMS module against attacks. For this, we tested our module under three different attack scenarios that we discussed in Section I. In all scenarios, we assume an attacker targeting the external memory in order to tamper the running software. We do not cover any attacks against the processor or its caches, so we consider the processor as trusted. We assume that our target IoT environment implements all known security measures to prevent network-based attacks.

1) **Fault/Code/Data injection:** The main concern for the situation considered in this work is an attacker that is able to run malicious code in an IoT device. The attacker can achieve this with either fault, code, or data injection. All these can be simulated by injecting faults or data in the memory, especially by targeting the parts used for storing instructions. Therefore, we flipped random bits in 10 instruction memory locations, during a run of the Median benchmark [35] (see Subsection IV-C). Figure 4 shows the results of this attack on three EMS module variants: unprotected, cipher, and MAC-I (see Subsection III-C). We only include these three as MAC-II and cipher&MAC-II are area optimizations without any effect on security (this is also the case for the other attack scenarios). We do not include cipher&hash-I either, as the security results for that variant can be derived from cipher-I and hash-I.



(a)

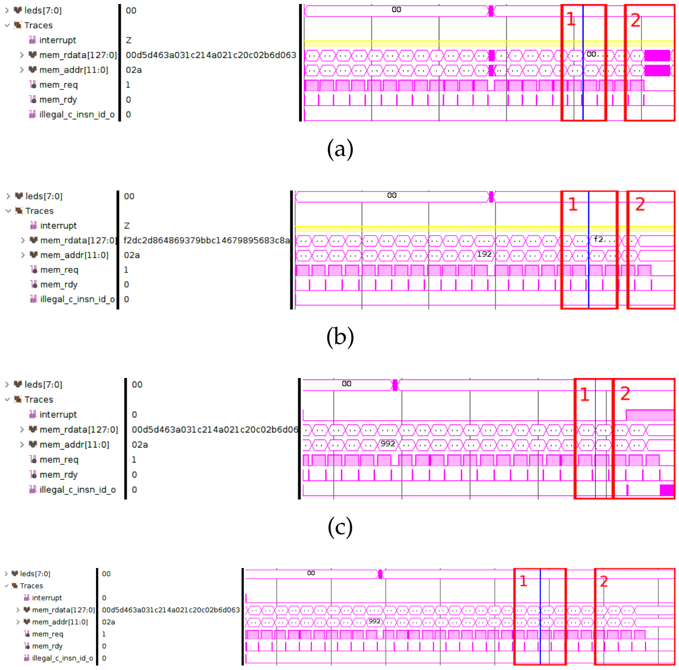
(b)

(c)

Fig. 4: Evaluation results of fault/code/data injection attacks on EMS variants: (a) unprotected, (b) cipher, and (c) MAC-I.

Figure 4 shows the results of the fault injection campaign for the different designs. In (a) - unprotected, the time frame surrounded by a box with number 1 indicates where a glitch took place. The processor deviates from intended behavior some cycles later, as shown by the signals in the box with number 2. In this time window, the processor seemingly reads and writes random data, and does not perform the correct behavior. In (b) - cipher, after the alteration of the memory line during the time window denoted by box 1, the decryption of the following encrypted instruction from the instruction memory leads to an illegal instruction. This results later in a crash, and consequently any further operations are halted (as observed in the time frame surrounded by box 3). In (c) - hash-I, which includes hash-based memory integrity check, the glitched memory (in time frame 1) is immediately detected during time frame 2 and the processor stops executing this application in a controlled manner. The glitch is detected as the calculated and read (from memory) MAC values do not match.

2) **Rogue memory:** A rogue memory attack refers to changing contents of a memory, or even swapping the memory chip with another one. This can result in the execution of malicious applications in an unprotected system. We discuss this attack with a sample application.
This attack against the unprotected variant is trivial and works every time. For this case, the processor executed the tampered program without any crashes or halts. For the cipher case, this attack is not possible as the attacker do not possess the encryption/decryption key of the EMS module. Thus, the result is the same as the fault attack case, where the processor encounters invalid instructions due to an incorrect decryption (see Figure 4 (b)). Lastly, this attack also does not work for the MAC-I case, as the attacker does not possess the MAC key. Hence, operations loaded from the rogue memory will not be validated by the hash block. Consequently, the results for this case is identical to the fault attack case (see Figure 4 (c)).
3) **Replay attacks**: In this attack, the attacker manipulates the same memory by reverting it to an earlier state. Thus, the attacker has access to a collection of valid MACs, which makes the detection harder. However, this attack is also more complicated and limits the attacker in executing custom instructions. Figure 5 shows the evaluation results of four variants under a replay attack. Here, we also included the MAC-II variant, as different memory usage creates a difference in security with MAC-I.
In Figure 5, the attack always occurs at the time frame denoted by box 1. The attack modifies the contents of one particular memory address. The unprotected variant continues the execution as usual, as indicated by the box in time window 2. In the cipher case (b), the swapped memory content still leads to a valid instruction. Hence, the signals during the window surrounded by box 2 show identical behavior like the unprotected variant. The only difference from the unprotected case is that address and data signals from the memory are different. This slight difference is caused by the influence of EMS. In the MAC-I case (c) however, the hash block recognizes the mismatch between the MAC values, and halts the operation. Here, a special case occurs for MAC-II case (d). If the attacker is able to replace both the data and the corresponding hash in the memory, the attack succeeds and the processor continues the execution as normal. This last case can only be avoided if EMS also adds timestamps to the MAC values. We leave this feature for future work. Note that it is not very practical to perform replay attacks on encrypted content, as the attacker should know what the encrypted content represents in order to effectively reuse it.



(a)

(b)

(c)

(d)

Fig. 5: Evaluation results of replay attacks on EMS variants: (a) unprotected, (b) cipher, (c) MAC-I, (d) MAC-II

C. *Performance Penalty*

We evaluate the performance penalty of our EMS module based on the execution time metric. This is the total elapsed time of a benchmark application, from start to finish. As the elapsed time value for a benchmark is data dependent, we provided the same inputs to all variants and took the average of 10 runs. We provide the results for different cache sizes. Table III shows these results for the different EMS variants.

As observable from the table, the full protection schemes (Cipher&MAC-I and II) introduce a delay in the execution time; especially for Median, Qsort, and Vvadd. However, this delay reduces when the cache size increases. Furthermore, variants of Cipher, MAC-I and II introduce considerably less delay. The amount of cache that is needed for a program to reduce the miss rate and hence the execution time depends on the amount of data that needs to be processed, and in which order data is accessed. Based on the execution time of the unprotected case, it is reasonable to assume that a system running these applications would come with an 8kB cache as the execution times do not improve much for higher cache sizes. For this particular cache size configuration, the overhead introduced by the full functionality EMS (Cipher&MAC-I) is between 0.3 and 35%.

D. *Hardware Overhead*

In order to determine the hardware area and timing requirements, We synthesized all EMS variants, along

	Unprotected				**Cipher**			
	2kB	4kB	8kB	16kB	2kB	4kB	8kB	16kB
Median	678	401	277	277	709 (4.6%)	412 (2.7%)	279 (0.7%)	279 (0.7%)
Multiply	751	751	751	751	752 (0.1%)	752 (0.1%)	752 (0.1%)	752 (0.1%)
Qsort	10965	9606	7588	6625	11345 (3.5%)	9891 (2.9%)	7731 (1.9%)	6701 (1.1%)
Towers	2811	2811	2811	2811	2812 (0.1%)	2812 (0.1%)	2812 (0.1%)	2812 (0.1%)
Vvadd	564	308	172	172	593 (5.1%)	320 (3.9%)	174 (1.2%)	174 (1.2%)
	MAC-I				**MAC-II**			
	2kB	4kB	8kB	16kB	2kB	4kB	8kB	16kB
Median	1250 (84.3%)	606 (51.1%)	319 (15.2%)	319 (15.2%)	1355 (99.8%)	638 (59.1%)	319 (15.6%)	319 (15.6%)
Multiply	767 (2.1%)	767 (2.1%)	767 (2.1%)	767 (2.1%)	767 (2.1%)	767 (2.1%)	767 (2.1%)	767 (2.1%)
Qsort	17998 (64.1%)	14874 (54.8%)	10229 (34.8%)	8028 (21.2%)	20000 (82.4%)	16265 (69.3%)	10786 (42.1%)	8038 (21.3%)
Towers	2820 (0.3%)	2820 (0.3%)	2820 (0.3%)	2820 (0.3%)	2820 (0.3%)	2820 (0.3%)	2820 (0.3%)	2820 (0.3%)
Vvadd	1112 (97.1%)	527 (71.1%)	214 (24.4%)	214 (24.4%)	1189 (110.8%)	554 (79.9%)	214 (24.4%)	214 (24.4%)
	Cipher&MAC-I				**Cipher&MAC-II**			
	2kB	4kB	8kB	16kB	2kB	4kB	8kB	16kB
Median	1268 (87.0%)	612 (52.6%)	320 (15.0%)	320 (15.0%)	1372 (102.3%)	644 (60.6%)	320 (15.5%)	320 (15.5%)
Multiply	768 (2.3%)	768 (2.3%)	768 (2.3%)	768 (2.3%)	768 (2.2%)	768 (2.2%)	768 (2.2%)	768 (2.2%)
Qsort	18215 (66.1%)	15036 (56.3%)	10310 (35.9%)	8071 (21.8%)	20217 (84.3%)	16428 (71.0%)	10867 (43.2%)	8081 (21.9%)
Towers	2821 (0.3%)	2821 (0.3%)	2821 (0.3%)	2821 (0.3%)	2821 (0.3%)	2821 (0.3%)	2821 (0.3%)	2821 (0.3%)
Vvadd	1129 (100.1%)	533 (73.1%)	215 (25.0%)	215 (25.0%)	1206 (113.8%)	560 (81.8%)	215 (25.0%)	215 (25.0%)

TABLE III: Benchmark execution times (in µs) for the EMS variants, where the additional time percentage is included for variations other than the unprotected. In the table, kB represents the cache size and NA unavailable data.

with the processor (CV32E40P core [32]) in a PYNQ-Z1 FPGA [34] (see Subsection IV-A). Table IV shows the results for all EMS variants except for the unprotected case, as well as a cost breakdown.

	Slice LUTs	Slice Regs.	WNS @ 25MHz	RAM capacity lost to MAC
Cipher				
Full platform	35176, +9.7%	45025, +2.6%	4.30ns	0.0%
EMS module (x2)	2090	560		
Prince core	1905	130		
MAC-I				
Full platform	34003, +6.1%	46056, +4.9%	3.88ns	50.0%
EMS module (x2)	1505	1084		
SipHash core	863	467		
MAC-II				
Full platform	34197, +6.7%	46461, +5.9%	4.38ns	33.3%
EMS module (x2)	1596	1277		
SipHash core	851	467		
Cipher&MAC-I				
Full platform	36126, +12.6%	46589, +6.2%	4.21ns	50.0%
EMS module (x2)	2563	1346		
Prince core	1526	130		
SipHash core	762	467		
Cipher&MAC-II				
Full platform	36427, +13.6%	46983, +7.1%	1.75ns	33.3%
EMS module (x2)	2713	1539		
Prince core	1590	130		
SipHash core	847	467		

TABLE IV: Hardware area usage of EMS variations and the platform with 2kB, 16-way caches, where the additional overhead is included as a percentage

As expected, the full protection variant creates the most overhead. However, it is limited, as none of our EMS variations have an area overhead of more than 14%.

To further verify that our solution is lightweight, we compared the overhead of the full protection cipher&MAC-II with state-of-the-art solutions. Table V shows the results.

The results indicate that while EMS uses slightly more LUTs, it requires fewer registers. Our module also does not require any block RAM usage. Overall, the usage is very similar to state-of-the-art solutions, if not less. This is because in the table, we only show the synthesis results of the core elements of the state-of-the-art solutions. These do not include control and internal buffers.

	Slice LUTs	Slice Regs.	BRAM
Encrypt+MAC (D)	2713 (100%)	1539 (100%)	0
Prince core	1590	130	0
SipHash core	847	467	0
GCM-AES [38]	2670 (98%)	1568 (102%)	5
SHA-256 [39]	2027 (75%)	1830 (119%)	0

TABLE V: Hardware requirements comparison to state of the art

V. Discussion and Conclusion

In this work, we presented the embedded memory security module that provides data confidentiality, authenticity and integrity in IoT node devices. We accomplished this by introducing data encryption/decryption and MAC calculation blocks between the processor/cache and external memory. Furthermore, we conducted experimentation to show that it is indeed effective against well-known attacks, while being lightweight. To conclude this paper, we highlight the advantages and limitations of EMS:

Customization: A strong point of our detector is its customization ability. A specific variant of our EMS module can be selected based on various factors; such as security/privacy requirements, available processing power, used cache sizes, etc.

Additional hardware security: An interesting side effect of storing data encrypted in the external memory is that it prevents observability from outside. This makes conducting side-channel attacks considerably harder if not impossible, as well as some kinds of fault injection attacks.

Boot memory optimization: As our module provides security to the external memory, the boot software can also be placed there and the size of the boot loader can be significantly reduced if not completely removed. This simplifies the internal SoC structure and reduces area.

Applicability: Our module relies on a secure network implementation and that updates do not contain malicious code. This is feasible in IoT, where secure network and cloud communication can be expected. However, in other digital device networks, additional protections other than EMS are required to ensure software integrity.

Acknowledgment

This work was labelled by the EUREKA cluster PENTA and funded by Dutch authorities under grant agreement PENTA-2018e-17004-SunRISE. Furthermore, the content of this article has been part of the Master's thesis of Hans Okkerman [40].

References

[1] Gartner, Inc, "IoT Security Primer: Challenges and Emerging Practices," 2018, https://www.gartner.com/.

[2] J. Wurm *et al.*, "Security analysis on consumer and industrial iot devices," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2016, pp. 519–524.

[3] J. Deogirikar *et al.*, "Security attacks in iot: A survey," in *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, 2017.

[4] W. H. Hassan *et al.*, "Current research on internet of things (iot) security: A survey," *Computer networks*, vol. 148, pp. 283–294, 2019.

[5] P. Gawkowski *et al.*, "Speeding-up fault injection experiments with dynamic code injection," in *2nd International Conference on Information Technology, (2010 ICIT)*, 2010.

[6] D. Mitropoulos *et al.*, "Fatal injection: a survey of modern code injection attack countermeasures," *PeerJ Computer Science*, vol. 3, p. e136, 2017.

[7] G. Saileshwar *et al.*, "Synergy: Rethinking secure-memory design for error-correcting memories," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[8] A. Seshadri *et al.*, "Scuba: Secure code update by attestation in sensor networks," in *Workshop on Wireless Security*, 2006.

[9] B. Ngabonziza *et al.*, "Trustzone explained: Architectural features and use cases," in *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, Nov 2016, pp. 445–451.

[10] S. Agrawal *et al.*, "Program integrity verification for detecting node capture attack in wireless sensor network," 12 2015, pp. 419–440.

[11] M. Conti *et al.*, "Remote attestation as a service for iot," in *Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, 2019.

[12] G. E. Suh *et al.*, "Efficient memory integrity verification and encryption for secure processors," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, 2003, pp. 339–350.

[13] J. Daemen *et al.*, "Aes proposal: Rijndael," 1999.

[14] G. Leurent *et al.*, "Sha-1 is a shambles," *Online at https://shambles. github. io*, 2020.

[15] R. Elbaz *et al.*, "A parallelized way to provide data encryption and integrity checking on a processor-memory bus," in *2006 43rd ACM/IEEE Design Automation Conference*, 2006, pp. 506–509.

[17] C. H. Lim *et al.*, "mcrypton – a lightweight block cipher for security of low-cost rfid tags and sensors," in *Information Security Applications*, J.-S. Song *et al.*, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 243–258.

[16] J. Crenne *et al.*, "Configurable memory security in embedded systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, 03 2013.

[18] A. Bogdanov *et al.*, "Present: An ultra-lightweight block cipher," in *Cryptographic Hardware and Embedded Systems - CHES 2007*, P. Paillier *et al.*, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 450–466.

[19] K. Shibutani *et al.*, "Piccolo: An ultra-lightweight blockcipher," in *Cryptographic Hardware and Embedded Systems – CHES 2011*, B. Preneel *et al.*, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 342–357.

[20] J. Borghoff *et al.*, "Prince – a low-latency block cipher for pervasive computing applications," in *Advances in Cryptology – ASIACRYPT 2012*, X. Wang *et al.*, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 208–225.

[21] W. Zhang *et al.*, "Rectangle: a bit-slice lightweight block cipher suitable for multiple platforms," *Science China Information Sciences*, vol. 58, 11 2015.

[22] G. Hatzivasilis *et al.*, "A review of lightweight block ciphers," *Journal of Cryptographic Engineering*, vol. 8, pp. 141–184, 2018.

[23] S. Badel *et al.*, "Armadillo: A multi-purpose cryptographic primitive dedicated to hardware," in *Cryptographic Hardware and Embedded Systems, CHES 2010*, S. Mangard *et al.*, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 398–412.

[24] J. Guo *et al.*, "The photon family of lightweight hash functions," in *Advances in Cryptology – CRYPTO 2011*, P. Rogaway, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 222–239.

[25] A. Bogdanov *et al.*, "Spongent: A lightweight hash function," in *Cryptographic Hardware and Embedded Systems – CHES 2011*, B. Preneel *et al.*, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 312–325.

[26] T. P. Berger *et al.*, "The gluon family: A lightweight hash function family based on fcsrs," in *Progress in Cryptology - AFRICACRYPT 2012*, A. Mitrokotsa *et al.*, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 306–323.

[27] J.-P. Aumasson *et al.*, "Siphash: a fast short-input prf," in *International Conference on Cryptology in India*. Springer, 2012, pp. 489–508.

[28] H. Krawczyk *et al.*, "Hmac: Keyed-hashing for message authentication," 1997.

[29] N. Mouha *et al.*, "Chaskey: An efficient mac algorithm for 32-bit microcontrollers," in *Selected Areas in Cryptography – SAC 2014*, A. Joux *et al.*, Eds. Cham: Springer International Publishing, 2014, pp. 306–323.

[30] G. Bertoni *et al.*, "Keccak sponge function family main document," *Submission to NIST (Round 2)*, vol. 3, pp. 320–337, 2009.

[31] U. Rührmair *et al.*, "Pufs at a glance," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2014, pp. 1–6.

[32] *RI5CY: User Manual*, OpenHW Group, 4 2019, rev. 4. [Online]. Available: https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf

[33] B. Jacob *et al.*, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[34] *PYNQ-Z1 Board Reference Manual*, Digilent, 4 2017. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/reference-manual

[35] riscv-tests benchmark repository. [Online]. Available: https://github.com/riscv/riscv-tests

[36] C. Hoare, "Quicksort: Algorithm 64," *Comm. ACM*, vol. 4, pp. 321–322, 1961.

[37] D. R. Hofstadter, *Metamagical themas: Questing for the essence of mind and pattern*. Basic books, 2008.

[38] T. Ahmad. Gcm-aes verilog implementation. [Online]. Available: https://opencores.org/projects/gcm-aes

[39] J. Strömbergson. sha256 verilog implementation. [Online]. Available: https://github.com/secworks/sha256

[40] H. Okkerman, "Embedded memory security: Preventing hardware based attacks on the memory of iot devices," Master's thesis, Delft University of Technology, the Netherlands, 2020.