MemEnc: A Lightweight, Low-Power, and Transparent Memory Encryption Engine for IoT

Naina Gupta, Arpan Jati, and Anupam Chattopadhyay, *Senior Member, IEEE*

***Abstract*—Recent advancement in technologies has led to the widespread adoption and deployment of Internet-of-Things devices. Because of the ubiquitous nature of these devices, they process large amounts of personal and sensitive data. These data are typically stored on DRAM chips, and hence becomes an easy target for attackers. Memory encryption is a commonly adopted solution to provide confidentiality. However, realizing a lightweight, low-latency, low-power solution for resource-constrained devices is a challenge. To address this, we designed MemEnc, a purely hardware-based solution that performs encryption on-the-fly and handles memory requests transparently without any OS intervention. MemEnc runs at a maximum frequency of 401 MHz and requires about 23.2 kGE (gate equivalents) on 65-nm ASIC and consumes only 1.9 mW of power at 250 MHz. Using comprehensive benchmarking, we also analyze the applicability of the proposed solution on real-world workloads. Our experiments show that certain real-time applications can run with full memory encryption and still meet system requirements. Moreover, we integrated our memory encryption engine with ARM TrustZone and present comparative results for the different case studies with Intel SGX. We show that static and dynamic efficiency-security tradeoff is necessary for all scenarios.**

***Index Terms*—Advanced encryption standard (AES), ARM TrustZone, DRAM, encryption, FPGA, Intel SGX, Internet of Things (IoT), memory, security, Zynq.**

I. Introduction

With the increase in usage of smart and intelligent devices nowadays, security becomes even more crucial. These devices store and process large amounts of personal/sensitive data. Even in the presence of secure communications between these devices using SSL/TLS, the "critical data" are typically stored in RAM in the *clear*. As the decision-making process of these devices depends largely on this stored data, they are becoming one of the major attack targets. Many of these devices are being deployed in open environments or are lost easily such as mobile phones, thus enabling physical attacks. To protect sensitive data against unauthorized access, snooping, data remanence, hardware probing, etc., memory encryption is recommended. However, no prior work shows the impact of such security measures on edge applications and real-time systems with strict timing requirements.

The introduction of new hardware-assisted solutions, such as Intel SGX [1], ARM TrustZone [2], AMD SEV [3], etc., to provide security has gained a lot of importance. Both Intel SGX and AMD SEV offer solutions implementing memory encryption to ensure security of the data stored in DRAM. Whereas, ARM TrustZone does not perform encryption of the data stored in the DRAM [4]. The current solution only prevents unauthorized access. Hence, TrustZone is still vulnerable to certain attacks, such as hardware probing, cold boot attacks [5], etc. ARM is a widely deployed platform, especially in the medium to low end edge devices. To secure these devices with memory encryption, Zhang *et al.* [6] presented a software-based memory encryption approach with TrustZone. This approach is suitable for many applications as hardware changes are not required. But, task-based encryption has significant overhead for a large number of tasks. Furthermore, the rest of the DRAM including heap is not protected natively. In order to protect all this data using software, the overhead becomes very high. Hence, in our work, we focus on hardware-based transparent memory encryption in combination with TrustZone. This combination provides significant performance benefits and improved security without any changes to the software.

Werner *et al.* [7] in 2017 presented an open-source implementation for transparent memory encryption supporting multiple ciphers and modes of operation. Their architecture supports *burst modes* and pipelined ciphers resulting in high performance. Even though the architecture is modular and provides many implementation options, it is complex and utilizes a large amount of area. Moreover, the architecture has a lot of latency because of the long pipelines. This might become a limiting factor for certain applications running on resource constrained platforms. As our work focuses on resource constrained systems, we implemented a custom lightweight and low-power memory encryption engine (MEE). To compare the performance of both the architectures, we present results for four different case studies targeting different aspects of real-world workloads. We also discuss several tradeoffs in terms of latency, throughput, response time, power consumption etc., to provide valuable insights for real-world applications.

The contributions of this article are as follows.

1) We present design and experimental results for a lightweight, low-latency, and low-power transparent MEE. We report results for both FPGA and ASIC.

Manuscript received July 21, 2020; revised October 15, 2020; accepted November 12, 2020. Date of publication November 26, 2020; date of current version April 23, 2021. This work was supported in part by the Singapore National Research Foundation (SOCure—http://www.green-ic.org/socure) under Grant NRF2018NCR-NCR002-0001, and in part by the BMW Asia Pte. Ltd. *(Corresponding author: Naina Gupta.)*

Naina Gupta and Anupam Chattopadhyay are with the School of Computer Science and Engineering, Nanyang Technological University, Singapore (e-mail: naina003@e.ntu.edu.sg).

Arpan Jati is with the School of Physical and Mathematical Sciences, Nanyang Technological University, Singapore.

Digital Object Identifier 10.1109/JIOT.2020.3040846

2) To the best of our knowledge, this work presents the first in-depth study for performance overhead due to memory encryption on real-time systems and edge computing applications.
3) We compare our results with the work in [7] and show that a MEE using *burst mode* operating at a lower frequency versus the one without *burst mode* operating at about thrice the frequency and utilizing thrice less resources can have similar or better performance in certain scenarios.
4) We also present results for TrustZone combined with hardware-based transparent MEE. To the best of our knowledge, this work presents first such performance figures. Furthermore, we compare the performance with Intel SGX using the same case studies.

Outline: The remainder of this article is organized as follows. Section II starts with the discussion of the design challenges and optimizations used. Section II-B presents the security considerations and how we addressed them. Finally, in Section II-C, we present the design architecture of MemEnc. Section IV describes the experimental setup and results. We first compare different implementations in terms of bandwidth in Section IV-A, latency in Section IV-B, and area and power in Section IV-C. Then, we discuss and present results for four different case studies in Section IV-D. We also present and compare results with Intel SGX in Sections IV-E and IV-F. We finally conclude this article with a discussion in Section V.

II. Lightweight Memory Encryption Engine

In this section, we discuss challenges faced during the design process, the security design considerations, and how we addressed them followed by the design architecture.

A. Design Challenges and Optimizations

1) Encryption Mode and Block Size: Most encryption schemes work in chunks of data known as blocks. A memory read or write request can be aligned or misaligned as per the cipher block size. For instance, AES (advanced encryption standard) [8] has a block size of 16 bytes. Assuming the data is stored in RAM sequentially starting from address 0. A two byte write request for address location 15 is a misaligned request and thus, overlaps two blocks. In order to service this request, one needs to fetch the blocks 0 and 1 (32 bytes of data), perform decryption, update the data with the two new bytes, and write back after encryption as done in [7]. This leads to significant overhead for small data updates, which is usually the case for Internet-of-Things devices. In order to improve performance, we used AES in counter mode, with the address used as the plaintext and using only 4 bytes of the output. Even though it wastes the remaining encrypted 12 bytes, for every write operation., this approach saves extra read and write operations.

2) Encryption Execution Time: In a typical scenario, memory requests through AXI bus are serviced immediately. But, in order to support memory encryption transparently, one needs to halt AXI transactions for the cipher execution to complete. For instance, during a read transaction, the fetch request is forwarded to the memory subsystem right away and we have to wait for the encrypted data from the memory before the decryption can start. These data are then processed to obtain the plaintext and only then the transaction can complete. Both read and write transactions require careful consideration of the various AXI control signals and their dependencies. In our design, we implemented multiple state machines for each of the five channels on both master and slave interfaces to handle these AXI transactions while targeting low latency.

3) Resource Utilization: Resource utilization is primarily attributed to control logic, data pipelining, and cipher blocks. In order to have maximum throughput, a design should support: 1) AXI *burst mode* with *burst length* of atleast 16 and 2) pipelined cipher blocks and datapath as done in [7]. This requires large amount of resources. As per our experiments, even without a pipelined cipher, implementing *burst mode* requires more than 4000 LUTs (lookup tables). As we aim to utilize less resources while maintaining good performance, we used a non-pipelined implementation of the AES cipher with *burst length* set to one. Typically, a misaligned request requires two invocations of the cipher (for current and next address). A naive implementation may utilize two AES instances to encrypt both the addresses simultaneously to save some clock cycles. To further optimize the design and reduce the overall area, we did some analysis (discussed later in Section V and Table III). Based on this analysis, we observed the following.

1) Utilizing two AES instances to encrypt both the addresses simultaneously has minimal performance benefit while requiring significant amount of area. Hence, one can encrypt both addresses in a sequential manner; requiring one AES instance each for read and write operations.
2) Furthermore, there is zero overlap between a read and a write request for both cipher execution as well as overall execution. Thus, the read and write requests can be handled by a single AES instance with multiplexing.

B. Security Design Considerations and Evaluation

In this section, we present the security considerations focused in this work. For this, we consider the following attack scenarios and discuss how we addressed them.

1) *External DRAM Access:* In this, the attacker tries to access sensitive data stored on DRAM using snooping, cold-boot attacks, etc. In a cold boot attack [9]–[12], data remanence effect of the memory is exploited to extract sensitive information stored in the memory. Whereas in the case of bus monitoring attacks, an attacker tries to probe on the critical buses carrying data between various components [13] to read the data in transit, or even tamper the data. As all the sensitive data that goes off-chip are encrypted by the implemented transparent peripheral, any attempt to read the contents of DRAM results in garbage data; thus, ensuring confidentiality.
2) *Key Reuse Attack:* It is known in the literature that using the same {Key, nonce} pair can leak information about the plaintext [14]. In order to prevent this attack, we are using a 32-bit nonce for every write operation. The basic idea is shown in Fig. 1. For each 32-bit data block,

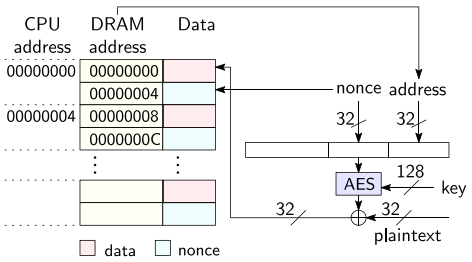

Fig. 1. MemEnc: Encryption process.

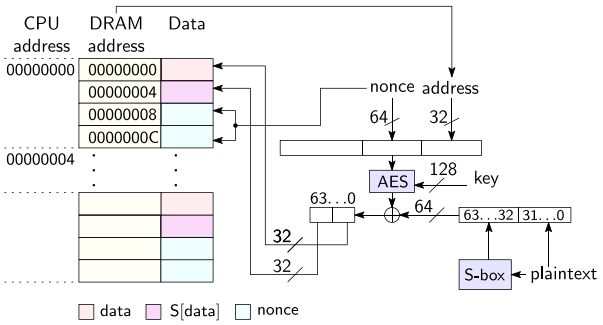

Fig. 2. Alternate approach for encryption (with 64-bit nonce and fault protection). The 64-bit plaintext is composed of 32-bit data and 32-bit S[data].

the data and its corresponding nonce are stored in the DRAM. The DRAM address is used as counter along-with the nonce as input to the AES module. The output of the AES module is then XORed with the data to generate the final ciphertext. Since, the nonce is generated randomly for each request, an attacker in this case requires about 2^{32} requests to the same address (counter) to have a different ciphertext with the same {Key, nonce} pair. This solution requires twice the normal amount of DRAM. As shown in Fig. 2, one can also use 64-bit nonce to increase the attack complexity to 2^{64} requests. Furthermore, to prevent fault attacks, the S[data] XORed with the AES output is stored, where S is the AES SBox applied bytewise. In order to successfully mount an attack, one needs to precisely inject a fault at both the data and the corresponding bit locations in S[data]. For such an approach, one needs to write data, S[data], and 64-bit nonce for a 32-bit data block requiring four times the DRAM than the unprotected implementation.

Discussion: As discussed above in the design with 32-bit nonce, an attacker needs to perform 2^{32} write operations at the same location, requiring 16 GB of writes before the {Key, nonce} pair is repeated. This takes about half an hour at about 10 MB/s. In a real application, the same location is not accessed continuously, so the possibility of collision is further reduced significantly. As even a collision does not guarantee a real attack, the system provides adequate security for many applications. If the application requires writing to the same location continuously, we suggest 1) using internal SRAM for such memory accesses or 2) using the alternative approach with 64-bit nonce as shown in Fig. 2. Compared to the 32-bit nonce scheme, the overhead is just around 100–200 LUTs, with minimal degradation in performance. In this case, it will take about 2.23×10^5 years to get a collision. Considering the very low cost of DRAM, both the solutions are practical for deployment.

3) *Unauthorized Access:* The increase in design complexity of embedded systems has increased the usage of third-party applications/libraries. An attacker can exploit a vulnerability in any of these to gain access to the whole system. In order to avoid such access between cross-applications, we are using TrustZone with the encryption engine. As per the TrustZone threat model, memory access is restricted for applications outside the trusted execution environment. As the peripherals are memory-mapped, hardware access is also restricted. Therefore, in combination with points above, many software and common hardware attacks become infeasible; enhancing the security of the overall system to a great extent.

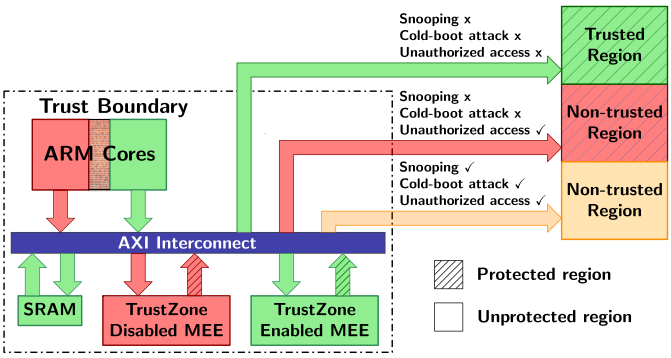

Fig. 3. MemEnc with TrustZone (access control).

4) *Side-Channel Leakage Resistance:* It is well known in the literature that using side-channel attacks [15], [16] (SPA, DPA, etc.), one can successfully recover the secret key used for encryption. In order to protect the algorithms from side-channel leakage, several countermeasures, such as masking [17], [18], threshold implementations [19]–[22], etc., are used. But, such countermeasures are usually expensive in terms of area and performance. Hence, it would defeat the overall purpose of a lightweight memory encryption engine. Another approach is to use fresh rekeying technique as shown in [23]. In order to successfully mount a DPA attack, one needs to find the correlation between a known value and a guessed value. In our design, we are using upto 64-bits of random nonce for every encryption to initialize the AES engine. Furthermore, we only use a few bytes of the ciphertext. These design choices make standard DPA-based key recovery attacks on round-based implementations impossible.

The high-level overview of the interactions between the secure/non-secure world with the encryption engine is shown in Fig. 3. The figure also shows how different regions in the DRAM are protected from different attacks.

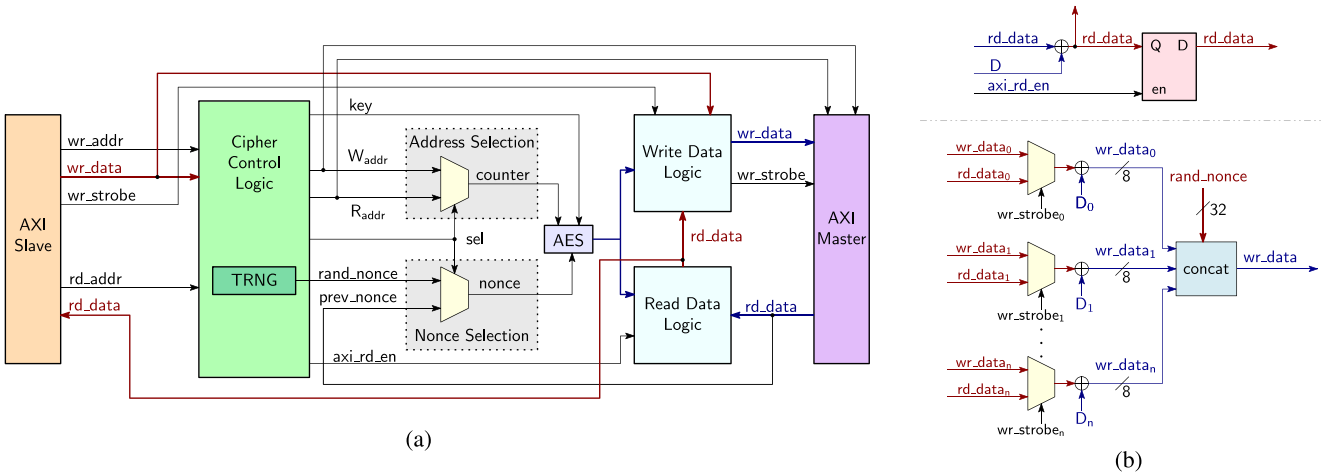

Fig. 4. MemEnc architecture. (a) Memory encryption engine. (b) Read (top) and write (bottom) data logic.

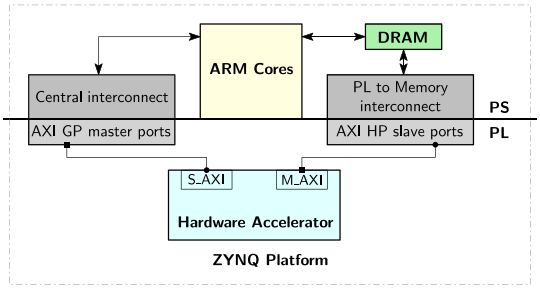

Fig. 5. Architecture for memory encryption accelerator.

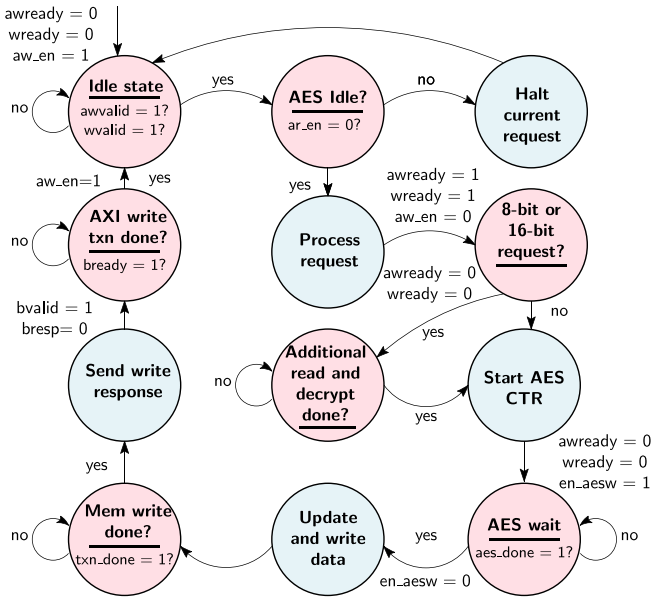

Fig. 6. Write transaction state machine.

C. Design Architecture

In this work, we designed and implemented a lightweight hardware accelerator to handle all the memory requests transparently. For the implementation, we used Xilinx Zynq-based platform, which combines ARM cores with reconfigurable logic. The hardware accelerator is connected to the ARM core using AXI-buses. When any task requests a write operation to the memory, the request is send to the hardware accelerator module using the AXI slave port. This, in turn, automatically calls the encryption module to encrypt the received data. The write/read to/from the memory is done using a custom peripheral having two ports. An AXI slave connected to the GP0 port and an AXI Master connected to the high performance (HP0) port of the Zynq PS. The benefit of this implementation over a software-based approach is that the reads/writes are completely transparent and no changes are required to the software. Furthermore, the OS can run other tasks in parallel as interrupts are not needed. The peripheral appears as an *encrypted memory space*, which in turn maps to the DRAM. So, the data transferred to and from the peripheral would be using the RAM, but all the data would be encrypted. Fig. 5 shows the basic architecture for the custom peripheral.

The design for our MEE is shown in Fig. 4(a). The cipher control logic block is used to map the CPU address (wr_addr and rd_addr) to the corresponding DRAM address (W_{addr}) and to generate keys and random nonce for the cipher. For key generation, a TRNG (true random number generator) can be optionally used. As mentioned above, the signals are multiplexed to the single AES instance for both read and write requests. The ciphertext output of the AES block is passed to the read and write data logic blocks depending on the type of request. The output from these blocks is used to complete the read/write request, respectively. Apart from this, in order to handle 8-bit or 16-bit data block write operation, a read from the memory and decryption of that block is required prior to the encrypt-then-write operation. This is because we are using AES in the counter mode with nonce; hence, we need to update the complete 32-bit data block with the randomly generated fresh nonce. For this purpose, the decrypted data (rd_data) is also shared with the write data logic unit and a control signal (axi_rd_en) is used to decide whether to latch the decrypted data on the AXI bus or not.

Fig. 4(b) shows logic for the read and write data blocks. For read operation, output of the AES block (let us say D) is XORed with the data read from memory (rd_data). It is latched depending on the value of axi_rd_en signal. As

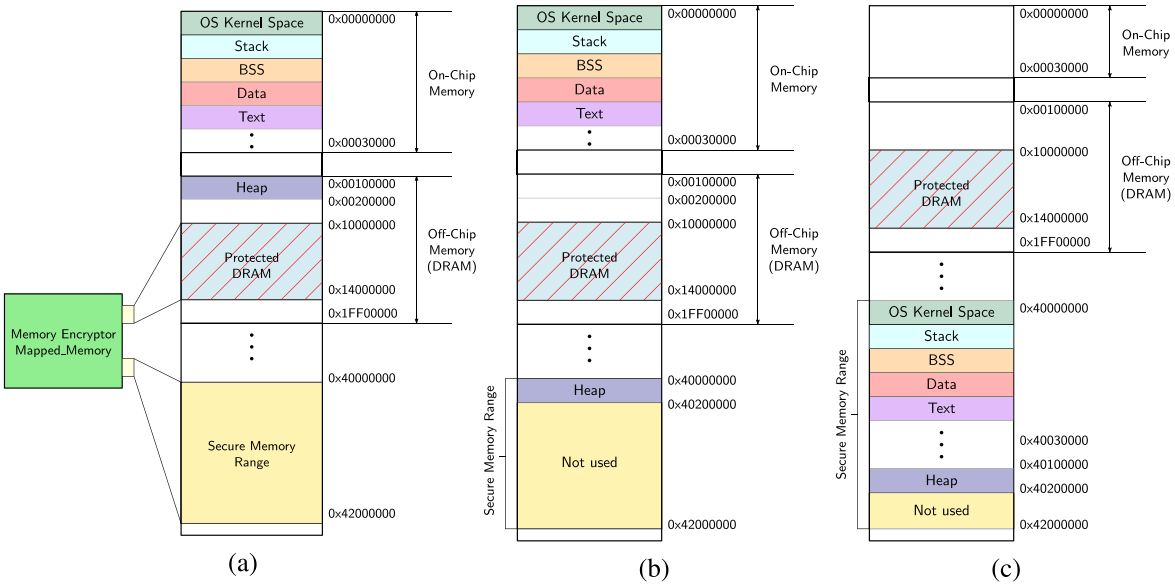

Fig. 7. Memory map. (a) Unprotected (protected DRAM is not being used). (b) PME. (c) FME.

mentioned above, a write operation may additionally require decrypted data, hence, the input for a write operation is selected depending on the value of the write strobe signal (wr_strobe). For each byte, if wr_strobe = 1, then data received using AXI bus (wr_data) is used, otherwise, (rd_data) is used as plaintext. This is then XORed with the corresponding AES output byte and concatenated with a fresh nonce before writing to the memory.

Fig. 6 shows high-level interactions of the AXI signals with the encryption module for a write request. As can be seen from the figure, a write request is accepted only when the AES module is idle. This is ensured using the write enable aw_en and read enable ar_en signals. The two signals are always high in the default state and depend on each other for respective request execution. For instance, when a write request is being processed, the ar_en should be in high state. A low state indicates the AES module is busy, hence cannot process the current request. As mentioned previously, a 8-bit or 16-bit request requires an additional read and decrypt process to update the encrypted data with the fresh nonce. The states 8-bit or 16-bit request and Additional read and decrypt done are used to handle this requirement. The state AES wait checks for the aes_done signal. Once encryption is completed, the data is written to the memory and AXI write transaction is completed after sending and receiving the write response. A similar state machine is required for read transactions as well.

III. Memory Map

The memory map for the various implementations is shown in Fig. 7(a). From the software perspective, any data written to memory location 0x40000000 upto 0x42000000 is encrypted and stored in the DRAM starting from memory location 0x10000000 onward, transparently. The memory ranges can be adjusted as per application requirements. There are two possibilities for memory encryption using this approach.

1) *Partial Memory Encryption (PME):* As shown in Fig. 7(b), a partial region of the memory can be encrypted. In our experiments, only the Heap section is mapped to the secure memory range and was encrypted, the rest of the memory space was left unprotected. To increase the overall performance of the system, one can map critical sections (storing sensitive data) to the encrypted memory region. The linker script can be modified to map which sections to encrypt and which sections to leave unprotected.
2) *Full Memory Encryption (FME):* In this case, all sections of the application were mapped to the secure memory range and hence, are encrypted as shown in Fig. 7(c).

IV. Implementation Results

In this section, we present different case studies and compare our performance with the work in [7]. As their implementation does not support AES-CTR, we used the supported AES-ECB with *burst length* = 16 for our evaluation. This comparison was primarily done to showcase the performance differences as a result of varying *burst lengths*. We refer to our implementation as Burst1 and the work in [7] as Burst16 in the rest of this article. The maximum achievable frequency for Burst1 was 175 MHz, while the frequency for Burst16 was 56 MHz. This difference is because in the latter there are about 29 levels of logic compared to 7 in the former. All the experiments were performed using 175 and 50 MHz for the designs, respectively.

The HDL for Burst1 is written in Verilog whereas the HDL design for Burst16 is in VHDL. *Xilinx Vivado* [24] version 2020.1 was used for functional testing and experiments while *Synopsys Design Compiler* version J-2016 was used for synthesis of both the designs. We used TSMC 65-nm Low Power Standard Cell Library (TCBN65LP) for the ASIC implementation. For the experiments, we used Digilent MiniZed as the target platform. The board is built around a Xilinx Zynq XC7Z007 FPGA with an embedded single core ARM Cortex-A9 processor running at 667 MHz and has a Micron DDR3

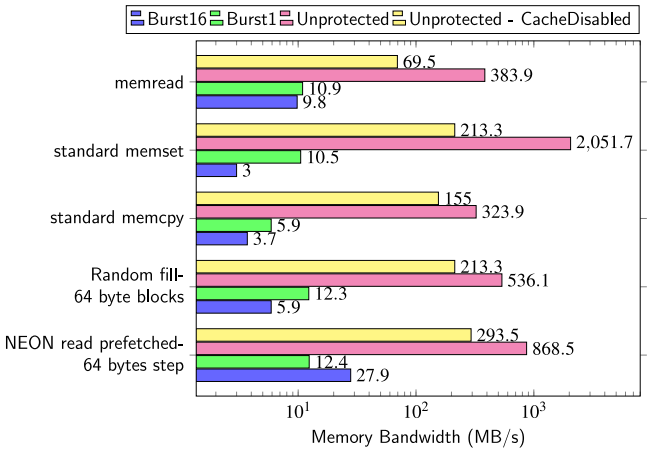

Fig. 8. Memory bandwidth.

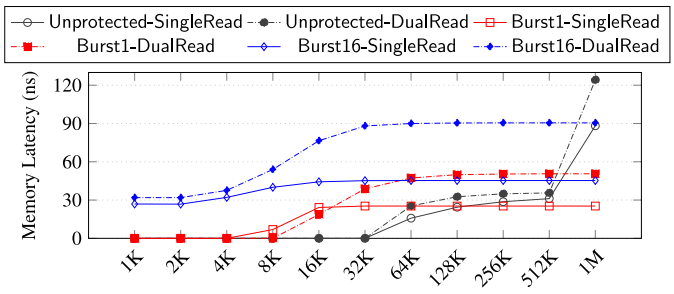

Fig. 9. Memory latency for single and dual random read.

TABLE I
RESOURCE UTILIZATION AND POWER CONSUMPTION FOR BURST1 AND BURST16

	FPGA			ASIC		
	LUTs	Registers	Max. Freq. (MHz)	Total GE (kGE)	Power (mW)	Max. Freq. (MHz)
AES128	1282	276	-	17.2	0.86	-
Burst1	1648	885	175	23.2	1.95	401
Burst16	4701	2332	56	69.2	6.42	223

RAM with 16-bit data bus working at 533 MHz. Any other similar platform can also be used.

For the protected implementations on Zynq, one needs to use CPU programmed I/O instead of the high performance cached DMA memory interface. The theoretical achievable bandwidth using such an interface in the Zynq architecture is less than 25 MB/s [25]. Because of such limitation in the Zynq architecture, a drastic degradation in interface performance is expected for any peripheral even without encryption. This is reflected in the results as well.

A. Memory Bandwidth

Here, we present the comparison results for maximum bandwidth achievable using unprotected implementation versus protected implementation. For all the tests except memread, we used TinyMemBench [26] for benchmarking.

Fig. 8 shows the achieved memory bandwidth for the presented designs. As can be seen in the figure, we are able to achieve a bandwidth of 10.9, 10.5, and 5.9 MB/s for a read, write, and memcpy operation, respectively, for Burst1. This is about 42% of the maximum achievable bandwidth for read and write requests. Whereas for Burst16, it is about 39.2% and 12%, respectively. Standard memread and standard memcpy are the "C" library functions provided in the Xilinx SDK. Furthermore, the performance for random memory access is similar to the sequential access for Burst1. This indicates that the implementation does not add additional overhead for random memory accesses patterns. Interestingly, for Burst16, random write is faster than most of the other operations. Moreover, the performance figures for Burst1 are better than Burst16 in all cases except NEON read prefetched. The performance difference between Burst1 and Burst16 can primarily be attributed to the differences in working clock frequency, complexity in design, and different latency figures.

B. Memory Latency

We used TinyMemBench for reporting additional memory latency with varying block sizes as shown in Fig. 9. For Burst1, the memory latency due to encryption is mostly either similar or smaller than the corresponding unprotected implementation. This is because the encryption module handles a single request at a time. Thus, the memory latency remains more or less constant and does not vary much with an increase in block size. Whereas, in the case of the unprotected implementation, latency increases when reading block size larger than 32 KB. It is almost 3.5× compared to the Burst1 implementation for reading 1 MB data. For Burst16, there is a significant latency overhead for even very small block sizes as a result of the added burst support, multiple buffers, and pipelines.

C. Resource Utilization and Power Requirements

Table I shows results for both FPGA and ASIC. In the case of FPGA, the resource utilization is reported in terms of LUTs and Slice Registers. Both the implementations do not require DSPs or BRAMs. Whereas in the case of ASIC, the area is reported in terms of gate equivalents (GEs). A single instance of AES128 requires 1282 LUTs and 276 slice registers on an XC7Z007S FPGA and around 17.2 kGE on an ASIC. The table also presents resource utilization for the complete memory encryption peripheral (including the AES instance). As discussed in Section II, we are using one instance of AES multiplexed for read and write channels. Hence, the overall resource utilization consists of the area required for one AES instance and the control logic in the case of Burst1.

As shown in the table, Burst16 requires 2.85× LUTs and 2.64× slice registers compared to Burst1. When comparing ASIC results, Burst16 requires almost 3× GEs compared to Burst1. This is mainly because about 50% of the resource utilization is required for the control logic and the rest 50% is for the two AES instances in the case of Burst16, whereas in Burst1, the resource utilization for control logic is only around 24.4% of the total area. The table also reports power consumption for both the designs. To present a fair comparison, the designs were run at 250 MHz. One can see that power consumption is also high (almost 3.3× compared to Burst1) in the

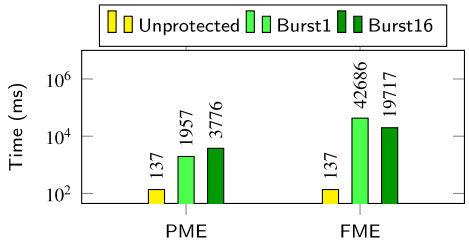

Fig. 10. Pi calculation (1000 digits).

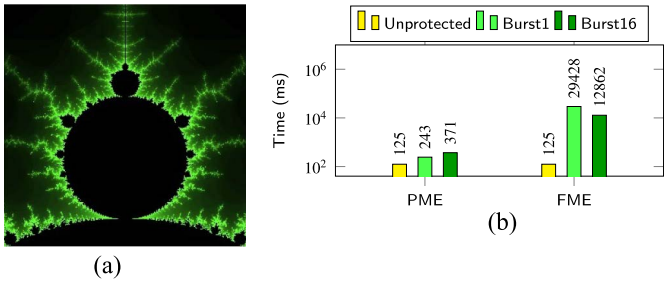

Fig. 11. Mandelbrot set (150 × 150) with *scale factor* = 0.4 and *maximum iteration* = 256. (a) Rendered fractal. (b) Performance results.

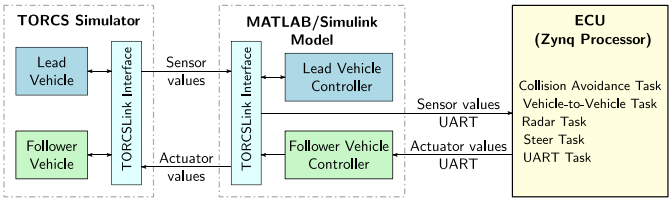

Fig. 12. Testbed/setup.

case of Burst16. This difference is a result of a significantly complex control logic and buffering required for Burst16 as the cipher block requirements are similar.

D. Case Studies

In this work, we analyze different aspects of using memory encryption using four different scenarios. The details are discussed as follows.

1) How the Performance Is Affected for an Application Under Very High Computational Load?: Almost every application experiences sudden intermittent high loads. Thus, it is important to understand how memory encryption affects the performance in such a scenario. In order to understand this, we present the results for Pi digits calculation—a common benchmarking and stress-testing application. A lot of work has been done in the past few years to develop algorithms to calculate the correct value of Pi to a large number of digits [27], [28].

For our analysis, we used the Spigot algorithm [29] for calculating upto approximately 1000 digits. The execution time for the same is depicted in Fig. 10. Considering Burst1, the performance overhead for Pi calculation is around 14.3× and 311.6× in the case of PME and FME, respectively. Whereas, for Burst16, the overhead is around 27.56× and 143.9×, respectively. The differences in performance between Burst1 and Burst16 are noteworthy. In the case of PME, Burst1 performs 1.93× faster than Burst16 whereas, for FME, Burst16 performs 2.16× better than Burst1. We believe this is due to the fact that the complete memory region is encrypted for FME and Burst16 supports bursts, thus enabling large chunks of data to be transferred quickly in single requests.

2) What Is the Performance Overhead for Applications Requiring Fast Floating Point Calculations?: Many critical applications involving image/video processing, sensor data analytics, industrial control systems, etc., require fast *floating point* calculations. Hence, it would be interesting to study the performance of memory encryption in such a compute intensive scenario. Mandelbrot set [30] is a very famous fractal and a common benchmarking application for the FPU. It is generated by iteratively applying the complex function $f_c(z) = z^2 + c$. Fig. 11(a) shows the generated image from our implementation.

Fig. 11(b) shows the time required to generate an image for the Mandelbrot set of dimension 150×150. Considering the case of FME, it requires about 29.4 s using Burst1 and 12.8 s using Burst16 which is about 235.4× and 102.89× slower compared to the unprotected implementation. In case of PME, performance overhead due to memory encryption is very small (just about 1.9× using Burst1 and 2.9× using Burst16) compared to the unprotected implementation. These results demonstrate that using memory encryption for only those regions which store critical data does not add too much performance overhead while providing overall security of the data. Furthermore, even though FME degrades performance, it can be useful in scenarios where security is of utmost importance and performance is not a major factor.

3) What Is the Performance Overhead for Time-Critical Real-Time Systems?: In order to study real-time performance, we used hardware-in-the-loop simulation of the adaptive cruise control [31] application based on the open-source racing game TORCS [32]. A *MATLAB Simulink* [33] model acts as an interface between the TORCS and the ECU (running on the Zynq board). All the communication between ECU and the *simulink* model is done using UART. The testbed setup is as shown in Fig. 12.

In the implemented testbed, the lead vehicle is moving on a racetrack at speeds of about 80–90 km/h, while braking intermittently. The sensor values for the lead vehicle are passed on to the ECU (for the follower). The ECU computes *throttle*, *brake*, and *turn* actuator values based on the current sensor values for both the vehicles. We used FreeRTOS [34] for creating the different tasks to compute these values [35]. These tasks were running at different update rates as required. We used a tick-rate of 1000 for the system and UART update packets were sent every 40 ms.

The results presented in this case corresponds to FME only. Fig. 13 shows plots corresponding to the UART packets received, processed, and being sent. One can clearly see that the number of packets sent and received per second is quite similar to when there is no encryption (unprotected). One can also notice the increase in number of packets transferred as a response to braking (four events).

Fig. 14 shows average distance between leader and follower vehicles and the average lateral error (distance from the center of the track) for the follower vehicle. The average distance

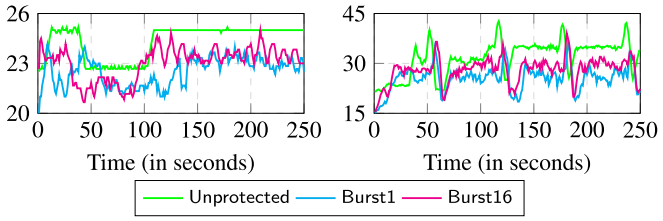

Fig. 13. Packets received per second (left) and packets sent per second (right).

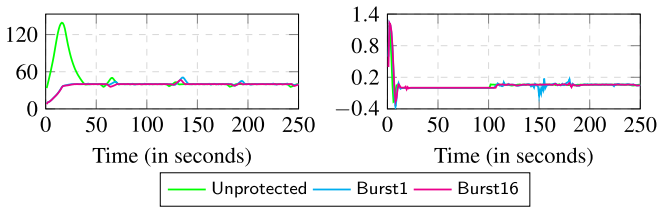

Fig. 14. Distance between the leader and follower (left) and average lateral error of the follower vehicle (right).

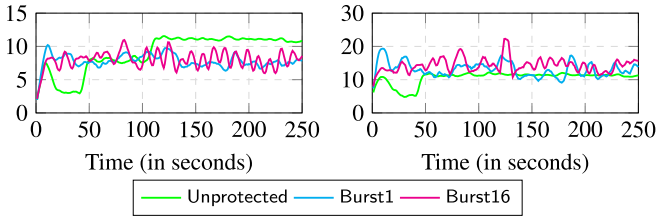

Fig. 15. Mean (left) and standard deviation (right) of the RTT (ms).

remains constant throughout and the average lateral error is also negligible. This illustrates that even though the complete memory region including actuator variables were protected using real-time memory encryption, one vehicle (follower) was able to follow another vehicle (Leader) without going offtrack. The small peaks in the distance between the two vehicles correspond to braking events of the leader. Although the response (throttle) of the follower in both protected and unprotected cases is different, they both take similar time to settle down.

Fig. 15 shows the average round-trip time (RTT) and standard deviation statistics over the task execution time. In the case of an unprotected implementation, the mean RTT is almost constant whereas using memory encryption, this time constantly changes depending on packet losses. If a packet is lost, the round-trip increases. This increase in RTT further increases the standard deviation as shown in the figure.

For applications requiring even faster response, `PME` can also be used. Furthermore, all the plots show that both Burst1 and Burst16 have similar performance. Hence, one can use Burst1 for resource-constrained devices while still meeting strict-timing requirements.

4) How the Performance Is Affected for Edge Computing Applications?: Edge computing [36], [37] is a new paradigm that targets toward shifting the computation and data analytics closer to the location where data is generated rather than at the server. This helps in applications that require faster response times as well as providing data privacy. Hence, it is crucial

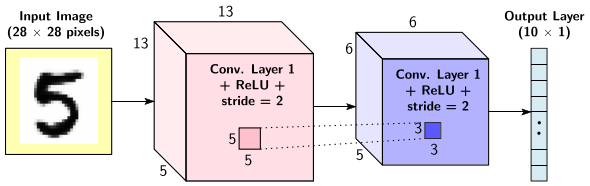

Fig. 16. 4-layer DNN.

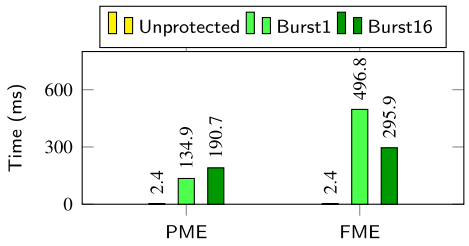

Fig. 17. 4-layer DNN inference.

TABLE II
EXECUTION TIME (IN μS)

Case Study	Intel SGX (disabled)	Intel SGX (enabled)
Pi Test	14923	19350
Mandelbrot Set	14140	19321
DNN Inference	222	365

to determine the impact of memory encryption on edge computing applications. Neural network inference is a good target application to understand this impact. We implemented a 4-layer DNN with one input layer, two hidden layers, and one output layer as shown in Fig. 16. The model is customized for MNIST hand-written digit recognition [38].

Fig. 17 shows the execution time required for recognizing a single digit using a trained model. As the trained model is loaded into DRAM from the SD Card, and MiniZed lacks the required slot, this case study was performed using the ZedBoard. As shown in the figure, for Burst1, it requires about 0.13 s and 0.49 s in the case of `PME` and `FME` for a single inference. Whereas, it is about 0.19 s and 0.29 s for Burst16.

This indicates that even though memory encryption leads to performance degradation (50–80× for `PME` and 120 − 200 × for `FME`), it can still be used for critical applications requiring a response time of a few hundred milliseconds. Furthermore, the results presented are specific to the implemented network, reducing the network size may be acceptable in some applications requiring better performance.

E. Intel SGX Results

We also perform the same benchmarking tests using the Intel Platform with SGX enabled and disabled. For our experiments, we used Intel Core i5-7200U processor running at the base frequency of 2.50 GHz. Table II shows the execution time for implementation with SGX enabled and disabled for different test cases. As shown in the table, the performance overhead due to SGX is about 29.6%, 36.6%, and 64.4% for Pi Test, Mandelbrot Set, and DNN Inference, respectively.

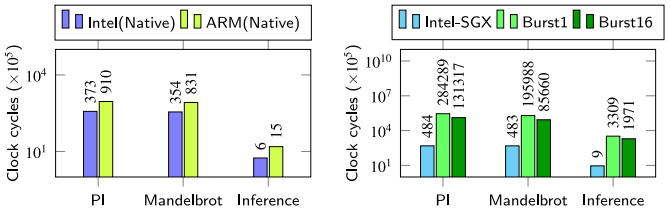

Fig. 18. Intel SGX versus ARM TrustZone results—unprotected (left) and protected (right).

TABLE III
CLOCK STUDY FOR MANDELBROT TEST

Operation	Clock Cycles
read_busy	3128668
write_busy	7767530
execution_overlap	0
read_cipher_busy	1734150
write_cipher_busy	3028476
cipher_execution_overlap	0

F. Intel SGX Versus ARM TrustZone

Fig. 18 shows the performance comparison between Intel and ARM for both unprotected (native) and protected implementations. In order to present a fair comparison, we show the execution time in terms of clock cycles for both the platforms. Furthermore, since Intel SGX protects both the code and data of the enclave, we compare TrustZone with FME only. As can be seen from the figure, the difference in performance for native implementations is only about 2–3×. Whereas, this difference is quite large for protected implementations. The execution times in the case of TrustZone using Burst1 is almost 587.37×, 405.77×, and 367.66× slower compared to SGX for Pi, Mandelbrot, and DNN Inference, respectively. Using Burst16, it is about 271.31×, 177.35×, and 219×, respectively. We believe the major factor for this difference is the maximum achievable bandwidth being limited by the Zynq architecture. Also, the memory width and the corresponding bandwidth is much smaller compared to SGX as there is no cache in the case of Burst1 and Burst16, which significantly reduces the performance. Furthermore, as Burst1 is handling single read/write requests at a time to minimize area compared to SGX handling multiple requests at a time, the performance difference is quite high.

V. DISCUSSION AND CONCLUSION

Apart from AES128 we also performed all the experiments utilizing AES256 as well for Burst1. The AES256 module required 1.38× the LUTs and 1.74× the slice registers compared to AES128. We observed that there is negligible performance overhead while using AES256 even though it has four extra rounds. In order to understand this, we performed a separate analysis by counting the number of clock cycles spent during the AXI transactions.

In Table III, the signals read_busy and write_busy show the total number of clock cycles spent during read and write transactions, whereas the signals read_cipher_busy and write_cipher_busy show the same just for the block cipher invocations. From the values, one can easily see that around 55% of the read and 38% of write duration is spent by the cipher blocks. A significant amount of time is spent waiting on the memory interface to respond. This means that even if the cipher requires slightly more number of clock cycles, the overall performance is not affected in any major way.

TABLE IV
SUMMARY FOR THE CASE STUDIES. BURST1 FME AND PME ARE DENOTED BY ▲ AND △. BURST16 FME AND PME ARE DENOTED BY ● AND ○

Application Requirements	Compute Intensive	Fast FPU	Edge Inference	Real-time Systems
Fast Response	△	△	△ ○	△ ○
Low Area	▲ △	▲ △	▲ △	▲ △
High Throughput	△ ●	△ ●	△ ●	△ ▲ ● ○
Very Tight Latency	△	△	△	△
Low Power	▲ △	▲ △	▲ △	▲ △

Table IV summarizes the multiple case studies presented in the previous sections. Our evaluation demonstrates that different implementations and strategies are suitable for different applications. In real-world deployments, there are many requirements based on application. For instance, critical applications, such as in healthcare, autonomous systems fast and timely response is a key requirement. Similarly, high speed dynamic control systems, such as robotics and automated manufacturing systems have very tight latency requirements. Because of the large number of variations, all these requirements may overlap. These conclusions are specific to the presented case studies and may vary for different applications or requirements.

To conclude, in this work, we designed a lightweight memory encryption scheme. Even though the results are presented for ARM, the engine is portable to any embedded processor, such as RISCV [39], MIPS [40], etc. The implemented design operates at 175 MHz and requires 1648 LUTs and 885 registers on an FPGA. Whereas on an ASIC, it can run at 401 MHz while requiring 23.2 kGE and is very low power (about 1.9 mW at 250 MHz). We also compare our results with the work in [7] and show that our MEE can perform better in some scenarios. Furthermore, using a comprehensive benchmarking, we demonstrated that even though memory encryption has overheads, it is suitable for real-time systems with strict timing requirements as well as edge computing applications. Moreover, for certain applications, partial encryption of critical data can be performed with a very small overhead.

REFERENCES

[1] F. McKeen *et al.*, "Innovative instructions and software model for isolated execution," in *Proc. 2nd Workshop Hardw. Architect. Security Privacy (HASP)*, Tel-Aviv, Israel, Jun. 2013, p. 10.
[2] P. Varanasi and G. Heiser, "Hardware-supported virtualization on ARM," in *Proc. ACM Asia–Pac. Workshop Syst. (APSys)*, Shanghai, China, Jul. 2011, p. 11.
[3] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption, white paper," 2016.
[4] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "CaSE: Cache-assisted secure execution on ARM processors," in *Proc. IEEE Symp. Security Privacy (SP)*, San Jose, CA, USA, May 2016, pp. 72–90.
[5] M. Gruhn and T. Müller, "On the practicability of cold boot attacks," in *Proc. IEEE Int. Conf. Availability Rel. Security (ARES)*, Sep. 2013, pp. 390–397.

[6] M. Zhang, Q. Zhang, S. Zhao, Z. Shi, and Y. Guan, "SoftME: A software-based memory protection approach for TEE system to resist physical attacks," *Security Commun. Netw.*, vol. 2019, pp. 1–12, Mar. 2019.
[7] M. Werner, T. Unterluggauer, R. Schilling, D. Schaffenrath, and S. Mangard, "Transparent memory encryption and authentication," in *Proc. IEEE 27th Int. Conf. Field Program. Logic Appl. (FPL)*, 2017, pp. 1–6.
[8] J. Daemen and V. Rijmen, *The Design of Rijndael—The Advanced Encryption Standard (AES)* (Information Security and Cryptography), 2nd ed. Cham, Switzerland: Springer, 2020.
[9] S. Lindenlauf, H. Höfken, and M. Schuba, "Cold boot attacks on DDR2 and DDR3 SDRAM," in *Proc. IEEE 10th Int. Conf. Availability Rel. Security*, 2015, pp. 287–292.
[10] S. F. Yitbarek, M. T. Aga, R. Das, and T. M. Austin, "Cold boot attacks are still hot: Security analysis of memory scramblers in modern processors," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2017, pp. 313–324.
[11] T. Müller and M. Spreitzenbarth, "FROST—Forensic recovery of scrambled telephones," in *Proc. Int. Conf. Appl. Cryptography Netw. Security*, vol. 7954, 2013, pp. 373–388.
[12] J. A. Halderman *et al.*, "Lest we remember: Cold-boot attacks on encryption keys," *Commun. ACM*, vol. 52, no. 5, pp. 91–98, 2009.
[13] E. Solutions, "Analysis tools for DDR1, DDR2, DDR3, embedded DDR and fully buffered DIMM modules," 2014.
[14] R. Housley, "Using advanced encryption standard (AES) counter mode with IPsec encapsulating security payload (ESP)," IETF, RFC 3686, pp. 1–19, 2004. [Online]. Available: https://doi.org/10.17487/RFC3686
[15] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Proc. 19th Annu. Int. Cryptol. Conf. Adv. Cryptol. (CRYPTO)*, Santa Barbara, CA, USA, Aug. 1999, pp. 388–397.
[16] S. Mangard, "A simple power-analysis (SPA) attack on implementations of the AES key expansion," in *Proc. Int. Conf. Inf. Security Cryptol.*, 2002, pp. 343–358.
[17] E. Prouff and M. Rivain, "Masking against side-channel attacks: A formal security proof," in *Proc. Annu. Int. Conf. Theory Appl. Cryptograph. Technol.*, 2013, pp. 142–159.
[18] J. Blömer, J. Guajardo, and V. Krummel, "Provably secure masking of AES," in *Proc. 11th Int. Workshop Selected Areas Cryptography (SAC)*, 2004, pp. 69–83.
[19] S. Nikova, C. Rechberger, and V. Rijmen, "Threshold implementations against side-channel attacks and glitches," in *Proc. Int. Conf. Inf. Commun. Security*, 2006, pp. 529–545.
[20] A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang, "Pushing the limits: A very compact and a threshold implementation of AES," in *Proc. 30th Annu. Int. Conf. Theory Appl. Cryptograph. Techn. Adv. Cryptol. (EUROCRYPT)*, 2011, pp. 69–88.
[21] A. Jati, N. Gupta, A. Chattopadhyay, S. K. Sanadhya, and D. Chang, "Threshold implementations of GIFT: A trade-off analysis," *IEEE Trans. Inf. Forensics Security*, vol. 15, pp. 2110–2120, Dec. 2020.
[22] B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen, "A more efficient AES threshold implementation," in *Proc. Int. Conf. Cryptol. Africa*, 2014, pp. 267–284.
[23] T. Unterluggauer, M. Werner, and S. Mangard, "MEAS: Memory encryption and authentication secure against side-channel attacks," *J. Cryptograph. Eng.*, vol. 9, no. 2, pp. 137–158, 2019.
[24] T. Feist, "Vivado design suite," *White Paper*, vol. 5, p. 30, 2012.
[25] *Zynq-7000 All Programmable SOC: Technical Reference Manual. UG585, V1.8.1*, Xilinx, San Jose, CA, USA, 2014.
[26] S. Siamashka. (2019). *TinyMemBench*. [Online]. Available: https://github. com/ssvb/tinymembench
[27] D. H. Bailey, "Some background on Kanada's recent Pi calculation," May 2003.
[28] J. M. Borwein, P. B. Borwein, and D. H. Bailey, "Ramanujan, modular equations, and approximations to Pi or how to compute one billion digits of pi," *Amer. Math. Monthly*, vol. 96, no. 3, pp. 201–219, 1989.
[29] S. Rabinowitz and S. Wagon, "A spigot algorithm for the digits of π," *Amer. Math. Monthly*, vol. 102, no. 3, pp. 195–203, 1995.
[30] B. Mandelbrot, *Fractals and Chaos: The Mandelbrot Set and Beyond*. New York, NY, USA: Springer, 2013.
[31] G. Marsden, M. McDonald, and M. Brackstone, "Towards an understanding of adaptive cruise control," *Transp. Res. C Emerg. Technol.*, vol. 9, no. 1, pp. 33–51, 2001.
[32] B. Wymann *et al.* (1997). *TORCS, The Open Racing Car Simulator*. [Online]. Available: https://sourceforge.net/projects/torcs/
[33] D. K. Chaturvedi, *Modeling and Simulation of Systems Using MATLAB and Simulink*. Hoboken, NJ, USA: CRC Press, 2017.
[34] R. Barry *et al.*, "FreeRTOS," Oct. 2008.
[35] V. K. Sundar and A. Easwaran, "A practical degradation model for mixed-criticality systems," in *Proc. IEEE 22nd Int. Symp. Real Time Distrib. Comput. (ISORC)*, 2019, pp. 171–180.
[36] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.
[37] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.
[38] L. Deng, "The MNIST database of handwritten digit images for machine learning research [best of the Web]," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 141–142, Nov. 2012.
[39] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," EECS Dept., Univ. California at Berkeley, Berkeley, CA, USA, Rep. UCB/EECS-2014-146, 2014.
[40] J. Hennessy *et al.*, "MIPS: A microprocessor architecture," *ACM SIGMICRO Newslett.*, vol. 13, no. 4, pp. 17–22, 1982.

Naina Gupta received the bachelor's degree from Guru Gobind Singh Indraprastha University, New Delhi, India, in 2013, and the master's degree from the Indraprastha Institute of Information Technology (IIIT), New Delhi, in 2015. She is currently pursuing the Ph.D. degree with the School of Computer Science and Engineering, Nanyang Technological University (NTU), Singapore.

She has worked as a Researcher with IIIT from 2015 to 2017 and as an intern with NTU from 2017 to 2018. Her research interests are cryptography, side-channel attacks, hardware security, postquantum cryptography, and efficient implementations of cryptographic algorithms.

Arpan Jati received the bachelor's degree from West Bengal University of Technology (WBUT), Kolkata, India, in 2010, and the master's degree from the Indraprastha Institute of Information Technology, New Delhi, India, in 2013, where he is currently pursuing the Ph.D. degree.

Since 2018, he has been working as a Research Associate with the Physical Analysis and Cryptographic Engineering Lab, School of Physical and Mathematical Sciences, Nanyang Technological University, Singapore. His research interests are cryptography, side-channel attacks, hardware security, efficient implementations of cryptographic algorithms, blockchain protocols, and circuit design.

Anupam Chattopadhyay (Senior Member, IEEE) received the B.E. degree from Jadavpur University, Kolkata, India, in 2000, the M.Sc. degree from Advanced Learning and Research Institute, Lugano, Switzerland, in 2002, and the Ph.D. degree from RWTH Aachen University, Aachen, Germany, in 2008.

From 2008 to 2009, he worked as a member of Consulting Staff with CoWare R&D, Noida, India. From 2010 to 2014, he led the MPSoC Architectures Research Group, RWTH Aachen University as a Junior Professor. Since September 2014, he has been an Assistant Professor with the School of Computer Science and Engineering, Nanyang Technological University, Singapore, where he got promoted to an Associate Professor (with tenure) in August 2019. In the past, he was a Visiting Professor with the École polytechnique fédérale de Lausanne, Switzerland, and Indian Statistical Institute, Kolkata. His research advances have been reported in more than 150 conference/journal papers (ACM/IEEE/Springer), multiple research monographs and edited books (CRC and Springer), and open-access forums. Together with his doctoral students, he proposed novel research directions, such as domain-specific high-level synthesis for cryptography, high-level reliability estimation flows for embedded processors, generalization of classic linear algebra kernels, and multilayered coarse-grained reconfigurable architecture. His research in the area of emerging technologies has been covered by major news outlets across the world, including Asian Scientist, Straits Times, and The Economist.

Dr. Chattopadhyay received the Borchers Plaque from RWTH Aachen University for outstanding doctoral dissertation in 2008 and the nomination for Best IP Award at DATE 2016.