Memory Authenticated Encryption Engine for a RISC-V processor

Karim Ait Lahssaine and Olivier Savry

Univ. Grenoble Alpes, CEA, Leti, F-38000 Grenoble, France *
Email: firstname.name@cea.fr

Abstract

In this paper, we present the Memory Authenticated Encryption Engine (MAEE) hardware countermeasure to ensure the confidentiality and authenticity of data in RAM and the associated interconnect bus. Using the Subterranean 2.0 authenticated encryption algorithm, data used by a processor is secured at the output of cache memory, and stored in memory as chunks, containing encrypted data and metadata for authenticity verification. The MAEE provides protection against attacks targeting the memory and its bus, such as Rowhammer, fault injections or side-channel attacks. We are also evaluating the performance of this countermeasure, by associating it with the RISC-V CVA6 application core.

Introduction

The issues surrounding the cybersecurity of microprocessors are becoming increasingly significant as a result of the multiplicity of attacks. In response, countermeasures are being developed to secure the critical modules of the architecture. Among these countermeasures, some secure the memory, by encryption solutions, thus guaranteeing the confidentiality of stored data. But this is not enough, it is also necessary to guarantee the integrity of the data to counter fault injection or Rowhammer attacks and to ensure the authenticity of the programs. Finally, in addition to memory, the memory bus must also be secured, as it's a place where side-channel leaks can occur.

However, this countermeasure must limit the performance degradation of the host system, such as latency or memory and logic footprint. The hardware countermeasure Memory Authenticated Encryption Engine (MAEE) was designed on the basis of these two observations. In a first step, we introduce this countermeasure, the technological choices made and the associated constraints. Secondly, we present its implementation and its performance.

Authenticated Encryption

In order to address the problem, countermeasures exist, but as shown by [1], most encrypt word by word and associate a MAC (Message Authentication Code) with each word, which is prohibitive in terms of memory footprint. To limit the cost, authenticated encryption algorithms are the preferred solution, because in addition to encrypting the data per chunk, they associate a MAC (Message Authentication Code) with the entire chunk. Several algorithms exist, but the one chosen must be lightweight. The choice was made from the list of candidates for the NIST Lightweight Cryptography (NIST LWC) competition.

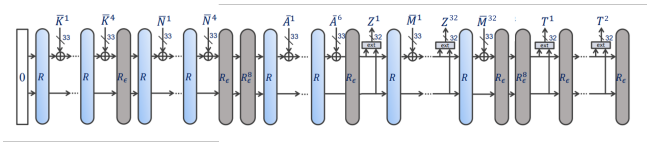

Figure 1: *Subterranean 2.0 algorithm*

After studying the algorithms and their benchmark[2], we chose Subterranean 2.0[3], for its small logical footprint, its high data rate and the absence of security holes in the state of the art. In addition, Subterranean 2.0 has a shorter initialization time and allows 2 rounds to be instantiated for the same cost as a single ASCON round.

This algorithm, as shown in Figure 1, is made up of several rounds, and between them data is absorbed or extracted. In the initialization for the encryption/decryption, in order, a 128-bit key and a 128-bit nonce are absorbed. The nonce is composed of the address, nonce1, and a random part updated at each write, nonce2. Nonce1 prevents block swapping. Next, the metadata, the decrypted/encrypted data is absorbed and the encrypted/decrypted data is extracted. Finally, a 64-bit MAC is extracted.

For the initialization of the algorithm, 18 rounds are needed, which is very expensive. In order to minimise this cost, we need to do as little initialisation as possible, for a given number of read/write operations in memory. So instead of reading/writing a single word, we will read a whole set of words, called a chunk, and this set has the size of a cache line.

*This work was funded thanks to the French national program "Programme Investissement d'Avenir IRT Nanoelec" ANR-10-AIRT-05.

The encryption takes 40 cycles, and as we choose a chunk size of 160 bytes, with 128 bytes of data (16x64 bits), we have a throughput of 3.2 bytes per cycle.

In 32 bytes of metadata, (see Figure 2), there is a part of the nonce(64 bits), ptr_id pointers for each 64-bit data (16x8 bits) and the MAC (64 bits).

The ptr_id are optional and used to implement a tagged memory to avoid spatial and temporal leakage. The nonce stored in memory is the random part of the 128-bit nonce, the other part is extracted from the address.

Figure 2: *Memory chunk (20*64 bits)*

Implementation

For the implementation, two MAEEs are used, one for the write bus and one for the read bus, to avoid introducing a bottleneck. They are placed between the cache and the interconnect bus, to avoid leakage into the bus. On a write, a 128-byte cache line passes through the MAEE, and an authenticated 160-byte encrypted chunk comes out to be stored in memory. And vice versa for a read.

Encryption and metadata addition are transparent to the processor. Firstly, thanks to an address translation, which is light due to the size of the chunk chosen. Secondly, thanks to the chunk configuration which allows on-the-fly encryption, as it provides the data in the order desired by Subterranean 2.0.

In addition, in order to improve latency, 2 hardware rounds are implemented instead of one, in order to (de)encrypt 64 bits per clock cycle, and thus use the full width of the bus without the need to buffer. The MAEE, Figure 3, consists of the following modules :

- **Subterranean :** Main module that contains the 2 rounds and performs authenticated encryption
- **Address Translation :** Performs light address translation
- **RNG :** Generates a seed at processor startup that is used as the first nonce, then at each write the MAC is accumulated to this seed, to generate the random part of the following nonces.
- **Masking Unit :** Generates masks to send the masked data to the cache.

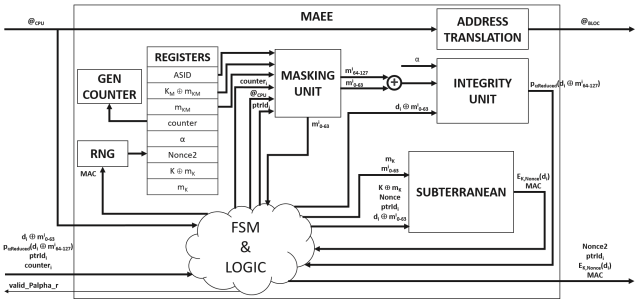

Figure 3: *MAEE inner modules*

- **Integrity Unit :** Generates an integrity tag for each piece of data sent to the cache.

Performance The two MAEEs were implemented on Genesys2 FPGA board, associated with a RISC-V CVA6 core. In addition to adding the MAEEs, we have to choose the write-back cache, and modify it to have a 128 bytes cache line, and handle atomic operations.

After emulating the SoC (CVA6 + MAEE + Peripherals including memory) on FPGA, and running a Linux kernel, we obtain the following performances Table 1.

	Without MAEE	With MAEE	Overhead %
SoC LUTs	86743	90889	4.78
SoC FFs	53593	54840	2.34
DRAM size	819 MB	1 GB	25
Linux Boot Time	2 min 57 s	3 min 17 s	9.7
Read Rate	56.26 MB/s	50.68 MB/s	-9.92
Write Rate	46.87 MB/s	37.70 MB/s	-19.56

Table 1: *MAEE Performances on CVA6*

The overhead for the logical footprint is less. For the memory footprint, it is 25%, which is important but it must be compared to existing authenticated encryption with an overhead of 100%. For throughput, the results were obtained with the RAMspeed benchmark on Linux (INTmark with 1Gbytes per pass)[4]. We can see that the overhead for writing is twice as high as for reading, this is explained by the choice of the WB cache which obliges to pass twice through the MAEE during a write.

In conclusion, our countermeasure has little impact on the hardware footprint, but significantly degrades throughput, especially write throughput. The presence of an L2 cache can greatly improve this situation. It remains to see the performance with a benchmark suite such as SPEC, and to perform a security evaluation.

References

[1] Pascal Nasahl et al. "CrypTag: Thwarting Physical and Logical Memory Vulnerabilities Using Cryptographically Colored Memory". In: (2021).

[2] Kamyar Mohajerani et al. "FPGA Benchmarking of Round 2 Candidates in the NIST Lightweight Cryptography Standardization Process: Methodology, Metrics, Tools, and Results". In: (2020).

[3] Joan Daemen et al. "The Subterranean 2.0 Cipher Suite". In: *IACR Transactions on Symmetric Cryptology* 2020.S1 (June 2020), pp. 262–294.

[4] Alasir. *RAMspeed, a cache and memory benchmarking tool*. v2.6.0. 2009. URL: `https://github.com/cruvolo/ramspeed`.