Memory Encryption for Smart Cards

Barış Ege[1], Elif Bilge Kavun[2], and Tolga Yalçın[2]

[1] Institute of Applied Mathematics,
Middle East Technical University, Turkey
`baris.ege@metu.edu.tr`
[2] Chair of Embedded Security, HGI,
Ruhr University Bochum, Germany
{`elif.kavun,tolga.yalcin`}`@rub.de`

Abstract. With the latest advances in attack methods, it has become increasingly more difficult to secure data stored on smart cards, especially on non-volatile memories (NVMs), which may store sensitive information such as cryptographic keys or program code. Lightweight and low-latency cryptographic modules are a promising solution to this problem. In this study, memory encryption schemes using counter (CTR) and XOR-Encrypt-XOR (XEX) modes of operation are adapted for the target application, and utilized using various implementations of the block ciphers AES and PRESENT. Both schemes are implemented with a block cipher-based address scrambling scheme, as well as a special write counter scheme in order to extend the lifetime of the encryption key in CTR-mode. Using the lightweight cipher PRESENT, it is possible to implement a smart card NVM encryption scheme with less than 6K gate equivalents and zero additional latency.

Keywords: memory encryption, smart card, low-latency block cipher, AES, PRESENT

1 Introduction

Smart cards and devices containing smart card ICs, have been playing increasingly important roles in our daily lives for years. Within the last decade, passports, credit cards, ID cards, and even public transport tokens have come to rely on smart card technologies. Non-volatile memories (NVMs) in those smart cards contain frequently updated data such as personal information about the card holder, and less frequently (or never) changing data such as program code, cryptographic keys, etc. Access to the frequently updated data should be executed with very low latency. Due to advances in attack methods, e.g., invasive attacks, it is often realistic to expect that the information would be compromised in case of a stolen or lost card. Therefore it is crucial to store the information securely using state-of-the-art cryptographic algorithms and technologies, and yet in an economic way.

This requires some form of data encryption algorithm to be implemented on the smart card, either in hardware or software. However, encryption (and decryption) of memory data comes with a specific set of requirements, which are not met with standard cipher implementations. For instance, while AES is extremely well tested and secure, it requires either several tens of thousands of gates for a fast hardware implementation, or several hundreds of cycles for software implementation on a standard embedded processor. On the other hand, mechanisms that allow access to data stored on the NVM of a smart card should be lightweight in terms of power consumption, latency and resource usage. This enforces utilization of specifically optimized algorithms and modes.

The memory encryption schemes proposed up to date mainly target real-time hard disk encryption [1–7], which differs considerably from the security in smart cards. They require lots of temporary storage for predictive processing of data, which makes the overall process seem like zero-latency to the external user.

On smart cards, NVM sizes are usually restricted. Even the future visions for the next two decades predict at most a few gigabytes of NVM storage[8]. In most cases, NVM also stores program memory, which has to be accessed with very low latency for seamless execution of the program. Standard memory encryption techniques usually require several clock cycles in order to decrypt the page header, calculate the tweak, and then use this information to further decrypt consecutive block data. This is unacceptable for low latency access demanding smart card NVM applications. Therefore, either the existing techniques have to be improved, or new memory encryption techniques have to be introduced together with supporting cryptographic primitives.

When considering security of smart card NVMs, the following assumptions have to be considered:

- An adversary can read the raw contents of the NVM at any time (which might be possible with either invasive attacks or attacks targeting the access mechanism).
- An adversary can fool the smart card to encrypt and store arbitrary data of his/her choosing on the NVM.
- An adversary can modify unused sectors on the NVM and then request their decryption.
- Furthermore, most data stored on the NVM has a predefined structure, such as headers, footers, fixed size and address, etc.

In order to resist such an adversary, the security module on the smart card should be able to efficiently encrypt memory contents of NVM and perform memory address scrambling. Scrambling (re-mapping of the address space) prevents the adversary from locating the exact location of the stored data. It is a crucial element of memory encryption, without which most memory encryption schemes become pointless.

The memory encryption on smart cards should be low-cost, security proven, low-latency, and low-complexity. In this work, we this target is accomplished by adapting CTR and XEX-modes of operation for memory encryption, and realizing them using AES[9] and PRESENT[10] block ciphers. The proposed systems are implemented at RTL level, synthesized using standard CMOS technology, and their performance figures are presented.

The rest of the paper is organized as follows: In section 2, an overview of existing and proposed memory encryption schemes is presented together with security discussion and performance analysis of two selected schemes for smart card NVM encryption. System design details and performance results are presented in section 3. Finally in section 4, the results are summarized.

2 Memory Encryption

2.1 Memory Encryption Background

Like most other fixed width data applications, memory encryption also relies on use of block ciphers. However, using a block cipher as the cryptographic primitive for memory encryption makes sense only when it is used with a proper mode of operation. In addition to the conventional counter mode (CTR), new modes of operation specific to memory encryption have been proposed: namely Liskov-Rivest-Wagner (LRW), XOR-Encrypt-XOR (XEX)[7] and XEX-based Tweaked CodeBook mode with ciphertext stealing (XTS)[6]. All three proposed schemes (LRW, XEX, XTS) depend on the tweakable block cipher idea in [11], which is composed of a block cipher and a tweak. This system is proven to be secure against the chosen-ciphertext and chosen-plaintext attack scenarios, provided that the number of plaintext-ciphertext pairs is limited. Basically, a pseudo-random tweak is computed for each block and then XORed with the input and output of the cipher, thereby randomizing any structure in the plaintext and ciphertext and thwarting attacks exploiting these structures. The CTR, LRW, XEX, and XTS modes of operation are briefly summarized below:

CTR-Mode CTR-mode mode enables the user to have parallel encryption and random access to the encrypted data. This makes it a viable candidate for memory encryption. However, it was shown that this mode is insecure when more than $2^{\frac{n}{2}}$ blocks of data are encrypted with a block cipher with block size n. The idea in the counter mode is to produce a one-time pad (OTP) using the underlying block cipher and XOR the plaintext with this OTP. This approach is quite problematic when used recklessly in a memory encryption scheme, as will be seen in Section 2.3.

LRW-Mode LRW-mode randomizes the input and output of the block cipher by XORing both with a tweak value, i.e. $C = E_{K_1}(P \oplus T) \oplus T$, where $T = K_2 \otimes I$ ($\otimes$ being multiplication over GF(2^n). Here, K_1 is the key to the block cipher, K_2 is a key of the same size as the block size of the underlying block cipher and I is the counter/index of the data to be encrypted. This mode of operation is directly applicable to memory encryption when I is used as the address of the block to be encrypted. However, for non-sequential data, a full GF(2^n) multiplier is required.

XEX-Mode Proposed by Rogaway in 2004 [7], XEX-mode enables the user to have an easier computation of the tweak T (Figure 1), i.e. $T = E_{K_1}(N) \otimes 2^I$. As a result, a full finite field multiplier is not required. Tweak computation gets even easier when encrypting sequential data, where it becomes simple doubling over GF(2^n).

XTS-mode XTS-mode is standardized as IEEE standard 1617 for Cryptographic Protection of Data on Block-Oriented Storage Devices in 2008 [6]. It is basically the same as the XEX-mode. However, a second key, K_2, is used in tweak compution, i.e. $T = E_{K_2}(N) \otimes 2^I$.

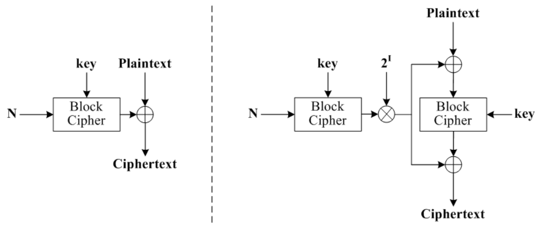

Fig. 1. Graphical illustration of the CTR and XEX-modes of operation

2.2 Previous Work

The previously proposed schemes for memory encryption mostly target everyday PCs or even larger systems, and they use large amounts of resources. In [1], CRYPTOPAGE extension of the HIDE infrastructure is proposed in addition to the address bus protection, memory encryption and memory checking which are combined in a way to provide a very low performance overhead. [2] presents predecryption as a method of providing security with less overhead by using well-known prefetching techniques to collect data from memory and perform decryption before it is needed by the processor. Their results show no increase in execution time despite an extra 128 cycle decryption latency per memory block access. In [4, 3], a technique is applied to hide the latency overhead of memory decryption (which is encrypted in CTR-mode) by predicting the sequence number and precomputing the OTP. This technique solves the latency problem by using idle decryption engine cycles to speculatively predict and precompute OTPs before the corresponding sequence number is loaded. Also, an adaptive OTP prediction technique is presented to further improve OTP prediction and precomputation mechanism. This scheme is not only able to predict encryption pads associated with static and infrequently updated cache lines, but also the frequently updated cache lines as well. [4] presents new hardware mechanisms for memory integrity verification and encryption. The integrity verification mechanism offers better performance when the checks are infrequent as in grid computing applications, and the encryption mechanism improves the performance in all cases. [12] shows a hardware implementation of an execute-only memory (XOM) form which allows instructions stored in memory to be executed but not manipulated in another way. In this work, all data that leaves the machine is encrypted, becuse the external memory is assumed to be insecure. However, for efficient operation, hardware assist to provide fast symmetric ciphers is also required. Some other works on memory-bus encryption [13, 14] mention hardware engines for bus encryption. [13] describes an engine called "Parallelized Encryption and Integrity Checking Engine (PE-ICE)", which guarantees the confidentiality and integrity of data exchanged between a system-on-chip (SoC) and its external memory. This approach is based on an existing block-encryption algorithm, to which the integrity checking capability is added. According to [13], it results in low performance overhead. In [14], a comprehensive survey on existing techniques for hardware engines which are used in bus encryption is presented.

Apart from these, there exists proprietary data bus encryption algorithms for on-chip NVMs in use by several chip vendors. However, these are not publicly accessible, and to the best of our knowledge, unlike us, none of the existing schemes focus on the cipher block in order to come up with a lightweight and cacheless solution.

2.3 Memory Encryption System Design Issues

So far, we have covered previously proposed modes of operation for memory encryption in the literature. In this study, our main concern is encyption on smart card NVMs. For this purpose, we choose two target modes, CTR and XEX, basically due to the simplicity and low implementation cost of CTR-mode (which make it weak in terms of security), and solid security proof and single key requirement of XEX-mode. In the rest of this section, we shall investigate design and implementation issues for both modes. Additionaly we shall introduce a simple address scrambling scheme, which is a practical requirement for all memory encryption systems.

In CTR-mode, the address α is encrypted using the encryption key, generating the OTP. Part or whole of OTP is XORed with the plaintext, P, or the ciphertext, C, resulting in ciptertext or plaintext outputs, respectively. In cases where OTP size is equal to or more than twice the plaintext/ciphertext block size (i.e. using a 64-bit block cipher with 32-bit memory), it is possible to use only part of OTP to encrypt/decrypt data.

This is not the case for XEX-mode, where a data pair (or even quartet) is required in order to be able to decrypt only a single word within the memory. Referring to the same 64-bit block cipher with 32-bit memory case, we see that updating an address within memory requires first decryption of the old contents of the target address together with the contents of its neigbouring address (which together form a 64-bit block), then writing to both addresses after encryption. This scheme aggravates the average latency. Furthermore, XEX requires both encryption and decryption modules, or a combined module in order to reduce area at the cost of further additional latency.

Clearly, counter mode presents a much simpler and compact solution, and a lot less implementation problems to deal with. This, of course, comes at the cost of lower security with respect to that of XEX-mode. This problem can be remedied with an address scrambling scheme, as we shall see in the next subsection. Furthermore, we modify the CTR-mode in order to have a cryptographically sound scheme, and introduce a write counter as explained later.

2.4 Address Scrambling

As mentioned before, address scrambling is an integral element of memory encryption and should be handled with care. The most important aspect to address scrambling is that the order of write addresses should not have any visible structure, or in other words, the order that the data is written to the memory should look random to an adversary. In this work, a key dependent address scrambling is proposed using a small scale block cipher. A small scale variant of the well known and secure lightweight block cipher PRESENT[15] is used for this purpose.

In our sample system, 16-bit memory address is encrypted via 4 rounds of the variant cipher, and the resultant ciphertext is used as the scrambled memory address. Simulations show that the auto-correlation of the 4-round output of the 16-bit version of the block cipher PRESENT gives a satisfactory distribution (see Figure 2). It is possible to store the regular 80-bit key and perform key expansion for each of the 4-rounds. Alternatively, 16-bit keys for the 5 rounds (which also sum up to 80-bits) can be stored directly. This way, not only the key expansion logic can be strip off the design, but completely random keys without any dependency in any round can be stored at no addition cost. Naturally, address scrambling requires $(4 + 1) \times 16 = 80$ bits of additional registers for key storage. It is also possible to use part or whole of data encryption keys for address scrambling. However, this requires careful investigation of the security implications.

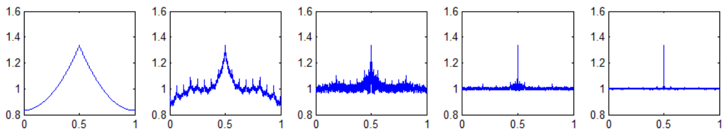

Fig. 2. Auto correlation of the input and the outputs of the small scale PRESENT for rounds 1 to 4

2.5 Performance Issues

In this subsection, we discuss how the performances of CTR and XEX-modes in terms of average latency per memory operation are affected by the cipher latency and the page size (or packet size in case of CTR). First, let's briefly discuss issues with both modes.

Issues with CTR In CTR-mode, the memory is divided into blocks that are identified by unique headers. For each block, a write counter is used. This scheme works as follows: Target write address width, which is the input to the encryption, is for all practical purposes much smaller than the cipher block size. Instead of using all zeros to pad it, a predefined address field (for example, most significant 5-bits) is reserved as the write counter. At each update of the block data (i.e. writing of the whole block), the address counter is incremented by one, resulting in a completely new OTP for the same addresses with respect to previous writes. In order to be able to decrypt a block, the header address is decrypted first, starting with zero write counter value, and incrementing it by one until the correct header value is read. The resultant write counter is used for the decryption of the rest of the data within the same memory block. This way, the uniqueness of the IV of each encryption is guaranteed. Although this is an expensive solution in terms of latency, larger packet sizes make up for the initial write counter value search delay.

Issues with XEX Memory encryption using XEX-mode of operation requires the memory to be divided into pages, so that only one address encryption is enough for each page. The rest of the page is encrypted using the same encrypted page address information and the index of the word within the page (see Figure 1). However, XEX requires both encryption and decryption, which results in doubling of the hardware area of the module, or doubling of the latency in case a single module is used for both operations.

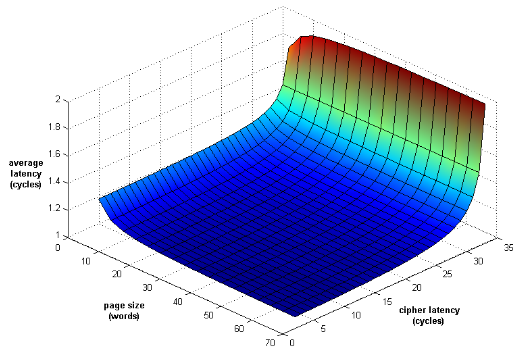

Fig. 3. Average latency of the proposed XEX scheme as a function of page size and cipher latency

For the evaluation of the performance, we assume a generic block cipher which can have up to 32 cycles latency to complete a single encryption/decryption operation. For simplicity, memory word width is assumed to be equal to the cipher block width. In our MATLAB model, we sweep the memory page size (packet size for CTR) from 1 to 64 words, and the cipher latency from 1 to 32, and then measure the average latency per memory operation (assuming equal number of reads and writes). Our simulations clearly indicate that cipher latency has a much higher impact than page and packet sizes over average latency (see Figure 3). As seen in the figure, page size has a significant effect only for very small values. As it goes above 10-15 words, its effect diminishes. However, the effect of cipher latency on the average latency increases almost exponentially. It should also be noted that the figure only shows the average latency for XEX-mode. In CTR-mode, unless the write counter mode is implemented, the average latency is equal to the cipher latency. Clearly, it is imperative that the cipher latency should be kept at minimum.

2.6 System Performance and Block Cipher Selection

Simulation results show that the best way to decrease average latency in memory encryption is to have a low latency cipher. This can be achieved in a number of ways. One is to simply design a low-latency block cipher from scratch, but this requires extensive analysis, expertise and time. Another approach would be to take a well analyzed block cipher and use reduced versions of that cipher, in terms of the number of rounds it requires to encrypt a message block. Although this approach is relatively more plausible, the security analysis done on the full cipher does not directly apply to the reduced round versions of it.

Another approach to achieve low-latency encryption/decryption is to implement a cascade block cipher, i.e. to execute more than one round in one cycle. In the remainder of this work, we implement several rounds of a block cipher (r rounds in one cycle) in order to come up with $\frac{n}{r}$ rounds for the encryption process. Inevitably, this approach brings a trade-off between the latency in terms of the number of rounds an encryption requires and the logical delay of the circuit.

The overall path delay is naturally a function of the underlying cipher function used. In our system, we consider AES and PRESENT as both of these block ciphers have gone through extensive cryptanalysis and remain unbroken. AES-128 requires only 10 rounds per encryption/decryption, but it requires relatively large area in hardware. On the other hand, although it requires 31-rounds, PRESENT-80 is a much more compact cipher, and therefore one can push the cascade implementation to the limit with relatively low area/dealay overhead. Implementation aspects and simulation results for cascade AES and PRESENT are explained in detail in Section 3.

3 Implementation Aspects and Simulations

In this study, we have decided to focus on two candidate systems, XEX and CTR, for reasons explained before. While XEX mode offers higher security compared to CTR mode, it also results in a more complex structure. Most important of all, XEX requires use of both encryption and decryption cores, and therefore is not very suitable for lightweight purposes. In addition, it is not suitable for low-latency requirements, because using 64 and/or 128-bit cipher with 32-bit memories requires reading (and therefore decryption) of neighboring words in order to form the 64/128-bit complete blocks for each write operation, and noticeably increases the average latency. However, CTR system presents more suitable results for both lightweight and low-latency purposes.

In order to give a comparison of both schemes, XEX and CTR-based systems are designed separately. In the first step, variants of AES and PRESENT ciphers, which are basic building blocks for both systems, are implemented in order to obtain performance figures. Furthermore, a reduced round and reduced width version of the PRESENT cipher is utilized as the memory address scrambler block.

Following this step, an XEX-based system is presented together with performance figures. The same is done for a CTR-based system as well. The before-mentioned address scrambling block is used in both modules. Additionally, the write counter scheme mentioned in 2 is also implemented as part of the CTR-mode based system. In the following two subsections, design and implementation details of each cipher core for various cascade combinations are explained in detail. Then, the proposed compact XEX and CTR-mode systems and their performance figures for variants of AES and PRESENT are given in the last subsection.

3.1 AES Round Function and Core Design

Advanced Encryption Standard (AES) [9] is a block cipher with a fixed block size of 128 bits and a key size of 128, 192 or 256 bits. In this design, only 128-bit key size (number of rounds $Nr = 10$) is used, as the main concern is to have a lightweight core. AES encryption and decryption pseudocodes are given as:

```
# Encryption
AddRoundKey(state, w[0,3])
for round=1 step 1 to Nr-1
  SubBytes(state)
  ShiftRows(state)
  MixColumns(state)
  AddRoundKey(state, w[4*round,4*(round+1)-1])
end for
SubBytes(state)
ShiftRows(state)
AddRoundKey(state, w[4*Nr,4*(Nr+1)-1])
```

```
# Decryption
AddRoundKey(state, w[4*Nr,4*(Nr+1)-1])
for round=Nr-1 step -1 to 1
  InvShiftRows(state)
  InvSubBytes(state)
  AddRoundKey(state, w[4*round,4*(round+1)-1])
  InvMixColumns(state)
end for
InvShiftRows(state)
InvSubBytes(state)
AddRoundKey(state, w[0,3])
```

Both encryption and decryption paths were designed and combined together in one block with encryption/decryption select. In addition, reverse key generation, which is required for decryption phase, is handled within the module.

AES Encryption Path As shown in Figure 4, encryption path of AES is a fully parallel implementation of the algorithm. It consists of state round and key round blocks, which handle state operations and key expansion operations, respectively. By connecting several of these paths, it is possible to implement any cascade configuration AES. State round block has a combined ShiftRows & MixColumns module and a SubBytes module. ShiftRows part is the direct mapping of the ShiftRows algorithm and MixColumns is optimized for each coefficient as in [16]. SubBytes module is implemented as a composite S-Box [17]. Key expansion, shown in Figure 4, is performed on-the-fly in parallel to state processing.

AES Decryption Path Decryption path of AES is designed in a structure similar to encryption path. It is also a fully parallel implementation of the algorithm optimized for area, as shown in Figure 4. It consists of state round and key round blocks as in the previous path. However, here the state round block has a combined InvShiftRows & InvMixColumns module to handle inverse cipher operations. In addition, there is InvSubBytes module, whose S-Box has the same composite inverter structure [17]. Reverse key expansion is performed on-the-fly in parallel to state processing, similar to encryption path, which is shown in Figure 4.

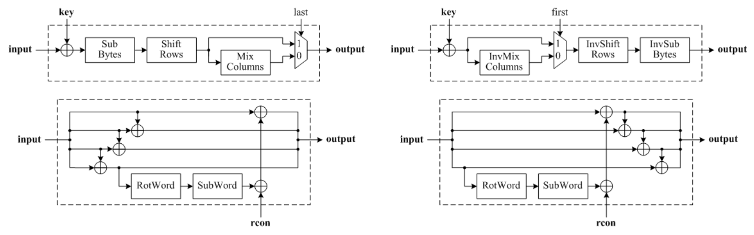

Fig. 4. AES encryption & decryption paths and forward key expansion

Combined AES Core In the combined AES core, encryption and decryption state processing and key expansion paths are integrated as shown in Figure 5. As the S-Box unit is the most area consuming unit of the core, the block is implemented in a way that the composite inverter [17] is shared between two paths in order to save area. The S-Box block is partitioned as a finite field inverter and pre/post matrix multiplications (transformations), resulting in the block in Figure 5. As a result of this approach, the area of this combined core is smaller than the total of encryption and decryption paths.

Key round block implements the key expansion for both encryption and decryption. Since both modes use the same S-Box, the only additional cost for the combined key expansion is using extra multiplixers, with further area savings (Figure 5).

Decryption Key Generation Initially, only the encryption key is stored in the key register. However, reverse key expansion for decryption requires use of

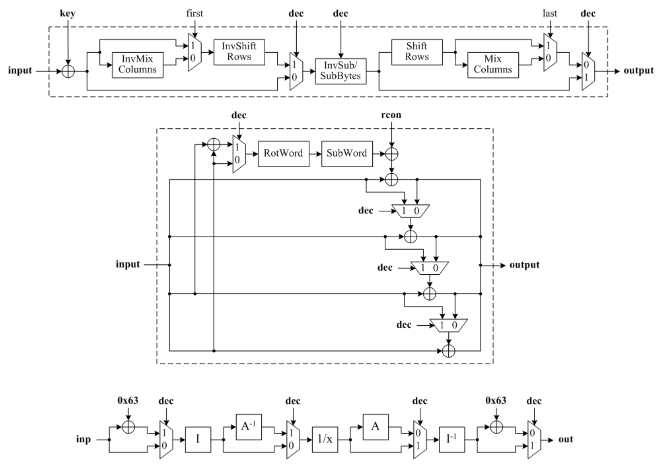

Fig. 5. Combined AES block with key expansion and AES S-Box

the last state of key expansion for encryption as the initial "decryption" key. In order to avoid using decryption key registers, a simple key generation scheme is applied: A dummy key expansion for encryption is executed before switching to decryption every time. As a result, the last state of the encryption key is generated and then saved in the key register as the decryption key. When the core needs to switch from decryption to encryption, similar operation is performed: A dummy reverse key expansion is executed, which generates the last state of the decryption key and writes it back on to the key register as the encryption key. Status of the key (encryption/decryption key) is stored in a 1-bit register as the key status, and control logic handles the dummy key expansions.

Area, Speed, Power Figures The combined AES core is implemented in various cascade combinations (1, 2, 5, and 10) as shown in 6, which results in varying latencies, gate counts, and operating frequencies (see Table 1). All combinations are synthesized and the obtained performance figures are used in the implementation of the memory encryption unit as guidelines.

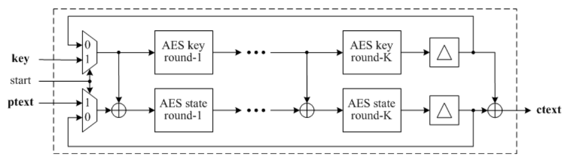

Fig. 6. Cascade implementation of AES

Table 1. Performance of different AES cores

Processor Core	Gate Count (GE)	Max Speed (MHz)	Power (uW/MHz)
acore 1-cyc (enc/dec)	115415	15,7	590,3
acore 1-cyc (enc)	76950	21,0	361,4
acore 2-cyc (enc/dec)	62307	29,5	273,3
acore 2-cyc (enc)	39790	39,6	168,9
acore 5-cyc (enc/dec)	26865	71,7	100,5
acore 5-cyc (enc)	17785	99,4	63,6
acore 10-cyc (enc/dec)	15132	137,2	45,4
acore 10-cyc (enc)	10458	191,2	30,1

3.2 PRESENT Round Function and Core Design

PRESENT [10] is a lightweight block cipher which is an SP-network that consists of 31 rounds. The block length is 64 bits and two key lengths of 80 and 128 bits are supported. For targeted lightweight applications, 80-bit PRESENT provides enough security with smaller area. PRESENT round function is defined as:

```
generateRoundKeys()
for round=1 31
  addRoundKey(state,Ki)
  sBoxLayer(state)
  pLayer(state)
end for
addRoundKey(state,K32)
```

Encryption and decryption paths for PRESENT were designed and combined together in a block. Reverse key generation required for decryption phase is handled within the module, as in the AES design.

PRESENT Encryption Path PRESENT encryption path is designed to be the fully parallel implementation of the algorithm, as shown in Figure 7. There are two blocks: state round and key round blocks, for state operations and key expansion operations, respectively. State round block has a Permutation and an S-Box module. Key expansion, shown in Figure 7, is performed on-the-fly in parallel with the state processing.

PRESENT Decryption Path Decryption path of PRESENT is designed similar to encryption path as a fully parallel core optimized for area. It consists of reverse state round and reverse key round blocks as in the previous path, as shown in Figure 7. The reverse state round block has an inverse permutation module as well as inverse S-Boxes. The key unit is also adapted for on-the-fly reverse key expansion, which is shown in Figure 7.

Combined PRESENT Core The combined PRESENT core (Figure 8) puts both encryption and decryption paths together. Unlike the AES S-Box, the PRESENT S-Box can not be formulated in a finite field. Therefore, both regular

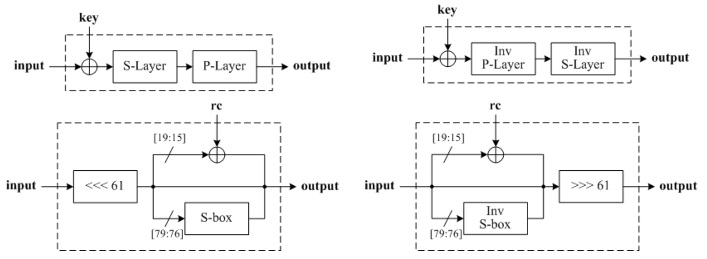

Fig. 7. PRESENT encryption & decryption paths and forward key expansion

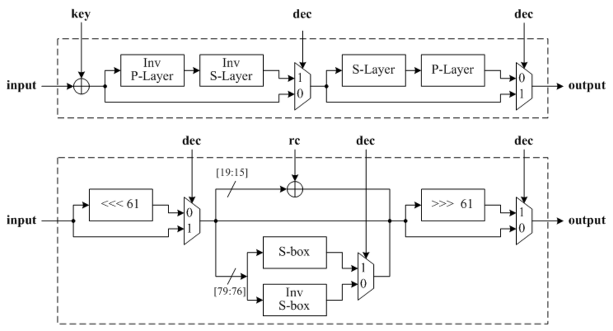

Fig. 8. Combined PRESENT block with key expansion

and inverse S-Boxes are used in the combined core, and selected via multiplexers. The direct result of this fact is the almost doubling of the gate count. The same approach is also applied to the key expansion unit, which has to use both regular and inverse S-Boxes, as well as left and right rotations in order to handle both encryption and decryption. Direct result is again doubled gate count.

Decryption Key Generation The decryption key generation scheme of the AES core is also used in the PRESENT core. That is, in switching from encryption to decryption and from decryption to encryption, dummy forward and reverse key expansions are executed in order to generate and store the decryption and encryption keys, respectively.

Area, Speed, Power Figures As in the case of AES, various cascade versions (1, 2, 4, 8, 16, and 32) of PRESENT are implemented as shown in Figure 9, and the resulting performance figures (Table 2) are later used as guideline in the design and implementation of memory encryption units.

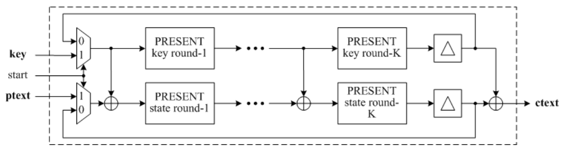

Fig. 9. Cascade implementation of PRESENT

Table 2. Performance of different PRESENT cores

Processor Core	Gate Count (GE)	Max Speed (MHz)	Power (μW/MHz)
pcore 1-cyc (enc/dec)	45700	32,4	172,9
pcore 1-cyc (enc)	20088	48,8	113,2
pcore 2-cyc (enc/dec)	25012	62,3	85,6
pcore 2-cyc (enc)	11863	90,4	47,1
pcore 4-cyc (enc/dec)	13273	122,4	42
pcore 4-cyc (enc)	6744	191,2	22,4
pcore 8-cyc (enc/dec)	7417	228,3	20,1
pcore 8-cyc (enc)	4208	350,9	11,9
pcore 16-cyc (enc/dec)	4547	390,6	10,1
pcore 16-cyc (enc)	2937	429,2	6,4
pcore 32-cyc (enc/dec)	3116	574,7	7,1
pcore 32-cyc (enc)	1839	598,8	3,5

3.3 Memory Encryption Module Design

The I/O signals for the memory encryption module are shown in Figure 10. From the processor point of view, the memory encryption module acts like an ordinary single-port memory. However, there is one important difference: The chip enable signal, *cen*, also acts as a memory access request signal. Once it is asserted, the control module inside the memory encryption unit starts the initialization. In the case of XEX-mode, this corresponds to the encryption of the page address.

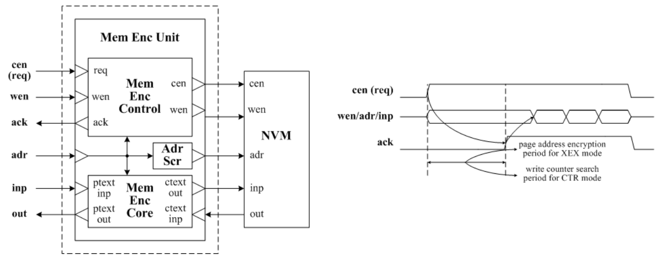

Fig. 10. Memory encryption unit I/O and timing

The situation is a bit different for the CTR-mode. During regular memory accesses, the counter mode requires no initialization, for example when it accesses the program memory code (i.e. only reads data). However, in case of repeatedly written data, the CTR-mode memory encryption unit can be operated in a special mode, where each write to a data packet is counted, and appended to the start of memory address (write counter). This scheme allows use of the same key over and over again until the predefined maximum counter value is reached. However, one has to know the header for every packet. Additionally,, and as the number of newly written data increases, the average memory access time also increases. Therefore, a maximum write counter value has to be determined as part of system design.

Upon completion of the initialization, the memory unit is ready to operate in sequential memory access mode. It asserts the acknowledge signal, *ack*, asking the processor side to send the next query. During the next query, the request (chip enable) signal stays asserted, and only de-asserted during page and/or packet switching (depending on the packet data). The timing for this operation is shown in Figure 10.

During each memory access, the address is scrambled using a reduced width and reduced round of the PRESENT cipher, which is a combined block with no clock cycle losses. From the memory (NVM) point of view, the memory encryption unit acts like an ordinary processor. The memory encryption unit acts as a bridge between the smart card processor and the memory. The most important parameter of this translation process is the average cipher latency. In an ideal case, the memory access latency is a single clock cycle. However, due to the initialization latencies, the average latency increases above the ideal value of single cycle. As shown before, the cipher module latency is the most important parameter in the whole process. Therefore it should be kept as low as possible, ideally at zero additional delay. As will be shown in the following subsections, even in this case, the initialization delay can only be lowered down to 1 cycle.

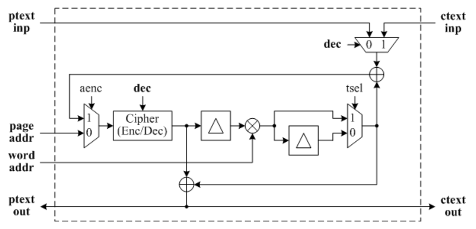

Fig. 11. XEX based memory encryption unit

3.4 XEX-Mode

Figure 11 shows the XEX-mode based memory encryption module. It uses a single cipher core for both page address encryption and data encryption/decryption operations. In page encryption, the cipher core operates in encryption mode, and stores the encrypted page address inside the page address register. It is multiplied with the word address to obtain the tweak value, which is added to both pre and post encrypted/decrypted data. Data encryption (memory write) uses only the instantaneous value of the tweak, whereas decryption (memory read) uses also the delayed value of the tweak. Tweak and page address registers have the same width as the cipher block size.

The XEX-mode based memory encryption unit is implemented using different versions of both AES and PRESENT ciphers. In the implementation, both ciphers are operated in single cycle mode with respect to the memory access clock. Therefore, for different versions of ciphers (except for the fully combinational version), a second cipher clock is required, which should be a multiple of the memory clock with respect to the number of combinational rounds inside the cipher core.

3.5 CTR-Mode

In Figure 12, the CTR-mode based memory encryption module is shown. Its structure is much less complex compared to the XEX-based system. Furthermore, it requires an only-encryption cipher core, which reduces the overall gate count and combinational path delay dramatically. It also does not require the encryption/decryption input data widths to be equal to the cipher width. Therefore, it can be run truly in single cycle mode, resulting in zero additional latency.

The CTR-mode memory encryption module is also implemented using various versions of AES and PRESENT ciphers. As in the case of XEX, it is possible to run the cipher module using a higher rate clock in order to complete the overall encryption process within the same cycle with respect to memory clock.

Table 3 shows the results for the best performing combinations of both modules. As seen in the table, an 8-cycle version of PRESENT, which runs at four times the memory clock rate, gives almost the same throughput result as the 1-cycle version, while occupying only one forth of the area. The result is a security proven ultralight memory encryption scheme implemented in less than 6K gates at zero additional latency.

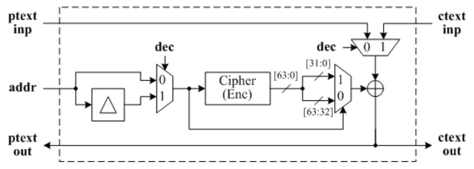

Fig. 12. CTR based memory encryption unit

Table 3. Performance of various XEX and CTR cores

Processor Core	Gate Count	Max Speed
XEX with AES 1-cyc	121,7KGE	15,7MHz
XEX with AES 5-cyc	33,2KGE	14,4MHz
XEX with PRESENT 1-cyc	51,8KGE	32,4MHz
XEX with PRESENT 8-cyc	11,4KGE	28,5MHz
CTR with AES 1-cyc	75,5KGE	21,0MHz
CTR with AES 5-cyc	19,9KGE	19,8MHz
CTR with PRESENT 1-cyc	21,6KGE	48,8MHz
CTR with PRESENT 8-cyc	5,9KGE	43,8MHz

4 Conclusions

In this study, we have investigated the existing memory encryption schemes and their suitability for smart card memory encryption. XEX and CTR-modes were selected as suitable schemes for their proven security and relatively simple structure. Two memory encryption units were designed for each of these schemes, and they were implemented for different versions of AES and PRESENT. PRESENT seems to be an ideal choice for memory encryption when used in CTR-mode. It is possible to implement a PRESENT-based CTR-mode memory encryption module in less than 6K gates with zero additional latency.

The proposed architecture is not only suitable for today's applications, but also capable of fulfilling requirements set forth in the 2023 vision of Eurosmart[8]. We furthermore enhanced our memory encryption module via a block-cipher based address scrambling scheme, where we implemented a small scale variant of PRESENT. For commercial applications, which require even more security, it is also possible to replace the standard PRESENT S-Boxes with secret proprietary S-Boxes to achieve higher levels of security.

References

1. Guillaume Duc and Ronan Keryell. Cryptopage: An efficient secure architecture with memory encryption, integrity and information leakage protection. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 483–492, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2716-7.
2. Brian Rogers, Yan Solihin, and Milos Prvulovic. Memory predecryption: hiding the latency overhead of memory encryption. *SIGARCH Comput. Archit. News*, 33:27–33, March 2005.
3. Weidong Shi, Hsien-Hsin S. Lee, Mrinmoy Ghosh, Chenghuai Lu, and Alexandra Boldyreva. High efficiency counter mode security architecture via prediction and precomputation. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 14–24, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2270-X.
4. G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Efficient memory integrity verification and encryption for secure processors. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 339–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2043-X.
5. Jun Yang, Youtao Zhang, and Lan Gao. Fast secure processor for inhibiting software piracy and tampering. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 351–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2043-X.
6. IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices. *IEEE Std 1619-2007*, pages c1–32, April 18 2008.
7. Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes ocb and pmac. In Pil Lee, editor, *Advances in Cryptology - ASIACRYPT 2004*, volume 3329 of *Lecture Notes in Computer Science*, pages 55–73. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-30539-2_2.
8. E. Alyanaklan et al. The smart & secure world in 2020. Technical report, Eurosmart, 2007.
9. Joan Daemen and Vincent Rijmen. The block cipher rijndael. In *Proceedings of the The International Conference on Smart Card Research and Applications*, pages 277–284, London, UK, 2000. Springer-Verlag. ISBN 3-540-67923-5.
10. A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe. Present: An ultra-lightweight block cipher. In *Proceedings of the 9th international workshop on Cryptographic Hardware and Embedded Systems*, CHES '07, pages 450–466, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-74734-5.
11. Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable block ciphers. *J. Cryptology*, 24(3):588–613, 2011.
12. David J. Lie. *Architectural support for copy and tamper-resistant software*. PhD thesis, Stanford, CA, USA, 2004. AAI3111747.
13. Reouven Elbaz, Lionel Torres, Gilles Sassatelli, Pierre Guillemin, Michel Bardouillet, and Albert Martinez. A parallelized way to provide data encryption and integrity checking on a processor-memory bus. In *Proceedings of the 43rd annual Design Automation Conference*, DAC '06, pages 506–509, New York, NY, USA, 2006. ACM. ISBN 1-59593-381-6.

14. Reouven Elbaz, David Champagne, Catherine Gebotys, Ruby B. Lee, Nachiketh Potlapally, and Lionel Torres. Hardware mechanisms for memory authentication: A survey of existing techniques and engines. *Transactions on Computational Science IV, Lecture Notes in Computer Science (LNCS)*, pages 1–22, March 2009 2009.
15. Gregor Leander. Small scale variants of the block cipher present. Cryptology ePrint Archive, Report 2010/143, 2010. http://eprint.iacr.org/.
16. Hua Li and Zachary Friggstad. An efficient architecture for the aes mix columns operation. In *ISCAS (5)*, pages 4637–4640. IEEE, 2005.
17. Christof Paar. *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*. PhD thesis, Institute for Experimental Mathematics, Universität Essen, 1994.