Transparent Memory Encryption and Authentication

Mario Werner, Thomas Unterluggauer, Robert Schilling, David Schaffenrath, and Stefan Mangard
Graz University of Technology
Email: {firstname.lastname}@iaik.tugraz.at

***Abstract*—Security features of modern (SoC) FPAGs permit to protect the confidentiality of hard- and software IP when the devices are powered off as well as to validate the authenticity of IP when being loaded at startup. However, these approaches are insufficient since attackers with physical access can also perform attacks during runtime, demanding for additional security measures. In particular, RAM used by modern (SoC) FPGAs is under threat since RAM stores software IP as well as all kinds of other sensitive information during runtime.**

To solve this issue, we present an open-source framework for building transparent RAM encryption and authentication pipelines, suitable for both FPGAs and ASICs. The framework supports various ciphers and modes of operation as shown by our comprehensive evaluation on a Xilinx Zynq-7020 SoC. For encryption, the ciphers Prince and AES are used in the ECB, CBC and XTS mode. Additionally, the authenticated encryption cipher Ascon is used both standalone and within a TEC tree. Our results show that the data processing of our encryption pipeline is highly efficient with up to 94 % utilization of the read bandwidth that is provided by the FPGA interface. Moreover, the use of a cryptographically strong primitive like Ascon yields highly practical results with 54 % bandwidth utilization.

***Index Terms*—RAM, encryption, authentication, Zynq, FPGA**

I. Introduction

Security is becoming an increasingly important aspect of FPGAs as many FPGA applications operate on valuable Intellectual Property (IP) and process sensitive data in hostile environments. For example, devices in the Internet of Things (IoT) are often physically accessible by numerous people that thus get capable of stealing hard- and software IP or other sensitive data. Similarly, many machine vendors use FPGAs to control the production units that are shipped to potentially hostile customers interested in sensitive parameters.

Common countermeasures to protect IP against physical access are bitstream encryption/authentication [14] and secure boot [13]. These ensure the confidentiality and authenticity of any IP deployed to FPGAs. However, most of nowadays applications based on (SoC) FPGAS also use Random Access Memory (RAM) to process the increasing amounts of data. This allows attackers with physical access to an FPGA to read from and/or tamper with sensitive data in RAM. Yet, current FPGAs do not protect data that is stored in RAM during runtime.

There already exist several encryption and authentication techniques to protect data in RAM. For example, CBC-ESSIV [7], XEX [10], XTS [1], and the counter mode [4, 11, 12, 15] have been proposed for RAM encryption. While these modes ensure confidentiality, none of them provides authenticity. Even worse, certain modes like counter mode encryption even lose confidentiality in case of active attacks such as spoofing, splicing and replay attacks [5]. In *spoofing attacks*, an attacker simply replaces an existing memory block with arbitrary data, in *splicing attacks*, the data at address A is replaced with the data at address B, and in *replay attacks*, the data at a given address is replaced with an older version of the data at the same address. To protect against these active attacks, various tree-based RAM authentication techniques, e.g., TEC trees [6], exist.

While previous work [4, 11, 12, 15] continually improved RAM encryption techniques, virtually all lack practical implementations and do only simulations to estimate the performance. On the other hand, recent implementations of RAM encryption and authentication as in, e.g., Intel SGX [8], AMD [9], remain closed source. Yet, there is a strong need for freely available implementations given the threat of physical attacks and the proliferation of both (SoC) FPGAs like Xilinx Zynq and increasingly relevant open-source hardware projects as RISC-V.

In this work, we present a modular open-source [1] framework for transparent RAM encryption and authentication which is configurable for different ciphers, cryptographic modes, and block sizes. The building blocks of our framework are written in VHDL and are suitable for both ASIC and FPGA designs. We evaluate our framework to give the first comprehensive comparison of performance results of practical implementations of RAM encryption and authentication in various cryptographic configurations. For evaluation, we use the Xilinx Zynq platform and let the ARM CPU access the memory via our transparent memory encryption module in the FPGA. At 50 MHz, our implementations of different cryptographic modes using Prince [2] and AES give an performance upper and lower bound of 187 and 35 MB/s read bandwidth, respectively. We further show that the Authenticated Encryption (AE) cipher Ascon [3] gives very practical results for RAM encryption and authentication when replay attacks are not concerned. For applications further threatened by replay attacks, we provide an Ascon-based implementation of the TEC tree, reaching up to 47 MB/s read bandwidth.

The remainder of this article is organized as follows. Section II gives details about the memory encryption framework and Section III describes the extensions for authentication trees. The evaluation and conclusion are content of Section IV and Section V, respectively.

II. RAM Encryption Framework

RAM is in general a very fast and heavily used system resource. Transparently encrypting it, by placing an encryption

[1] https://github.com/IAIK/memsec

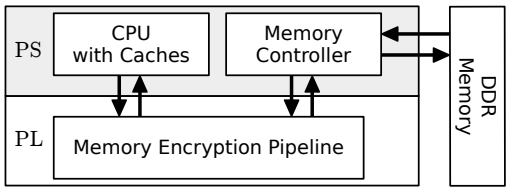

Fig. 1. Zynq platform with memory encryption module.

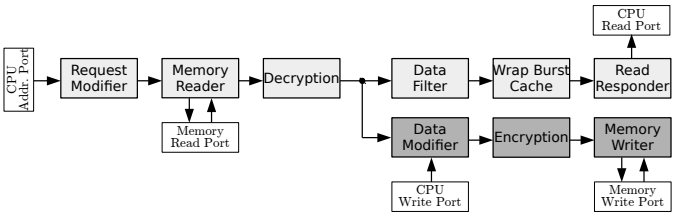

Fig. 2. Simple AXI4 memory encryption pipeline which processes write requests using a read-modify-write approach.

pipeline between CPU and memory controller (outlined in Fig. 1), is therefore a challenging task. This section discusses the various challenges involved and gives details on the functionality and design rationales behind our framework.

A. Challenges

In modern FPGAs, RAM is typically exposed to the programmable logic via memory controllers which feature standard bus interfaces (e.g., AXI4, Avalon, ...). Using such an interface, reading from or writing to memory can be performed by simply issuing the respective bus request. Even though in practice most of the memory requests have a well defined format (e.g., processor cache lines), there are in general no restrictions regarding alignment and request size. On the other hand, cryptographic primitives always have alignment and block size requirements which have to be matched. These diverging constraints make the transparent encryption of RAM quite challenging. Additionally, some ciphers and modes of operation additionally require metadata (e.g., counters, nonces, tags) to operate correctly. Processing this metadata at the correct time is essential to achieve good performance and complicates the issue of data alignment even further.

Finally, many optimizations and peculiarities of the used bus architecture itself have to be considered. A common performance tweak to speed up cache line fills is, for example, the use of wrapping burst. Such burst are problematic given that the requested data order does not match the order of the data in memory. Other peculiarities, which have to be considered for memory encryption, are for example write strobes, narrow transfers, and even complete interface width mismatches. In summary, every possible request which can be issued via the bus interface also has to be supported with transparent memory encryption in place.

B. Framework and Application to AXI4

Even though each individual challenge is minor, the overall resulting complexity is quite high. To cope with this complexity, a divide and conquer approach is used in our framework. The result is a comprehensive collection of modular building blocks which individually implement very limited functionality. However, arbitrary memory encryption units, with support for any cipher and encryption mode, can be built by arranging the individual modules in a pipeline structure.

Key to this flexibility are fully synchronized, unidirectional stream interfaces to interconnect the building blocks. On the majority of blocks, these stream interfaces receive and forward metadata (e.g., addresses, lengths, flags, ...) as well as a configurable amount of memory data (i.e., depending on the external interface widths). The synchronization ensures that neither timing issues nor congestion cause data to be lost. Furthermore, registers and FIFOs can be placed at arbitrary positions to cut combinatorial paths and to decouple the individual modules for better performance.

Transparent memory encryption for the AXI4 bus can, for example, be realized using a pipeline as shown in Fig. 2. The depicted pipeline provides one slave and one master interfaces (see boxes with white background). The boxes which are shaded in light gray are used for reading from encrypted memory. Blocks which are shaded in dark gray are dedicated to writing to encrypted memory.

The slave interface (denoted as CPU) receives unencrypted requests that are serviced like without memory encryption. The master interface (denoted as Memory) on the other hand is used to actually store the encrypted data to the physical memory. The pipeline in Fig. 2 is able to deal with all the previously discussed challenges and supports the use of arbitrary block-based cryptographic primitives or modes. Alignment and block size mismatches are addressed by artificially widening every request during request modification. For memory writes, this leads to the need for a read-modify-write (RMW) approach when writing small data fragments. Interestingly, a RMW approach is required for AXI4 in any case to properly support write strobes. Therefore, all memory writes in this example pipeline are performed as RMW, which even permits to reuse the request modification, the memory reader, and the decryption for writes. The following types of building block categories are provided by the framework:

Bus Interface: Depending on the FPGA vendor, different types of bus interfaces are used to interact with memory. To outsource this dependency, interface converters are needed to establish the connection between the external bus interfaces and the framework-internal stream interface. The bus interface plays a major role in the framework on both the unencrypted slave and the encrypted master side. On the slave side, converters are involved in the translation of the initial request, the decoding of written data, the encoding of read data, and the error reporting. Similarly, on the master side, support for performing memory reads and writes (e.g., request issuing, data encoding, response processing, ...) is required.

Request Modification: Probably the most important part of the memory encryption pipeline is request modification. In this step, requests from the slave interface are translated to

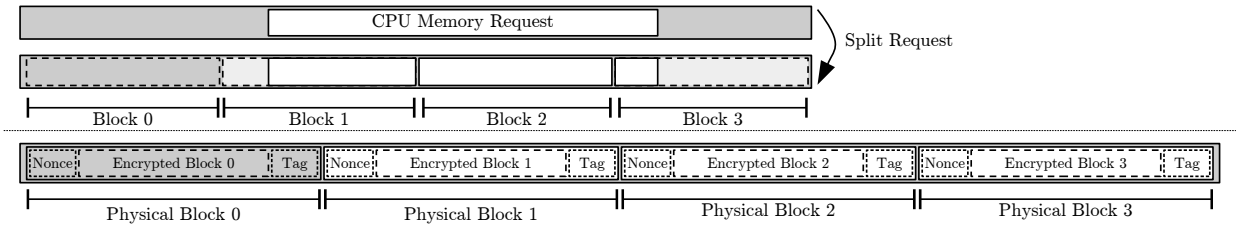

Fig. 3. Request modification for a nonce based encryption and authentication scheme like Ascon [3]. CPU memory requests are split into chunks with additional alignment constraints to incorporate metadata for the AE scheme.

the requests on the master interface. This translation takes the memory alignment and block size requirements of the employed cipher mode into account and widens the requests accordingly. Additionally, also metadata is considered in cases where the encryption scheme is not length preserving and even additional requests can be injected into the pipeline when needed.

An example for a translation, suitable for a nonce-based authenticated encryption scheme, is shown in Fig. 3. In the first step, the actually received CPU request is split based on the data block size of the cryptographic primitive. This splitting determines which logical blocks are affected by the request and have to be fetched. In the second step, taking into account the logical blocks and the amount of required metadata, it is then possible to determine the actual physical memory request. Note that during request modification only the size of the metadata is important. The actual semantic and positioning of the metadata within the physical block on the other hand is not.

En-/Decryption: Encryption and decryption blocks contain the actual ciphers which can typically be further decomposed into a cryptographic primitive and a suitable mode of operation. The cipher blocks only have to support encryption/decryption of memory requests with alignment and block size appropriate for the respective primitive, which greatly reduces implementation complexity. Furthermore, the actual layout of each block (e.g., what bytes are metadata) can be freely defined by the cipher blocks. However, to keep latency as low as possible it is advised to interleave the metadata with the cipher text. By doing so, the metadata arrives at the cipher exactly in the moment it is actually needed. Fig. 3 shows such an interleaving for a nonce based authenticated encryption scheme like Ascon. In this example, the nonce, used for initialization, is placed at the beginning and the tag, used for verification, is placed at the end of each block.

Data Stream Modification: Operating with the data which passes through the pipeline is another important part of the framework. Therefore, various building blocks which transform the data stream are provided. This includes support for injecting new data beats into the stream, for dropping existing data beats, for zero initializing whole requests, for filtering data based on the address, and for replacing individual bytes by taking into account address and write strobe information. Furthermore, the support for reordering individual data beats, which is needed to process wrapping bursts efficiently, can be assigned to this category of building blocks.

Misc.: In addition to the main building blocks, also a comprehensive selection of supporting building blocks is provided by the framework. These blocks provide common functionality to the main blocks and are further handy for newly developed components. Examples for such supporting blocks are synchronization primitives for handshake signals, register stages with synchronization, and serialization as well as deserialization blocks for data rate conversions.

C. Optimizations

Performance optimization is in general a tough challenge given that detailed knowledge about the usage profile is required. However, some simple tweaks can also be performed by exploiting knowledge about the used hardware. For example, a CPU cache with AXI4 interface typically refills cache lines by using wrapping bursts to decrease latency. The framework's `WrapBurstCache` permits to implement such bursts efficiently (i.e., single memory read) by reordering data beats within the pipeline.

Another important property of the framework regarding optimization is that each building block is highly configurable via VHDL generics. This not only is a necessity to support various ciphers, but also permits to perfectly adopt a memory encryption pipeline to the expected workload. Aligning the cipher block size (excl. metadata) with the expected request size (e.g., cache line size), for example, typically maximizes the performance.

III. Authentication Trees

In this section we extend our pipeline in Fig. 2 to implement authentication trees that provide replay protection.

A. Requirements

The pipeline depicted in Fig. 2 facilitates the implementation of various variants of RAM encryption and authentication that provide RAM confidentiality and protection against active RAM spoofing and splicing attacks. However, many applications also require protection against replay attacks, where an active attacker replaces parts of the memory with valid cipher texts (and tags) observed at a previous point in time. Such feature can be obtained from using an AE scheme like Ascon in an extended version of the pipeline in Fig. 2. Namely, this pipeline must store all nonces securely on the FPGA such that they cannot be modified by attackers. In this way, nonces cannot be replayed and any malicious modification is detected. However,

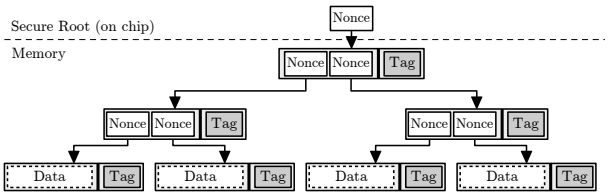

Fig. 4. Binary Tamper Evident Counter (TEC) tree.

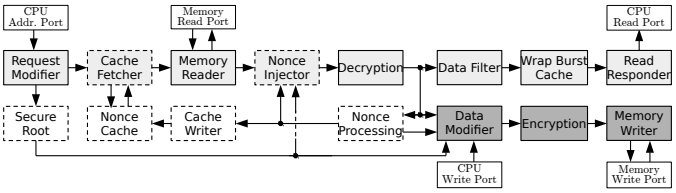

Fig. 5. Memory encryption and authentication pipeline.

since the amount of available secure storage is typically limited, RAM authentication with replay protection usually relies on authentication trees. By storing every block in RAM within an authentication tree, only the tree root must be stored in a trusted environment. Since the tree root reflects the current state of the tree and is authentic, any tampering in RAM can be detected.

B. Functionality

The authentication tree used in this work is a variant of the Tamper Evident Counter (TEC) [6] tree as depicted in Fig. 4. Hereby, the data in RAM is split into blocks and the blocks are authenticated and encrypted using the AE cipher Ascon. The nonces required for AE are recursively stored in a tree where all nodes are authenticated and encrypted as well. The root nonce is stored on the trusted FPGA chip. The tree nodes themselves are stored after the data nodes and are located at the end of the RAM.

To support TEC trees, the pipeline in Fig. 2 is extended as shown in Fig. 5. Its basic data flow is thus identical. Light and dark shades denote modules required for read and write operations, respectively. Modules required to support authentication trees are depicted with dashed edges. For both read and write accesses, the pipeline traverses the tree in a single pass from the root to the respective data leaf node by decrypting (and updating) the nonce of the next lower level and verifying (and computing) the respective node's tag. Hereby, the `RequestModifier` injects all tree node requests additionally required to decrypt and authenticate the data requested by the CPU. Besides, authentication errors of all nodes accessed in the read part are accumulated to form the CPU response.

However, for uninitialized memory tag verification will typically fail. Hence, zero-valued nonces are used to tag uninitialized memory. This works as follows. The root nonce is initialized zero upon startup. On read accesses, verification errors are suppressed whenever a zero-valued nonce is encountered on the path from the root to the accessed leaf. On write accesses, the plain texts of all nodes are initialized zero on the path from the root to the leaf from the point on when a zero nonce is encountered, and the actual write path is written as desired. In this way, uninitialized subtrees are automatically assigned a zero nonce.

C. Optimizations

Authentication trees as in Fig. 4 have significant memory and performance overheads due to the processing of the additional tree nodes. An important parameter to reduce this overhead is the tree arity since it has direct influence on the tree height. The implementation thus allows the configuration of different tree arities to find the setup with lowest overhead.

Furthermore, on the implementation side, the design in Fig. 5 uses a `NonceCache` to optimize read performance. The `NonceCache` is directly mapped using the least significant RAM address bits and can be configured for different sizes. However, the `NonceCache` does not improve write performance. An optimization that improves both read and write accesses are multiple tree roots in the `SecureRoot`. While multiple roots increase the demand for secure on-chip storage, they effectively reduce the tree height. Our implementation can be configured for an arbitrary number of on-chip roots.

IV. Evaluation and Discussion

The proposed framework has been evaluated using a ZedBoard featuring a Xilinx Zynq XC7Z020 SoC and 512 MB DDR3 RAM. This SoC provides a dual core ARM Cortex-A9 processing system (PS) and a Xilinx Artix-7 programmable logic (PL) which are connected using AXI interfaces. To ease comparison, all designs have been evaluated at 50 MHz FPGA frequency, provide 256 MB of protected memory to the ARM processors in the PS, and use the 32-bit GP0 interface to the CPU as well as the 64-bit HP0 interface to the memory. Note that operating the 32-bit interface at 50 MHz limits the maximum achievable bandwidth between processor and memory to 200 MB/s. As benchmarks, tinymembench 0.3 [2] and LMBENCH 3.0-a9 [3] have been used on top of the Xilinx Linux kernel 4.4 (tag 2016.2) [4]. Note that not only the benchmarks, but also the operating system itself has been executed within the transparently encrypted memory.

Cipher Modes and Configurations: The performance of the design in Fig. 2 has been evaluated with the Ascon AE cipher (64-bit nonce and tag) as well as the block ciphers Prince and AES in ECB, CBC-ESSIV, and XTS mode. Additionally, the design in Fig. 5, which provides full memory encryption and authentication using an 8-ary TEC tree with 1024 roots, has been measured using Ascon as the cryptographic primitive. Depending on the cipher, different implementation strategies (Prince = fully pipelined, Ascon + AES = round based) have been used.

However, we stress that these cryptographic primitives have only be chosen to evaluate how the performance of the

[2] https://github.com/ssvb/tinymembench
[3] http://lmbench.sourceforge.net
[4] https://github.com/Xilinx/linux-xlnx.git

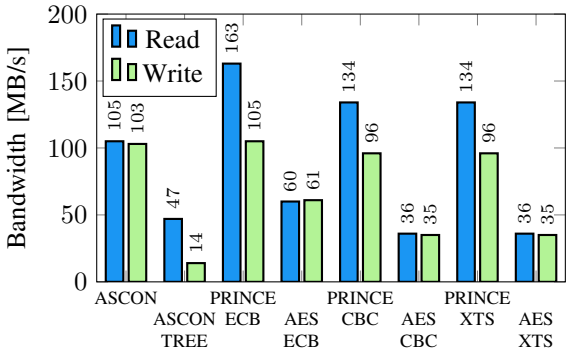

Fig. 6. Memory bandwidth determined with tinymembench (`NEON read prefetched (64 bytes step)`, `NEON fill`).

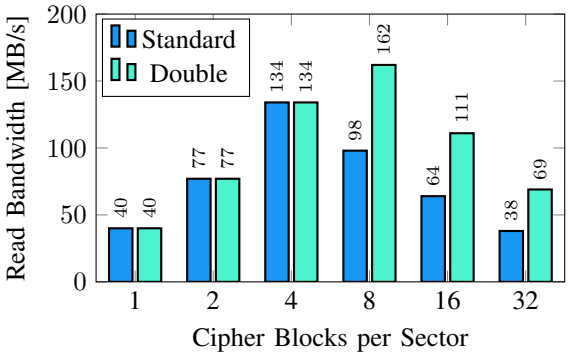

Fig. 7. Memory bandwidth determined with tinymembench of Prince CBC with different block sizes and cache controller configurations.

encryption pipeline is affected by the cipher/mode. Namely, from the security point of view, we pronounce against using AES and Prince in ECB mode. Furthermore, using Prince in CBC and XTS mode is also not recommended given that the cipher does not offer related-key security. In particular, both CBC and XTS use Prince in a related-key setting where the whitening key is either eliminated or tweaked.

All configurations have been evaluated in their most promising parameterization. In most cases, this corresponds to aligning the block size of the cryptographic mode with the processor's last-level cache line size (32 bytes). The only exception is the Ascon-based TEC tree, which is configured with a data block size of 64 bytes.

Bandwidth and Parameter selection: Fig. 6 depicts the memory bandwidth of the various ciphers and modes of operation. The results for Prince clearly dominate the comparison, reaching between 82 % and 67 % of the maximum possible read bandwidth. This is due to the the fully pipelined implementation which features only two cycles latency. The performance achieved with Prince ECB is in fact even comparable to using the pipeline without any cipher and for rate conversion only.

Regarding write bandwidth, all modes are capped at around 105 MB/s although the non-tree modes are supposed to have identical read and write performance. As it turns out, the reason for the observed write bandwidth limit is not the encryption pipeline itself, but the sequential way write requests are issued from the CPU cache in our setup. To achieve full write bandwidth, multiple parallel write requests would be needed.

Compared to Prince, the bandwidth results for the round-based AES implementation show the other side of the spectrum for the ECB, CBC, and XTS modes. Note that the use of multiple AES cores in parallel would be possible to increase the bandwidth of the ECB and XTS mode. CBC on the other hand would not benefit from additional cipher hardware at all given its algorithmic dependencies.

Ascon covers the middle ground regarding bandwidth, but additionally provides spoofing and splicing protection. Interestingly, also the replay-protected Ascon TEC-tree performs comparable to the AES modes regarding read bandwidth. The write bandwidth of the tree on the other hand is worse. However,

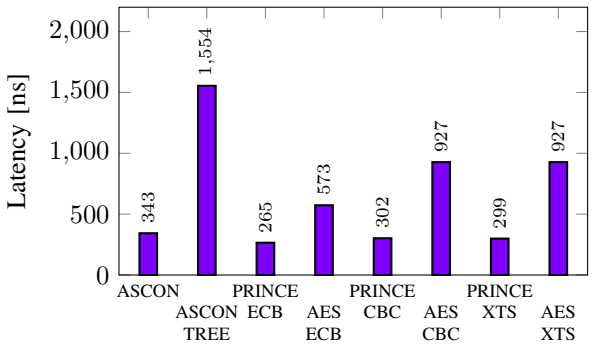

Fig. 8. Memory read latency determined with LMBENCH (`lat_mem_rd 8M`).

even this number is comprehensible considering that, in the evaluated parameterization, writing between 1 and 32 bytes of memory actually requires to read, decrypt, encrypt, and write 360 bytes. Decreasing the size of the protected memory as well as increasing the cache line size of the processor are ways to improve write performance for the tree.

At least for reads, the effect of bigger cache line sizes can be evaluated by enabling the double line fill feature of the cache controller. Due to the bigger requests, read bandwidth is typically increased. With double line fill enabled, Prince-ECB even reaches up to 94 % of the possible bandwidth (i.e., 187 MB/s). However, the correct parameterization of the pipeline is important as shown in Fig. 7. Unfortunately, the double line fill feature can not replace a cache with doubled line size since only read requests are widened. Namely, write requests still have standard size and scale like standard read requests. Operating the encryption pipeline with double line fill enabled and parameters that increase read performance thus typically reduces write performance.

Latency: Compared to cache accesses, RAM accesses are slow and adding transparent RAM encryption further exacerbates this situation. However, as shown in Fig. 8, the actual impact on the latency strongly depends on the used cryptographic primitive. The 265 ns from the Prince ECB implementation again can be considered as an estimate for the latency cost of our memory encryption framework. However,

taking the real memory latency of the hardware (~ 80 ns) into account, the actual overhead of the FPGA design is around 185 ns. At our evaluation frequency (50 MHz), this corresponds to a minimum overhead of merely 9 cycles. The Ascon-based TEC tree on the other hand has the highest latency of all evaluated designs. Yet, it has to be put into perspective that the tree mode has to decrypt much more data (4 tree nodes + 1 data node).

Frequency: The maximum clock frequency is also an important property given that higher frequencies increase bandwidth and decrease latency. While we evaluated all designs at the same frequency of 50 MHz, all of them can be clocked higher. Namely, enabling optimizations in the EDA tool (Vivado) already increases the maximum frequency of the designs to values between and 63 MHz and 75 MHz (upto +50 %). Nevertheless, even then, the critical path is mainly determined by routing delay. This is due to the fact that the used Artix 7 FPGA (speed grade 1) is an entry model. The next stronger Zynq XC7Z030 speed grade 1 model with Kintex 7 FPGA, for example, can already operate the slowest design (Ascon tree) at 93 MHz (= +86 %). Using an XC7Z030 with speed grade 3 even yields a maximum frequency of 126 MHz (= +152 %) for the tree design.

FPGA Utilization: The XC7Z020 features a total of 53,200 lookup tables of which between 8.9 % (Prince ECB) and 19.2 % (Ascon tree) are occupied by our designs. Similarly, between 2.3 % and 4.4 % of the 106,400 available flip flops are used. The use of 36 kB of block RAM is also negligible (4.5 blocks of the available 140) given that they are solely used in the Ascon tree design for the tree roots and as simple nonce cache. Considering that the used FPGA is more or less an entry-level device, more than enough resources remain available for other use cases.

V. Conclusion

In this work, we present an open source framework of modular building blocks to implement RAM encryption solutions. A simple, fully synchronized stream interface is used to connect the individual blocks and permits to easily replace specific components as needed. As the result, realizing arbitrary encryption pipelines is as simple as connecting the needed blocks according to the data flow graph of the design. The evaluation, using various cipher primitives and modes, shows that our framework is very flexible and can easily support differing block sizes and memory alignment constraints. Furthermore, the results demonstrate that retrofitting memory encryption to Zynq SoCs is feasible and that Ascon (with and without tree) is a decent choice for memory encryption, when authenticity is desired in addition to confidentiality.

Acknowledgements

This work has been supported by the Austrian Research Promotion Agency (FFG) under grant number 845579 (MEMSEC). Furthermore, this project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

References

[1] "IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices. IEEE Std 1619-2007," pp. c1–c32, April 2008.

[2] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, and others, "PRINCE–a low-latency block cipher for pervasive computing applications," in *Advances in Cryptology–ASIACRYPT*, 2012, pp. 208–225.

[3] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Ascon v1.2," 2016.

[4] G. Duc and R. Keryell, "Cryptopage: An efficient secure architecture with memory encryption, integrity and information leakage protection," in *Annual Computer Security Applications Conference (ACSAC)*, 2006, pp. 483–492.

[5] R. Elbaz, D. Champagne, C. H. Gebotys, R. B. Lee, N. R. Potlapally, and L. Torres, "Hardware mechanisms for memory authentication: A survey of existing techniques and engines," *Trans. Computational Science*, vol. 4, pp. 1–22, 2009.

[6] R. Elbaz, D. Champagne, R. B. Lee, L. Torres, G. Sassatelli, and P. Guillemin, "Tec-tree: A low-cost, parallelizable tree for efficient defense against memory replay attacks," in *Cryptographic Hardware and Embedded Systems - CHES*, 2007, pp. 289–302.

[7] C. Fruhwirth, "New Methods in Hard Disk Encryption," Tech. Rep., 2005.

[8] S. Gueron, "A memory encryption engine suitable for general purpose processors," *IACR Cryptology ePrint Archive*, vol. 2016, p. 204, 2016.

[9] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," 2016, http://developer.amd.com/resources/articles-whitepapers/.

[10] P. Rogaway, "Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC," in *Advances in Cryptology - ASIACRYPT*, 2004, pp. 16–31.

[11] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and bonsai merkle trees to make secure processors OS- and performance-friendly," in *IEEE/ACM International Symposium on Microarchitecture (MICRO-40)*, 2007, pp. 183–196.

[12] G. E. Suh, C. W. O'Donnell, and S. Devadas, "AEGIS: A single-chip secure processor," *Inf. Sec. Techn. Report*, vol. 10, no. 2, pp. 63–73, 2005.

[13] Xilinx Inc., "XAPP1175: Secure Boot of Zynq-7000 All Programmable SoC," 2015.

[14] ——, "XAPP1239: Using Encryption to Secure a 7 Series FPGA Bitstream," 2015.

[15] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," in *International Symposium on Computer Architecture (ISCA)*, 2006, pp. 179–190.