Journal of Supercomputing manuscript No.
(will be inserted by the editor)

DKEMA: GPU-Based and Dynamic Key-Dependent Efficient Message Authentication Algorithm

Hassan N. Noura[1], Raphaël Couturier[1], Ola Salman[2], and Kamel Mazouzi[1]

[1]Univ. Bourgogne Franche-Comté (UBFC), FEMTO-ST Institute, France
[2]American University of Beirut, Electrical and Computer Engineering Department, Beirut 1107 2020, Lebanon

the date of receipt and acceptance should be inserted later

Abstract Recently, benefiting from the advancement in the Graphic Processing Unit (GPU) technology, there is an increased interest in implementing and designing new efficient cryptographic schemes. Existing cryptographic algorithms, especially the Message Authentication Algorithms (MAAs) such as Hash Message Authentication Code (HMAC) and Ciphered Message Authentication Code (CMAC) are not designed to benefit from the GPU characteristics, which results in degraded performance of their GPU implementations. This gives rise to a trade-off between the design concept and the performance level. In this paper, a new MAA, called 'DKEMA', is proposed to better suit the GPU functionality. This scheme is based on the dynamic key-dependent scheme with one round of substitution and diffusion operations. The experimental results show that the proposed solution is highly effective on Tesla V100 and A100 GPUs, the throughput is respectively more than 400GB/s and 500GB/s. Therefore, DKEMA can be considered as a promising MAA candidate for GPU implementation, achieving the desired cryptographic properties such as high randomness, collision tolerance in addition to message and key avalanche effect. The experimental results show that the proposed solution, based on the dynamic key approach, is immune towards well-know authentication and cryptanalysis attacks. In addition, DKEMA, consisting of one round compression function, presents an enhancement in terms of performance compared to existing algorithms (e.g. AES and SHA).
The code is available in GitHub: `https://github.com/rcouturier/GPU-DKEMA.git`.

Keywords Lightweight GPU message authentication algorithm; Security and performance analysis; Parallel keyed hash function; Dynamic key dependent cryptographic primitives; Cryptanalysis

Address(es) of author(s) should be given

1 Introduction

With the data and Internet proliferation, there is a need to protecting all kinds of resources and data from various types of threats targeting different security services such as data confidentiality, Data Integrity (DI), and Source Authentication (SA). These security services are typically ensured by employing cryptographic algorithms. Existing attacks can be classified into two main classes: active or passive. While passive attacks can seriously impair the data confidentiality and user privacy, active attacks can compromise the data authentication, integrity, and availability. Moreover, an active attacker may insert, modify or delete data contents. Using Message Authentication Algorithms (MAAs) can solve the problems related to message authentication attacks. However, this requires a distributed scheme and a robust key exchange mechanism. Typically, symmetric-key message authentication or digital signature (asymmetric-key) schemes can be used to ensure data integrity and source authentication. Furthermore, conventional symmetric message authentication algorithms are either block cipher-based or hash-based.

1.1 Problem Formulation

Therefore, for a set of big data applications, there is a need to design an efficient and robust message authentication algorithm (MAA) to strike a good balance between the security level and the performance. However, the existing MAAs, such as CMAC, HMAC, were not designed to be implemented in parallel since they

are based on the chaining block operation mode, where each block requires the output (compressed block) of the previous block. In addition, these solutions require a high number of rounds and consequently high computation complexity and latency overhead.

Thus, to reach better efficiency, MAA should be designed to exploit the main benefits of parallelism. To validate the efficiency of such a scheme, the implementation could be realized on Graphic Processing Units (GPUs) as they are main parallel accelerators in the market. Moreover, a parallel MAA presents an accelerator for high security applications, benefiting from the GPUs included in heterogeneous-based systems.

1.2 Related Work

Existing MAAs use the Merkle-Damgard concept (see Fig. 1) that is based on the chaining operation mode between message blocks. In addition, they use a compression function that iterates a round function for a high number of iterations to reach the desired cryptographic properties and immunity against cryptanalysis. The compression function of MAAs can be divided into two main classes: cipher-based or hash-based.

Cipher-based MAAs use strong block or stream cipher such as the Advanced Encryption Standard (AES) [1]. Examples of well known cipher-based MAAs are: Galois Message Authentication Code (GMAC) or Ciphered Message Authentication Code (CMAC) operation mode. Besides, hash-based MAAs use hash function mechanisms such as the Hash Message Authentication Code (HMAC) that uses the Secure Hash Algorithm (SHA)-2 and SHA-3. In addition, the round function can either be based on a Feistel Network (FN) such as Data Encryption Standard (DES), or on Substitution-Permutation Network (SPN) such as AES [2]. SPN lends itself to parallel implementation and requires a lower number of rounds compared to FN, and hence, SPN exhibits lower latency and requires less resources than FN.

To reduce the execution time of existing cryptographic algorithms, GPU implementations are being adopted. The parallel cryptographic algorithms can benefit from the hundreds and even thousands of cores in a GPU to accelerate and parallelize their required computations.

Standard cryptographic algorithms, such as AES, have been implemented on GPUs with parallel operation modes such as the counter mode [3–5], which resulted

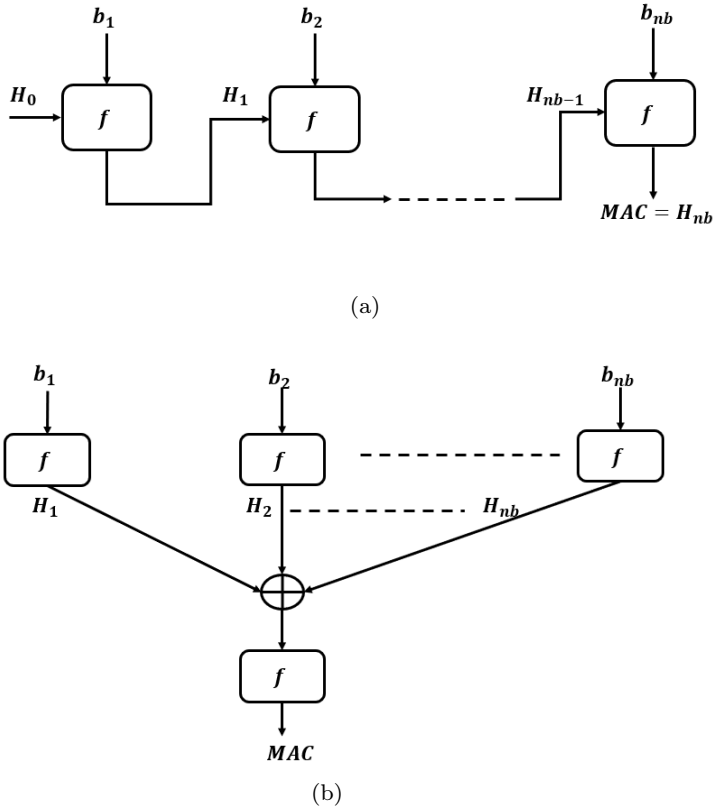


Fig. 1: (a) Structure of the Merkle-Damgard algorithm used traditionally to construct a message authentication algorithm, where H_0 represents a secret key and the message is divided into nb blocks $\{b_1, b_2, \ldots, b_{nb}\}$ that have a fixed size in addition to f, which represents the compression function. (b) Structure of the proposed parallel MAA.

in an impressive speed-up compared to the CPU implementation [6]. It is worth noting that the efficient implementation of an algorithm on a GPU requires the expertise to optimize the use of the GPU architecture [7]. GPUs can also be employed for implementing pseudo-random number generators such as in [8, 9].

Recently, an optimized and efficient implementation of AES on GPU was presented in [5]. It achieved an excellent performance and the authors made considerable optimization compared to the previous works. A more recent implementation of AES on GPU based on the PHAST library was presented in [10]. This implementation is more generic and it resulted in about 10% enhancement in terms of performance compared to [5].

On the other hand, the security level of existing symmetric ciphers, such as AES or keyed hash function

such as HMAC (using SHA-2 or SHA-3), against analytic attacks depends on applying a round function for a higher number of rounds r. This leads to a trade-off between the security level and the required latency and resources. Existing cryptographic algorithms, that are based on the static structure, have proven their resistance against analytic cryptanalysis [2, 11, 12]. However, the static structure of the round function makes these algorithm suffering from different security issues [13]. Unfortunately, fixed cipher structures lend themselves to future potential attacks [11, 12], that benefit from the fixed structure (substitution and diffusion primitives) to recover the secret key [2]. Examples of such attacks include implementation attacks such as side-channel attacks and fault attacks [2]. Hence, countermeasures against implementation attacks are required, at the cost of more latency and resources overhead. This could reduce the performance of the corresponding algorithms and make them unsuitable for some of the future systems and applications [14].

Moreover, since the cipher primitives are static, the required number of rounds r is high to achieve the desired cryptographic properties. Recently, a new kind of cryptographic algorithms is presented to reduce the number of rounds to 1 or 2, where different substitution and diffusion operations are performed within each input message in a dynamic pseudo-random manner [15–17]. Also, another advantage of the dynamic cryptographic approach is that it provides better resistance against implementation attacks.

1.3 Contributions

In this paper, the proposed message authentication solution follows the recent dynamic key-dependent approach of [15, 17, 18]. A high security level is achieved since the cryptographic primitives are updated for each new input message and the message avalanche effect is achieved based on the key avalanche effect.

The proposed message authentication scheme uses an efficient and simple compression function that uses dynamic substitution tables in addition to two different Pseudo Random Number Generators (PRNGs) with a large number of seeds. To the best of our knowledge, the proposed solution is the first message authentication algorithm designed to run on GPU. Indeed, as DKEMA is designed to benefit of parallelism, it is implemented on GPU instead of CPU since GPU provides better parallel computation capabilities (multi-threads concept) compared to CPU.

Therefore, the technical contributions of this paper compared to the existing solutions can be summarized as follows:

- Efficiency: DKEMA is designed to be executed in parallel instead of using the chaining mode as it is the case for most of the existing MAAs. In addition, DKEMA uses a simple compression function that uses simple operations. This compression function is iterated once for each input block. This minimizes the latency compared to existing MAAs that require a higher number of iterations.
- Optimized implementation: The GPU implementation of DKEMA is optimized according to the GPU characteristics (each block is processed by a thread) and its implementation is optimized (with memory management) to reach best performance compared to existing MAAs that cannot be implemented in GPU (due to chaining operation mode).
- Robustness against attacks: DKEMA is based on a dynamic key-dependent approach and cryptographic primitives. This makes it highly resistant against cryptanalysis attacks.
- Adaptability: the DKEMA cryptographic structure can be used to construct an efficient and robust stream-cipher that can be employed to ensure data confidentiality. This means that all data security services can be ensured in this work since DKEMA can ensure data integrity, source authentication and the proposed stream cipher can ensure data confidentiality.

1.4 Organization

The rest of this paper is organized as follows: Section 2 describes and analyzes existing GPU based ciphers implementations. In Section 3, the proposed dynamic key derivation process is presented in addition to the construction techniques of the required cryptographic primitives. Then, in Section 4, DKEMA structure and its corresponding compression function, along with the functionality of each operation are described in detail. The robustness of DKEMA and its performance are assessed in Section 5 and Section 7, respectively. Finally, in Section 8, the conclusion and future directions are presented.

2 Background and Preliminaries

In this section, the AES block cipher that can be employed with authentication operation mode such as CMAC and GMAC will be described. Then, we describe the main GPU architecture characteristics. After

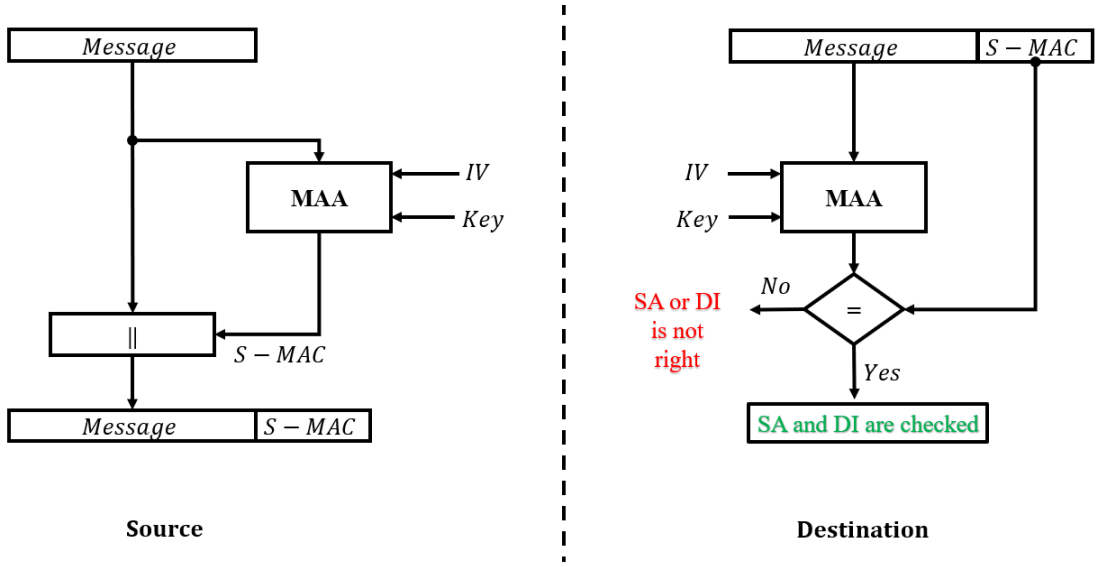


Fig. 2: Message authentication algorithm: signature and verification. Besides, a cipher-based MAA can be used instead of Keyed hash function.

this, the different AES implementations on GPU will be presented. Finally, we review the recent dynamic-key approach used to construct lightweight cipher schemes and we present our threat model. Let us indicate that the well-known MAA structure (Merkle-Damgard) is illustrated in Fig. 2. It should be noted that all used notations and abbreviations are included in Table 1 and 2, respectively.

2.1 AES

AES [19] is a block cipher that processes data in blocks of 128 bits (16 bytes), and it uses keys of size 128, 192 and 256 bits. The design of AES depends on the SPN principle. It includes a *round* function that is iterated for r times depending on the secret key size. The number of rounds, r, is equal to 10, 12, and 14 for a secret key of size 128, 192, and 256 bits, respectively. This round function consists of four operations, except for the last iteration. These operations can be described as follows:

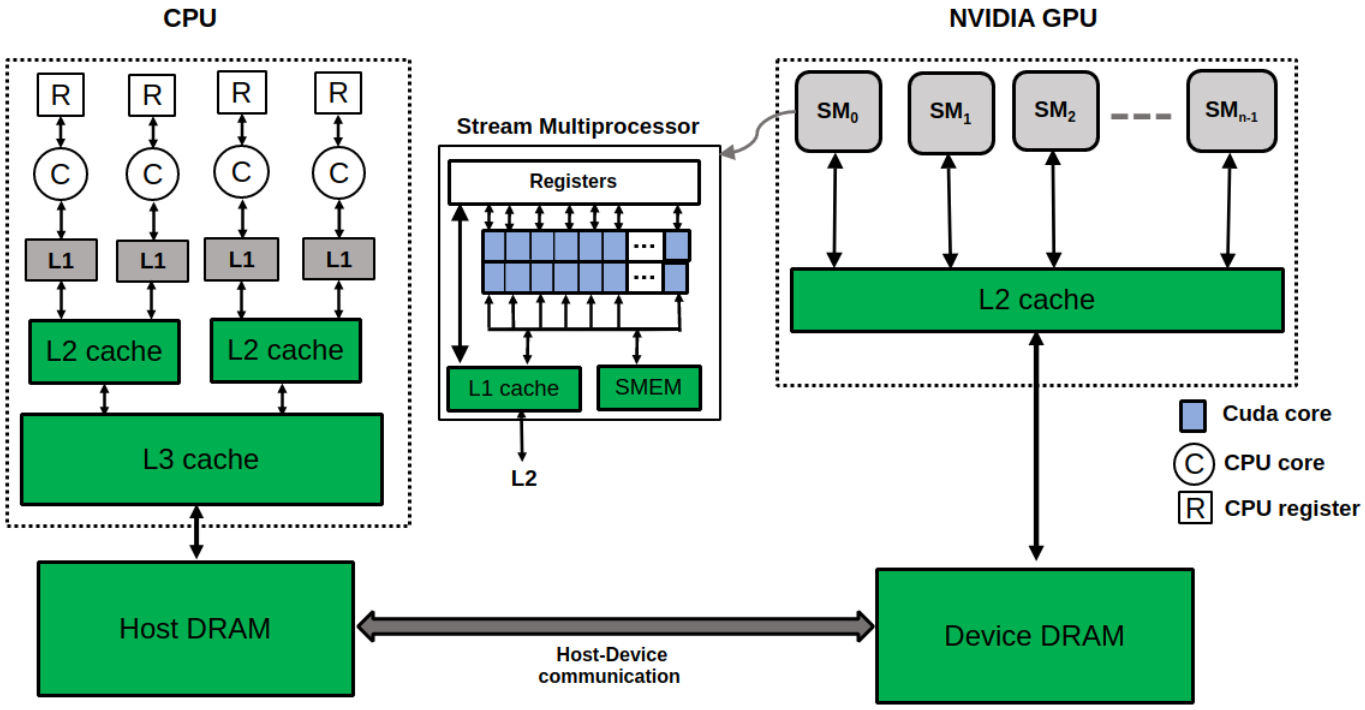
- RoundKeyAddition: it mixes the plain input block with the specific round key.
- ByteSubstitution: this operation employs a substitution table, S-Box, to ensure the confusion property.
- ShiftRows and MixColumn operations are used to ensure the diffusion property. Note that the MixColumn operation is eliminated in the last round.

2.2 GPU Architecture

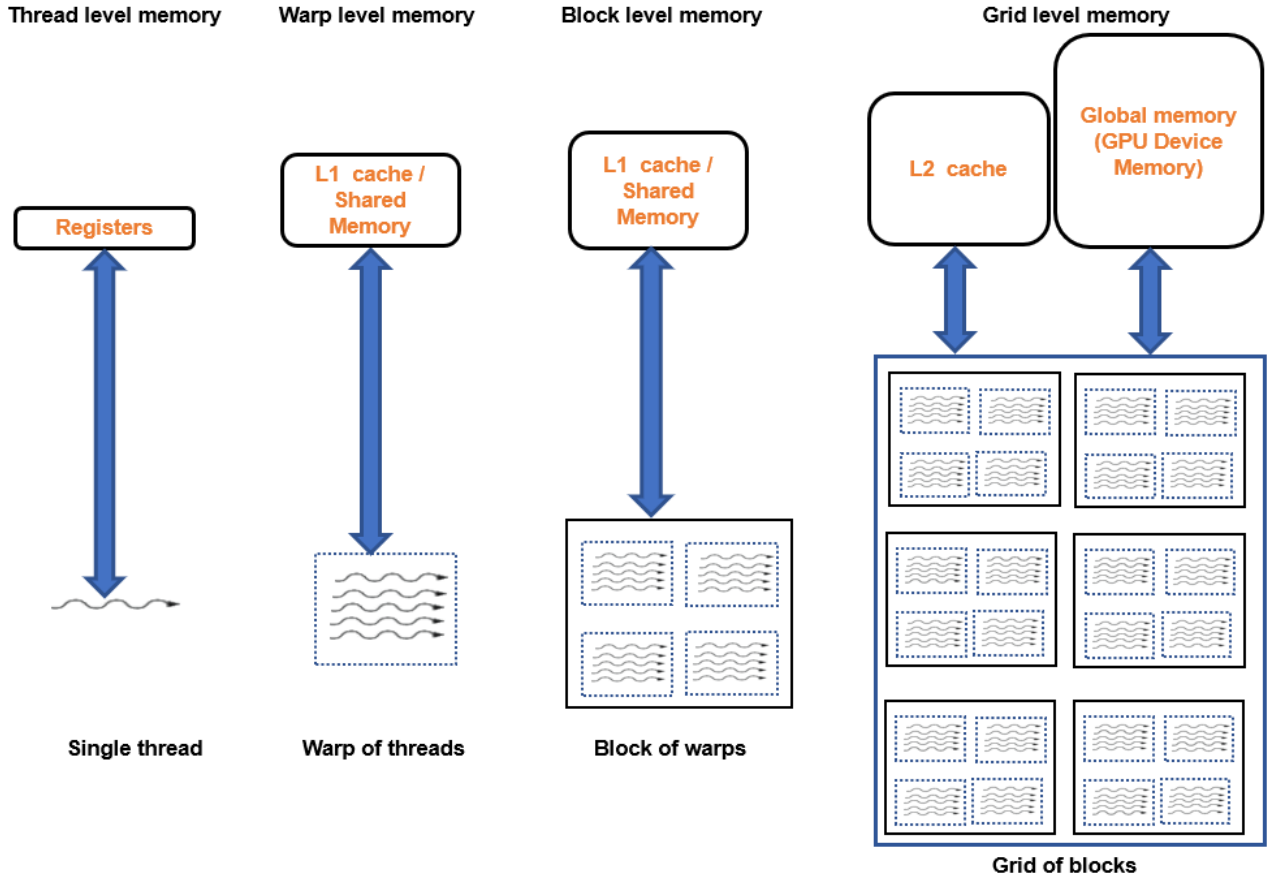
In this work, we are referring the Nvidia GPU architecture terminology, as this architecture is mainly used for most of GPU cards. The GPU architecture presents a high number of concurrent threads to maximize the efficiency (throughput) compared to the CPU architecture. There are hundreds and thousands of computer cores within a GPU. The GPU hardware is built to run several threads in parallel, even if the bottleneck is the memory access itself. Users can use a number of threads that surpass the number of cores to benefit from GPU' s computing capacity. In fact, while some threads are waiting for data from the registry ('inactive' threads), other threads can execute their instructions ('active' threads). Specifically, Nvidia GPUs consist of Streaming Multiprocessors (SMs), as illustrated in Fig. 3-a, that vary in number for different GPU types. SMs consist usually of 32 cores components that can only perform one instruction at a time. Each SM can execute multiple warps at the same time. A warp is a collection of 32 threads, that can be executed simultaneously on an SM.

Typically, several types of memory exist in a GPU: a global memory that is the slowest one, a cache memory, a texture memory, a shared memory, a local memory, and a small number of fast access registers. Thus, the memory management in GPUs is indeed critical and essential. Fig. 3-b presents the memory hierarchy in Nvidia GPU devices as follows:

- Registers: these are thread-private, which implies that thread-specific registers are not accessible to



(a) CPU-GPU functionality



(b) Nvidia memory hierarchical

Fig. 3: CPU and Nvidia GPU architecture (a) and Nvidia GPU memory hierarchical level (b)

Table 1: Table of notations

Notation	Definition
K	Secret key
N_o	Nonce
DK	Dynamic Key
K_{S1} and K_{S2}	First and second substitution sub-keys
K_rg	The seed generation sub-key
S_1 and S_2	The first and the second produced dynamic substitution tables
$Seed_i$	The i^{th} seed
N	Number of possible seeds
Qg	Word precision that can be 32, 64 or 128.
l	Number of bytes of the input message
nb	Number of blocks in an input message
M	The plain-text message
$\|M\|$	The length of plain-text message
b_i	The i^{th} block of plain message
$sb_{i,\ j}$	The j^{th} sub-block of the i^{th} message block
c_i	The i^{th} compressed block
$sc_{i,\ j}$	The j^{th} sub-block of the i^{th} compressed block
n	The number of possible active threads
Nw	The number of words per block (processed by a thread)
Tb	Number of bits per block and it is equal to $Nw \times Qg$
CTR	represent a word counter that uses for each block (thread) and it is depend of the number of block i and number of sub-block j
$x\%y$	produces the remainder of an integer division between x and y
$x <<< y$	rotated left of x (bits representation) for y times

Table 2: Table of abbreviations

Abbreviation	Definition
DKEMA	Dynamic Key-Dependent Efficient Message Authentication
DI	Data Integrity
SA	Source Authentication
AES	Advanced Encryption Standard
DES	Data Encryption Standard
SHA	Secure Hash Algorithm
MAA	Message Authentication Algorithm
MAC	Message Authentication Code
GMAC	Galois Message Authentication Code
CMAC	Ciphered Message Authentication Code
HMAC	Hash Message Authentication Code
FN	Feistel Network
SPN	Substitution-Permutation Network
PRNG	Pseudo Random Number Generator
KSA	Key Setup Algorithm of RC4
CF	Compression Function
RF	Round Function
MD	Merkle Damgrad
GPU	Graphic Processing Unit
SMEM	Shared Memory
CG	Cooperative Groups

other threads. The compiler makes judgments on register use.

– L1/Shared memory (SMEM): each SM includes an on-chip memory that may be utilized as an L1 cache or a shared memory. The L1 cache is privately accessible by threads. While all threads in a Cuda block can use the shared memory simultaneously.
– L2 cache: it is shared among all SMs and it is accessible to every thread throughout each Cuda block.
– Global memory: this refers to the size of the GPU's frame buffer and the DRAM included within the GPU.

2.3 Optimized AES GPU Implementation

Based on GPU, the improvement of the AES has been the subject of several research works [20], especially on the implementation techniques used for the AES kernel function. In the following, the most recent work to improve AES on GPU platforms are described, where the enhancement can be achieved at two main levels:

1. Kernel optimization: these techniques try to minimize the kernel time released inside the GPU to reach maximum throughput. These optimizations can be done at two levels:
   - GPU memory optimization: to enhance the kernel performance of AES, finding the best memory configuration for input and intermediate data is very vital. With AES, three different data with different sizes should be stored inside the GPU memory, including:
     - Input data: existing works recommend to store it in the global memory. We recommend, if AES is applied with the counter mode (as a stream cipher), to not transfer the input data to the GPU, towards reducing the data time transfer. "Exclusive or" between message and produced keystream can be applied on the CPU.
     - Cryptographic primitives: substitution/permutation tables, and round Keys should be stored on the shared memory [21, 22], which is the fastest one compared to other memories such as the global one.
   - Parallel granularity and Cuda thread configuration: detecting the optimal number of blocks per grid and the number of threads per grid block forms the thread granularity strategy. Thus, allowing a single thread to process more than one input block will result in reducing the required number of threads. This can improve the throughput, but will increase the load of functions performed by each thread, which will reduce the performance of the AES kernel. Therefore, the selection of the number of blocks and the number of threads per block depends on the desired performance of the AES kernel, which depends also on the characteristics of the employed GPU. Besides, [21, 22] recommends to encrypt two blocks per thread to achieve best performance for AES (Geforce 1080 GPU).
2. Data transfer optimization between CPU-GPU: the focus of these techniques is to reduce (minimize) the data transfer time between CPU and GPU and vice versa to maximize the throughput and consequently the speedup. GPU-CPU data transfer is essential for any GPU implementation. This is the case when the data should move from CPU memories to GPU memories or vice versa. Thus, the time required for this transfer via the PCI express bus must be taken into consideration and is known as the "data transfer time". It affects the real throughput of the GPU implementation. In general, the data transfer time is longer than the execution time of the kernel itself. Recent work recommends using the strategy of streaming technique and unified memory to overcome this overhead [23] .

2.4 Dynamic Key-Dependent Cryptographic Algorithms

Traditional symmetric cryptographic algorithms, such as AES, have a static structure: the substitution or permutation table consists of constant values, and the multiplication of columns is carried out always with the same diffusion matrix. Throughout the encryption/decryption processes, the confusion and diffusion primitives are kept fixed. The secret key is used only to produce a set of round keys.

The dynamic cryptographic approach consists of the use of a secret key with a nonce to produce a dynamic key used to produce dynamic cryptographic primitives, which can be changed for each new message or set of messages (depending of the configuration). Therefore, dynamic-key based cryptographic algorithms use variable substitution and/or diffusion primitives, which increases their level of robustness against security attacks.

Recently, there is a tendency to rely more on the dynamic-based approach to construct efficient and lightweight security schemes [15, 16, 24, 25]. In addition, the use of the dynamic operation mode was presented in [26], where blocks are selected in a pseudo-random manner for each input message. All dynamic cryptographic algorithms were designed to be realized on limited devices and not on a powerful device such as GPU. Differently, in this paper, dynamic cryptographic primitives are used to ensure the confusion and diffusion properties in DKEMA scheme designed according the GPU characteristics.

2.5 Threat Model

In this work, we consider two types of data integrity attacks or errors, as described in the following.

- A malicious attacker could modify the data without being able to generate the correct MAC value of the modified message. In this case, any MAA can detect the malicious data manipulation.

– A more dangerous attack is when a malicious attacker tries to deduce, by cryptanalysis, the secret key to modify the data and regenerate a new valid MAC value.

After reviewing the main concepts needed to understand the rest of this paper, we describe in the next section the dynamic key and primitives construction.

3 Dynamic Key Derivation Function and Cryptographic Primitives Construction

The proposed dynamic key generation and cryptographic primitives construction are illustrated in Fig. 4. The required cryptographic primitives (seeds and substitution boxes) are dynamic, being function of the dynamic key. The secret session key K is mixed with a nonce (unique for every input message) N_o to create a dynamic secret block $O = K \oplus N_o$, which will be hashed by using a secure cryptographic hash function to produce the dynamic key (DK).

In this paper, SHA-512 is used as a hash function, given that it ensures high collision resistance and can be employed for small messages (blocks of 512 bits length). This results into a different DK for each new input message. The dynamicity of DKEMA makes it more resistant against strong message authentication attacks. Then, the dynamic key DK, that has a length equals to 512 bits, is divided into three sub-keys $\{K_{S1},\ K_{S2}\ \ K_{rg}\}$.

The size of each substitution sub-key is equal to 16 bytes (128 bits), while the size of the seeds generation sub-key K_{rg} is equal to 256 bits (32 bytes). K_{S1} and K_{S2} are used to produce two different substitution tables (S_1 and S_2), while K_{rg} is used as a seed for a PRNG to produce N different seeds. In the following, the used sub-keys are described in detail.

– **Substitution sub-keys** K_S1 and K_S2: they represent the first and the second 16 most significant bytes, that are used to produce the substitution tables S_1 and S_2, respectively. Let us indicate that the substitution process is done at the byte level in contrast to other operations that are realized at the word level (64 bits). Therefore, the length of each substitution table is 256 bytes. The employed technique to produce dynamic substitution tables is described in [13]. This method employs the Key Setup Algorithm (KSA) of RC4. First, KSA-RC4 is iterated with K_S1 to produce the first substitution table S_1. Then, it is iterated with K_S2 to produce the second substitution table S_2.

– **Seeds sub-key** K_{rg}: it represents the first 32 least significant bytes of DK and it is used to produce N seeds. In this step, RC4 is iterated for $\frac{N \times Qg}{8}$ times to generate N different seeds, where N represents the possible number of threads, and Qg represents the precision of the employed generator, which can be equal to 32, 64 or 128. The output key-stream is reshaped to form a matrix of size $N \times \frac{Qg}{8}$, where each element is a byte. Each row of this matrix is converted to a binary sequence that has a length equals to Qg bits. This binary sequence forms a word of Qg bits and represents one of the N seeds. Any repeated row (seed) is eliminated from this list and RC4 is re-iterated to produce a new seed in this case. Note that $RC4$ is iterated to produce dynamic cryptographic primitives to achieve a high level of security, where N is selected according to the desired security level.

These steps guarantee a high level of sensitivity, where any tiny change in the dynamic key would result into a completely different set of cryptographic primitives, as shown in Section 5.2. The parameters' derivation is illustrated in Fig. 4. After detailing the dynamic key generation and the corresponding cryptographic primitives construction, in the next section, we present DKEMA scheme.

4 Proposed Message Authentication Algorithm

This section describes DKEMA, being a keyed hash function that uses a simple Compression Function (CF), that requires a single iteration of the proposed Round Function (RF) in contrast to existing symmetric MAAs, such as HMAC and CMAC, that use multi-rounds and mutli-operations cryptographic algorithms.

Besides, Fig. 5 presents the high level scheme of DKEMA solution. For each new input message, a dynamic key and a set of cryptographic primitives are generated at the CPU level. After this, the CPU transfers the message M and the required cryptographic primitives CM to the GPU. Then, the GPU implementation of MAA will be iterated on M using the corresponding CM (seeds and substitution tables). Let us indicate that the GPU implementation of DKEMA is parallel at the grid level (details are shown later in Fig. 9). This means that a part of the message (a set of blocks) will be selected. Furthermore, for each thread on each grid, an input block will be compressed by using the proposed single round function. Then, the compressed blocks of each grid will be mixed ("exclusive or") together and the final result will be compressed another time. In parallel,

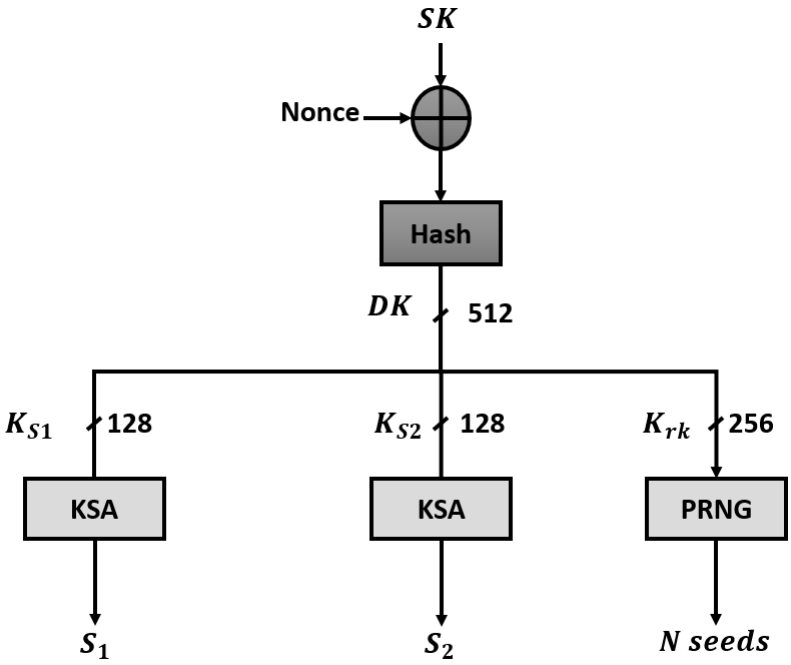


Fig. 4: The proposed Key derivation function and its corresponding construction of cipher primitives

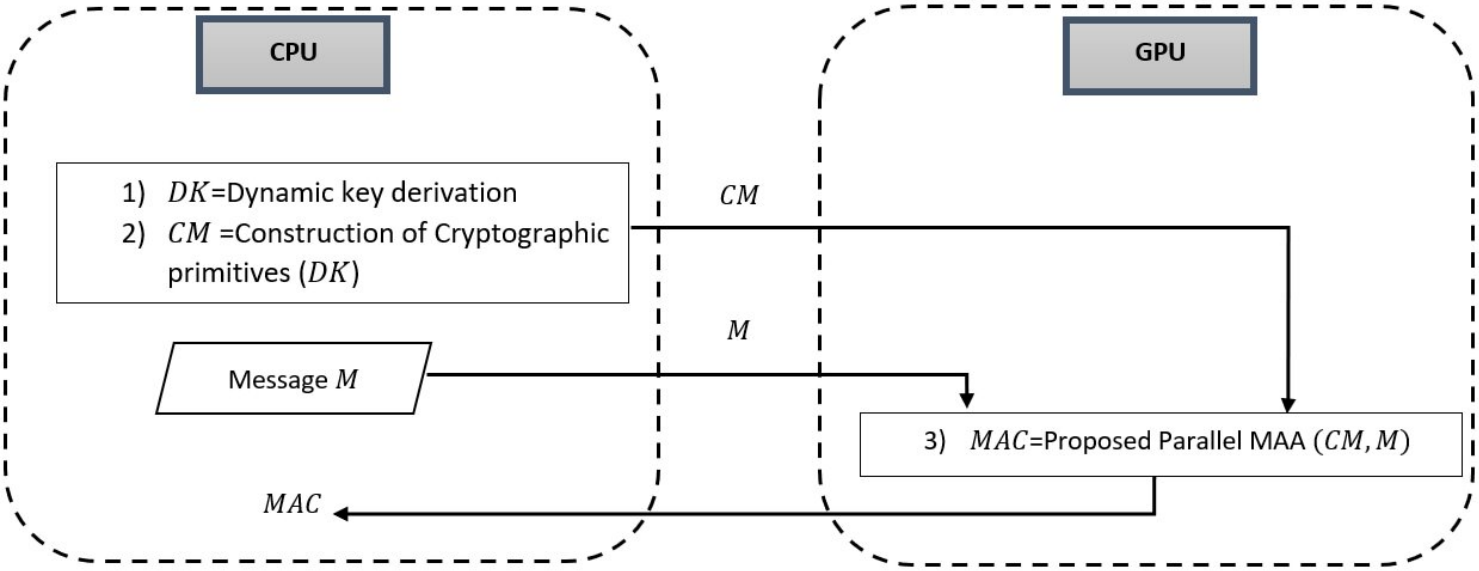


Fig. 5: High level scheme of DKEMA communication between CPU and GPU.

all grid compressed blocks will be mixed another time and then compressed to produce the MAC of M. After this, the MAC will be transferred to CPU.

4.1 Basic Concepts

The main properties of the proposed solution are:

1. High-security level
2. Reduced computational complexity
3. Simple and parallel hardware and software implementations

To gain these properties, DKEMA scheme is based on 3 main concepts:

- **Parallel Computing:** this algorithm is designed to be parallelized. All the threads are independent of each other and they could be all executed in parallel (see Fig. 6), even if it is not possible to schedule all of them at the same time. The SMs contain each 32 synchronized threads and a shared memory. Hence, the same operation (compression function kernel) can be applied on these synchronized threads but with different inputs. Thus, the 32 threads are iterated to perform the compression function.
- **Flexible Structure:** the structure of DKEMA allows for any size of input blocks (256, 512 and 1024). Moreover, any PRNG that exhibits good performance and satisfies the randomness properties could be used. For example, in this paper, we use "splitmix64", which was selected due to its simplicity and efficiency for a word precision equals to 64 bits.
- **Efficient & Lightweight Combination of Cryptographic Primitives:** the selected PRNG (according to word precision) is combined efficiently with substitution and binary diffusion operations to produce the compressed blocks. The proposed technique benefits from the shared GPU memory (substitution tables and compressed blocks). On the other hand, DKEMA can be adapted to be a stream cipher (where each thread produce a keystream block). The produced key-stream has a stable randomness and uniformity degree in addition to high periodicity.

4.2 Proposed Parallel Authentication Algorithm

The originality of this work is that DKEMA is designed to suit better the GPU characteristics, compared to existing MAAs. Furthermore, the proposed approach satisfies the message and key avalanche properties since for each new input message, a new dynamic key and new cryptographic primitives are generated. Moreover, the input message can be applied to any data type such as image, video, text, etc.

DKEMA requires a single round of the proposed compression function (CF), authenticating Nw sub-blocks simultaneously in one iteration. DKEMA presents a parallel structure different to the Merkle Damgrad (MD) structure, that employs the chaining operation mode. Instead, DKEMA applies the compression function in parallel, as shown in Fig. 6 and Algorithm 1.

Hence, after generating the cryptographic primitives, the input message M is padded, if necessary, to have a length divisible by the size of the data block, $Tb = Nw \times Qg$, which could be set to 128, 256, 512, or 1024 bits. Let us indicate that Tb should be multiple of Qg (Nw words and each ones has Qg bits). Besides, Qg can be 32, 64 or 128. Then, M is divided into nb blocks ($b_1, b_2, \ldots, b_{nb}$), where $nb = \frac{|M|}{Tb}$. The choice of Tb is related to the required security level. Afterwards, each data block b_i, $1 \leq i \leq nb$, goes through the compression function, f (see Fig. 7 and Algorithm 2). After processing all blocks, a final mixing (exclusive or) operation between all compressed blocks ($c_1, c_2, \ldots, cnb$) is done. Then, the mixed block is compressed using CF, where its corresponding output represents the MAC value. Thus, the MAC can be calculated according to the following equation:

$$\begin{aligned} c_i &= CF(b_i) \qquad i = 1, 2, 3, \ldots, nb \\ MAC &= CF(\oplus_{i=1}^{nb} c_i) \end{aligned} \tag{1}$$

In the following, DKEMA Compression Function (CF), and Round Function (RF) are described.

4.2.1 Compression Function

The compression function consists of dividing the input message blocks into Nw words, ($sb_{i,\,1}, \ldots, sb_{i,\,Nw}$), of Qg bits length each. In fact, the compression function is also designed to be realized in parallel, where the sub-blocks can be processed in parallel, as illustrated in Fig. 7, by using the proposed round function that will be iterated once.
After processing all sub-blocks, a binary diffusion mixing operation is applied on the processed sub-blocks, which results into the compressed block ($c_i = c_{i,\,1}, \ldots, c_{i,\,Nw}$). The 4× 4, 8× 8, 16×16, and 32×32 binary diffusion matrices, selected in [27], can be used for Nw= 4, 8, 16, and 32, respectively.

4.2.2 Round Function

The proposed round function is applied for each sub-block (word level) and it uses two substitution tables S_1 and S_2, and a seed which is dynamically chosen based on the block and sub-block number from a set of generated seeds (N), as shown in Fig. 8.

At each iteration of this round function (see Fig. 8), an input j^{th} sub-block ($sb_{i,j}$) of the i^{th} block is processed according to the following equation (Eq. 2):

$$\begin{aligned} x &= sb_{i,\,j} \oplus Seed_{(i+j\%N)+1} \oplus CTR_{i,\,j} \\ temp &= ROTL(x,\ CTR_{i,\,j}\%Qg) \\ sc_{i,\,j} &= Splitmix64(Substitution(temp,\ S_1,\ S_2)) \end{aligned} \tag{2}$$

In details, the j^{th} sub-block (word of Qg bits) of the i^{th} block is processed by applying these five steps:

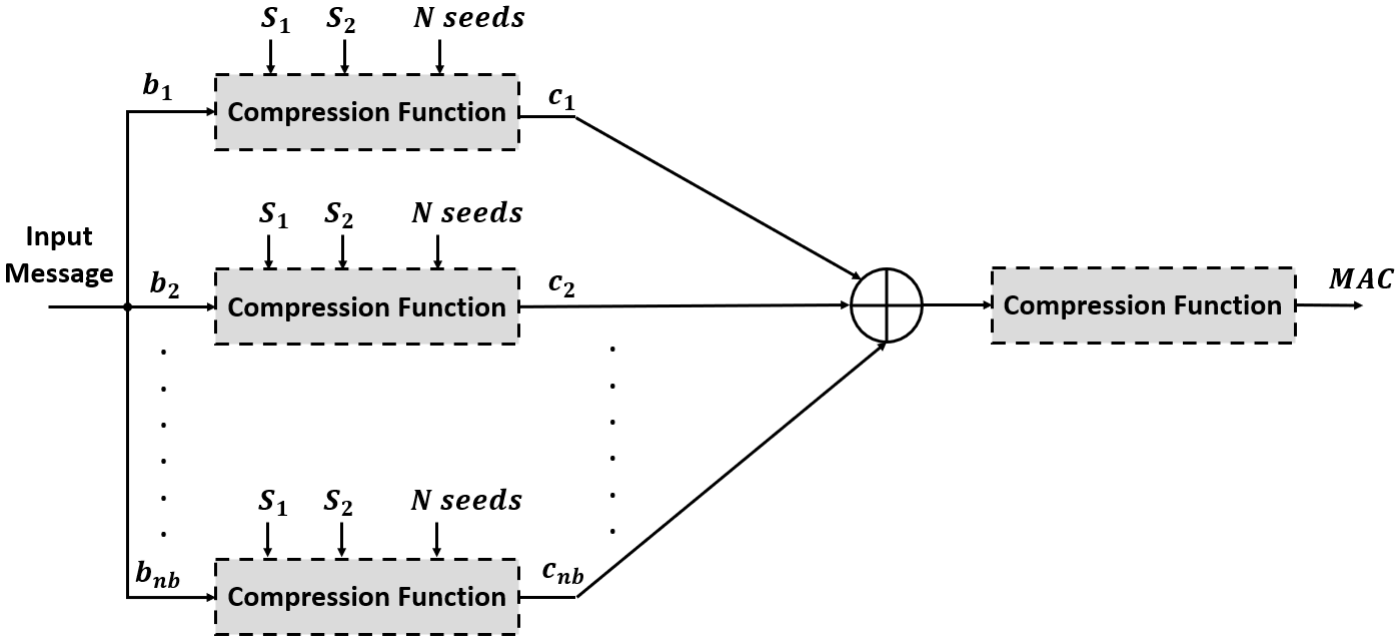


Fig. 6: Scheme of DKEMA (S_1 and S_2 can be the same substitution table S).

Algorithm 1 Proposed MAA (DKEMA)

Input:
Message M;
Substitution tables (S_1 and S_2);
N seeds: $Seeds = \{Seed_1,\ Seed_2,\ \dots\ ,\ Seed_N\}$
Output: MAC

procedure DKEMA($M,\ S_1,\ S_2,\ Seeds,\ Nw$)
  $Tb \leftarrow Nw \times Qg$ ▷ Number of bits per block
  $M \leftarrow Padding(M,\ Tb)$ ▷ Padding the last block if it is necessary
  $nb \leftarrow \frac{|M|}{Tb}$ ▷ Calculate the number of blocks nb per message
  ▷ Divide message M into nb blocks, each blocks consists of Nw words and each one has Qg bits.
  $\{b_1, b_2, \dots, b_{nb}\} \leftarrow DivideMessageToBlocks(M)$
  $MAC \leftarrow Zeros(1,\ Nw)$ ▷ Initialize a MAC block of Nw zeros words.
  ▷ In GPU implementation, each block is compressed independently at each thread
  for $i = 1 \rightarrow nb$ **do**
    $Y \leftarrow ProposedCompressionFunction(b_i,\ S_1,\ S_2,\ Seeds,\ Nw)$ ▷ See Fig. 7 and Algorithm 2
    $MAC \leftarrow MAC \oplus Y$
  Return: MAC

Algorithm 2 CF of DKEMA

Input:
i^{th} input block b_i;
Substitution tables (S_1 and S_2);
N seeds
Output: i^{th} compressed block c_i

procedure ProposedCompressionFunction($b_i,\ S_1,\ S_2,\ Seeds,\ Nw$)
  ▷ Divide b_i block into Nw sub-blocks and each one consists of Qg bits precision.
  $sb_{i,\,1},\ sb_{i,\,2},\ \dots,\ sb_{i,\,Nw} \leftarrow DivideBlockToWords(b_i)$
  $c_i \leftarrow Zeros(1,\ Nw)$
  for $j = 1 \rightarrow Nw$ **do**
    $c_{i,\,j} \leftarrow ProposedRoundFunction(sb_{i,\,j},\ S_1,\ S_2, Seeds)$ ▷ Illustrated in Fig. 8 and presented in Eq. 2
    $temp \leftarrow temp \oplus c_{i,\,j}$
  $temp \leftarrow ProposedRoundFunction(temp,\ S_1,\ S_2, Seeds)$
  $c_i \leftarrow BinaryDiffusion(temp,\ c_i)$
  Return: The i^{th} compressed block c_i, which consists of Nw words.

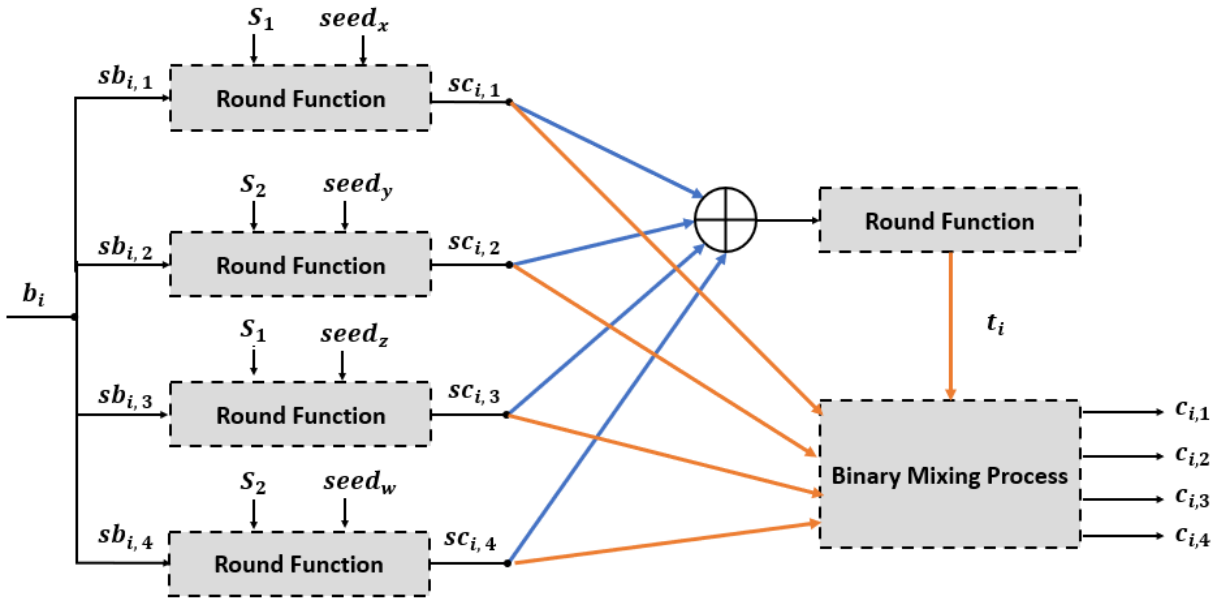


Fig. 7: Scheme of the proposed lightweight compression function for the i^{th} block (thread) with Nw =4. S_1 and S_2 can be the same substitution table S (depending on the configuration).

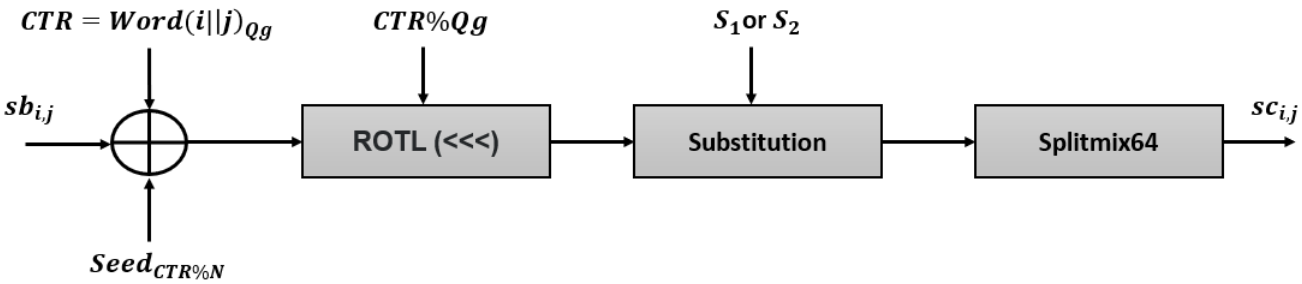


Fig. 8: Scheme of the proposed lightweight round function for the j^{th} sub-block of the i^{th} block (thread).

1. Calculate a counter CTR by concatenating the block number i and the sub-block number j. The counter will be converted to a word of Qg bits.
2. Mix $sb_{i,j}$ with the selected seed (CTR modulo N) and CTR using the logical "exclusive OR".
3. Then, a bit-wise left-rotating is applied on the output for $CTR\%Qg$ positions.
4. In this step, the substitution process is realized but at the byte level. This means that the output word is converted to $\frac{Qg}{8}$ bytes. After this, these bytes are substituted by using two substitution tables (S_1 and S_2). The first substitution table is used to substitute the bytes with even indexes, while the second substitution table S_2 is used to substitute the bytes with odd indexes. For example S_1 is used to substitute the bytes with index i with $i = 2,\ 4,\ 6, \ldots$ and S_2 is used to substitute the bytes with index j with $j = 1,\ 3,\ 5, \ldots$. Then, the substituted bytes are re-converted to word of Qg bits length.
5. Iterate the selected PRNG once. The output of splitmix64 is a word of 64 bits. Here, splitmix64 is used as a proof of concept. Otherwise, any secure and efficient PRNG can be used instead of splitmix64 if the required size of word is 128 or 256 bits. Let us indicate that this step is introduced to achieve better message and key avalanche effects.

Then, the compressed sub-blocks are "exclusive or"ed together to produce the word $r1 = sc_1 \oplus sc_2 \oplus sc_3 \oplus sc_4$. Then, $r1$ is compressed by using the proposed round function to produce t. Then, a binary diffusion process is applied between t and sc_1, sc_2, sc_3, sc_4, as shown in the following for the case of $Nw = 4$, to obtain the

compressed sub-blocks.

$$\begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} sc_1 \oplus t \\ sc_2 \oplus t \\ sc_3 \oplus t \\ sc_4 \oplus t \end{bmatrix}$$

where $\oplus$ represents the XOR operation.

In the following, the GPU implementation of DKEMA is described in detail.

4.3 GPU implementation of DKEMA

DKEMA relies on a single pass reduction using multi-block Cooperative Groups (CG). Nvidia introduced this new feature in Cuda 9.0. CG allow scalable cooperation among groups of threads. In the following, we detail the way CG are used to build an efficient implementation of DKEMA on GPU. It should be noted that we present only the version with 256 bits input block.

Besides, Fig. 9 presents the GPU implementation of DKEMA, which mixes all compressed blocks at the grid level first, then it mixes the obtained MAC blocks of the first step. Afterwards, the obtained result is compressed to produce the final MAC. The different steps of the GPU implementation of DKEMA, presented in Algorithm 4, are listed and described as follows:

- For each block, two parts of the shared memory are used. The size of each part is equal to the number of threads per block.
- In lines 11 and 12, shared arrays *sdata* and *sdata2* are initialized.
- In lines 13 to 15, *Sbox8* is copied into the shared memory array *ssbox*. It is essential to use the shared memory for this variable.
- The first loop starts in line 17. This loop iterates all elements of the input data stored in the global variable *g_idata*. Here the CG is used to let each thread of each block working on a different part of this array. For each new iteration of this loop, the increment is the size of the grid, i.e. the total number of used threads (number of blocks multiplied by the number of threads per block).
- Line 18, the input data is read and xored with DK, which represents the dynamic key. As the dynamic key is of 512 bytes length, it is first cast to 64 bits. This means only 64 numbers of 64 bits are available. Then, the dynamic key (casted to 64 bits) is xored with the loop iteration i.
- Line 19, the result of the last step is left rotated (using 64 bits) *ssbox[i&255]&63* times.
- Line 20, the kernel *apply_sub_and_prng* is called. This kernel is presented in Algorithm 5 and is explained in the following. Then, a substitution is applied on the last result and the PRNG *splitmix64* is called in order to produce a new 64 bits number.
- Line 21, the result of the substitution process is stored in the second shared memory array *sdata2*.
- Given that MAA works with an input block of 256 bits length, only 1 thread over 4 executes the block, starting line 23. Due to the fact that threads on the same warp are necessarily synchronized, all groups of 4 consecutive threads are synchronized (so it is useless to use an explicit synchronization). Line 23 consists of xoring the 4 previous results computed by the group of 4 threads. Then, the kernel *apply_sub_and_prng* is called to do the same as in Line 20.
- Lines 26 to 29, each obtained substituted word (Line 21) is xored with the computed word value obtained in Line 23. This step is done to reduce the required number of xor (optimization). The result of this step is stored in the first shared memory. Consequently, after this step, all the threads have computed the compression function (illustrated in Fig. 7).
- After the first loop, a synchronization of all the threads in the same block is required (line 32). Then, the kernel *reduceBlock* is called. The description of the *reduceBlock* kernel, presented in Algorithm 6, is detailed in the following.
- After this kernel, only the first thread of each block executes the lines 35 to 38. These lines are used to put the 4 numbers of 64 bits corresponding to the Message Authentication Code (MAC) of the block related text in the global memory, at the corresponding location based on the block id.
- Line 40, the whole grid is synchronized. Then, only the first thread of the whole grid executes the second loop. This loop aims at xoring all the 4 numbers of 64 bits of all the blocks together. Consequently, the message authentication of the whole text is computed.
- Lines 48 to 53, presenting the same mechanism described in Lines 23 to 29, this step consists of the final round applying the *apply_sub_and_prng* kernel in order to obtain the final MAC.

Algorithm 6 aims to mix ("exclusive or") the compressed blocks. The principle of this kernel is to xor all 4 numbers of 64 bits. The first loop (line 8) is executed on all 32 threads in parallel. 32 corresponds to the size of a warp. For each iteration of the loop, the number of the xor operations is divided by 2. Due to the fact that some threads do no execute the condition line 9, a synchronization is required (line 14). In lines 10 to 12,

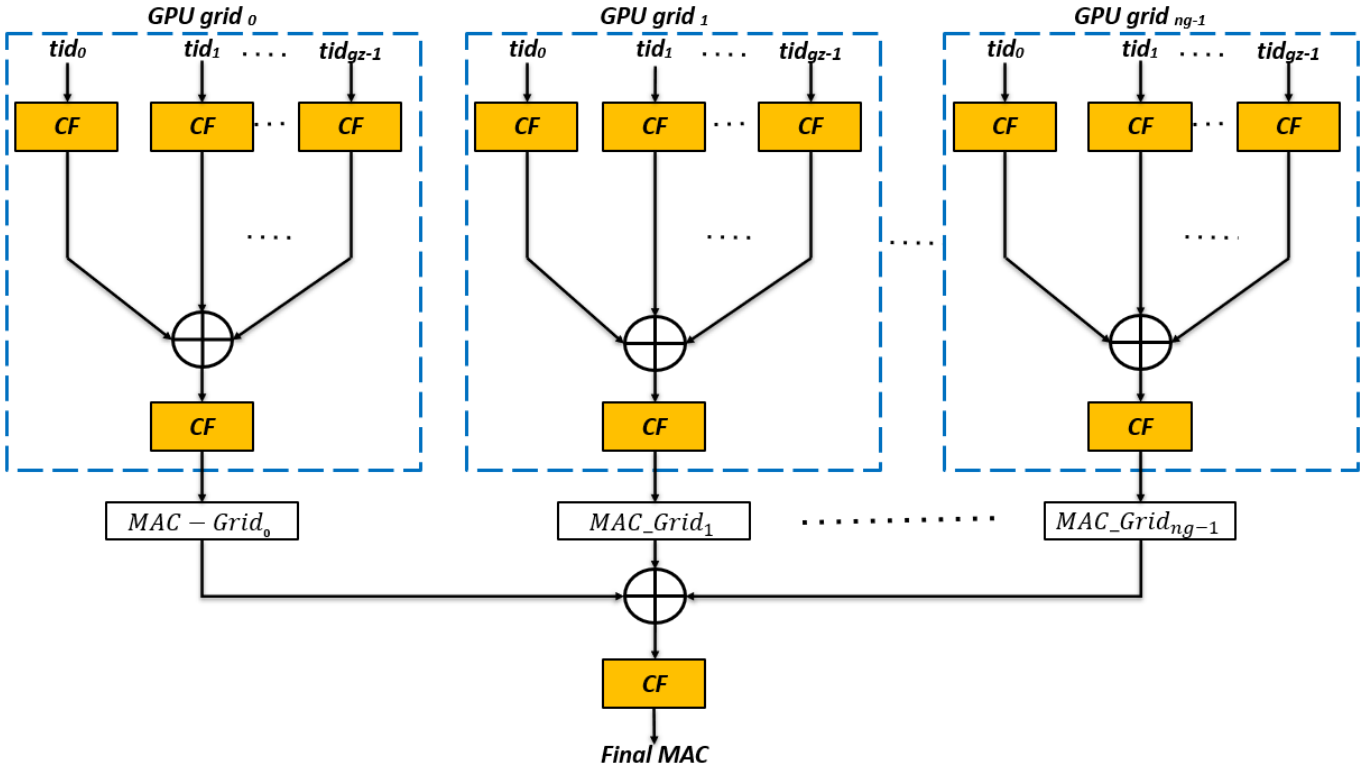


Fig. 9: Scheme of DKEMA using CF

we compute the xor between the element *tid* and *tid+i*. In line 16, all the threads of the block are synchronized. Then, only the first four threads of this block is used to make the xor between all the 32 elements.

Algorithm 5 is used to apply the Sbox on a 64 bits integer. Line 2 converts the 64 bits pointer into a 8 bits pointer. Then, the 8 bytes are substituted by using the substitution table on each byte. Finally, the *Splitmix64* PRNG is applied on the obtained result.
After detailing the implementation of DKEMA on GPU (Nvidia Tesla V100 and Tesla A100), it should be noted that DKEMA can be adapted to any other Nvidia GPU. Thus, there is no need to any modification or special requirements with less performed GPUs.

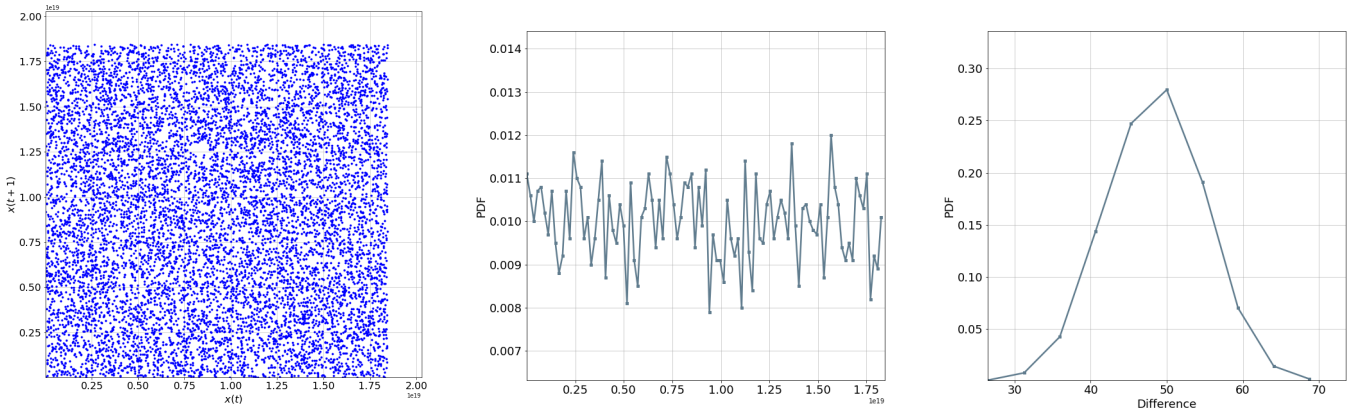
Algorithm 3 splitmix64 code

```
1  __device__ inline
2  ulong splitmix64(ulong x)
3  {
4          x+= 0x9e3779b97f4a7c15;
5          x^= (x >> 30) * 0xbf58476d1ce4e5b9;
6          x^ = (x >> 27) * 0x94d049bb133111eb;
7          x ^= (x >> 31);
8          return x;
9  }
```

In the following, the SplitMix-64 pseudo-random generator, used in DKEMA, is described.

4.4 SplitMix-64

The "SplitMix-64" algorithm [28], presented in Algorithm 3, is a fast splittable PRNG. It has a 64 bit state as input and output. By applying two standard statistical randomness test suites (DieHarder and TestU01), it has been shown that "SplitMix-64" is efficient for time-critical applications. The Splitmix64 PRNG presents numerous advantages including a low execution time as well as a simple implementation. However, it is not advocated for cryptographic or security applications, because the generated sequences of pseudo-random values are predictable since the mixing functions are easily invertible, and two successive outputs suffice to reconstruct the internal state. In the proposed solution, SplitMix-64 is used with dynamic seeds in addition to a non-linear operation (confusion). In this case, SplitMix-64 uses a carefully handcrafted shift/rotate-based linear transformation. This ensures a significant reduction in latency and the corresponding resources with a high level of randomness. Therefore, DKEMA scheme uses a high number of threads, and for each thread, a Splitmix64 PRNG is iterated with a different seed.



(a) (b) (c)

Fig. 10: Splitmix64 Recurrence (a) and PDF (b) for a random seed. In addition, the PDF of seed sensitivity (c) of Splitmix64 PRNG for a modified LSB bit and for 1000 times.

Three metrics were evaluated for the Splitmix64, which are the recurrence, uniformity and the seed sensitivity (described in Section 5.1). The seed sensitivity tests were carried out on a set of $1,000$ random seeds (see Fig. 10-c), where for each time, the Least significant bit of each seed is flipped. Let us indicate that the recurrence and uniformity results (see Fig. 10 a-b) are obtained by iterating Splitmix64 with a random seed. Similar results can be obtained for any seed.

After describing DKEMA and its GPU-based implementation details, in the next section, we include the experimental results that show the efficiency and robustness of the proposed solution.

5 Security Analysis

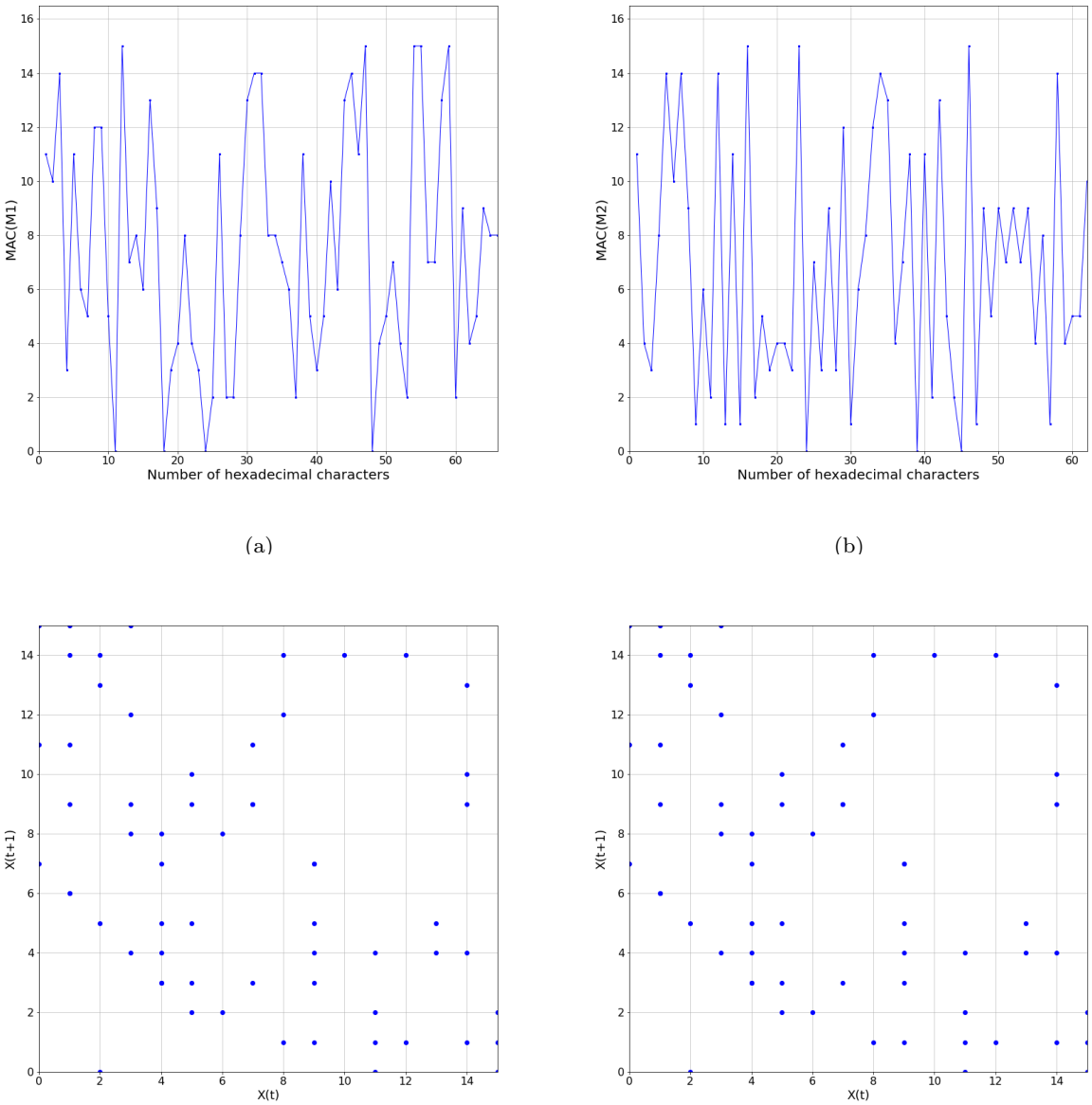
To analyze the security of DKEMA, several security tests were performed including randomness, uniformity and sensitivity tests described in [13, 17]. Note that DKEMA is flexible in terms of the Tb size. For the security analysis, we show the results of the randomness and uniformity tests with Tb equals to 256. Also, a comparison is conducted between the proposed solution and existing MAAs like CMAC and HMAC.

5.1 Randomness and Uniformity Tests

The randomness and uniformity of the obtained MAC values are evaluated in function of the message and the used cryptographic primitives. In this test, the MAC values are evaluated by simulating DKEMA with 1,000 different input messages and dynamic keys.

The MAC values are computed, each of size 32 bytes, for the messages using DKEMA. Then, each MAC value is represented in hexadecimal format, which yields 64 hex digits. As visual results, two MAC values are shown in Fig. 11, in addition to their recurrence results. The distribution of the obtained MAC values are illustrated in Fig. 12, showing that the hex values of (0 to 15) are uniformly spread.

On the other hand, we conducted a test to compute the number of unique bytes within each of the obtained MAC values. Table 3 includes the results corresponding to the percentages of unique byte elements within each MAC value for $Tb = 256$. For $Tb = 256$ (32 bytes), the results show that about $\approx 39.6\%$ of the MAC values have all of the 32 bytes unique, about $\approx 28.5\%$ of the MAC values have 31 unique bytes out of the 32 bytes, and about $\approx 20\%$ of the MAC values have 30 unique bytes. The above results confirm that the generated MAC values have uniform distributions with a high level of randomness. Moreover, the entropy test is applied on each of the obtained MAC values at the byte level and the distribution of the obtained entropy values is shown in Fig. 12-b. The entropy values have a normal distribution with a mean equals to 4.88, that is close to the ideal value ($log_2(Tb/8)$)= 5 for Tb=256, and a low standard deviation equals to 0.0831. The entropy results confirm that the MAC values follow the uniform distribution that was shown in Fig. 12-a.



(a) (b)

(c) (d)

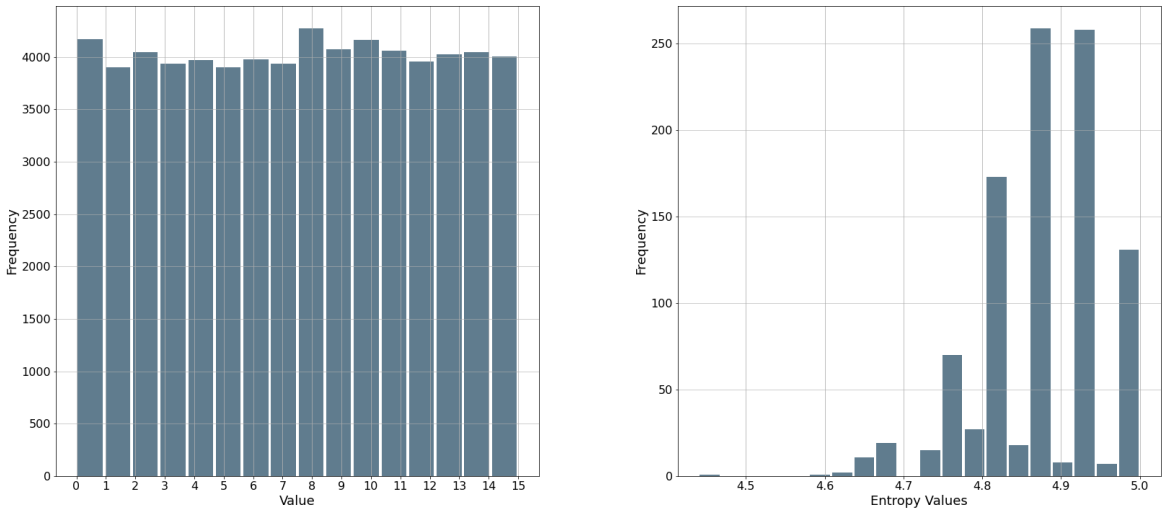
Fig. 11: Representation of the MAC value of two selected random messages (differ in one bit) in hexadecimal format (values between 0 and 15) (a) and (b), and their corresponding recurrence (c) and (d), respectively.

Table 3: Percent of the different number of ASCII characters for n_{MAC}=10000 with $Tb = 256$

Different number of ASCII characters	32	31	30	29	28	27	26	23
Percentage	39.6	28.5	20	8.6	2.9	0.3	0.09	0.01

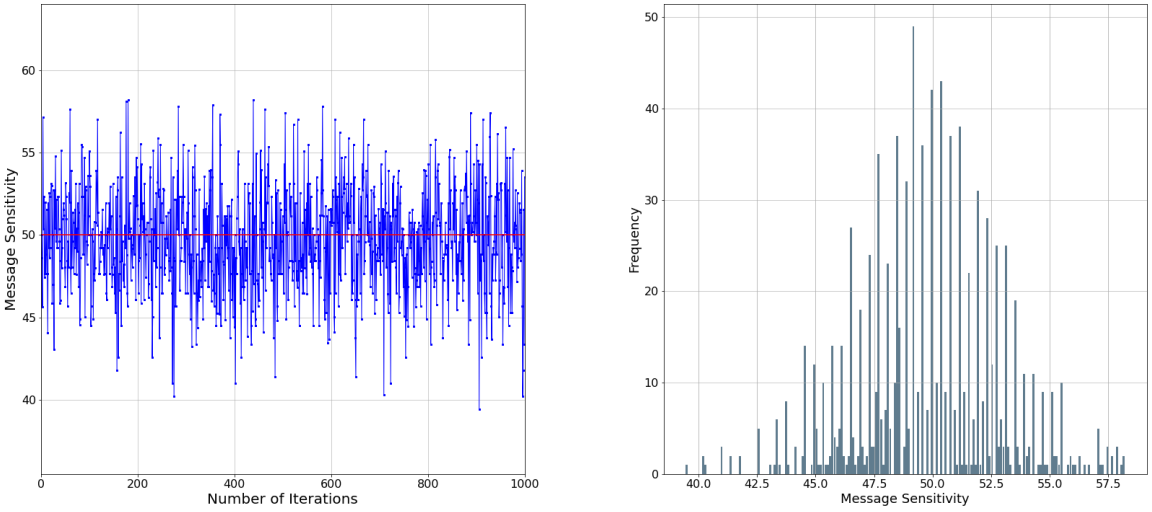
5.2 Sensitivity Tests

Message and key sensitivity (avalanche effect) tests are performed to validate that different MAC values are



(a) (b)

Fig. 12: Distribution of the MAC value in hexadecimal format for 1000 random messages.



(a) (b)

Fig. 13: Variation of the Message Sensitivity (MS) versus 1,000 random dynamic secret keys (changed random bit of the secret key) for DKEMA, (b) and its corresponding distribution, respectively.

obtained for a slight modification in the message or the secret key.

5.2.1 Message Sensitivity (Message Avalanche Effect)

In this test, we consider two messages, which are identical except for a single bit, to be authenticated. Given that the proposed solution is based on the concept of

different cryptographic primitives for each new message, the sensitivity of the MAC value is guaranteed by the key avalanche effect and the substitution-diffusion cryptographic structure. To confirm this, we conducted a sensitivity test of M, for 1,000 random keys, by computing the Hamming distance between the obtained MAC values when the same key is used for two messages differing in one bit. This distance is computed as follows:

$$MS_w = \frac{\sum_{k=1}^{T} MAA_{DK_w}(M_w) \oplus MAA_{DK_w}(M'_w)}{Tb} \times 100\% \quad (3)$$

where MS_w is the obtained percentage of Hamming distance between the MAC value obtained by using the w^{th} message M_w and the modified w^{th} message M'_w with the same dynamic key DK_w; Tb is the number of bits in the MAC value, and $MAC_w = MAA_{DK_w}(M_w)$ and $MAC'_w = MAA_{DK_w}(M'_w)$ are the obtained MAC values using M_w and M'_w, respectively. We consider the difference between M_w and M'_w in the Least Significant Bit (LSB) of a randomly chosen byte.

The message sensitivity test results are shown in Fig. 13 and statistical results are presented in Table 4. The obtained results are similar to the obtained ones with CMAC and HMAC. The plotted results of MS with respect to 1,000 random keys show that, overall, MS is approximately 50% and it follows a normal distribution. Therefore, it is clear from the results that any minor change in the original data message results in a very different MAC value with a difference around 50%. Similar results are obtained for the key sensitivity test, as shown in the following.

5.2.2 Key Sensitivity (Key avalanche effect)

This test aims at evaluating the change in the MAC values for a small change in the dynamic key DK. In the proposed scheme, all the cryptographic primitives, are generated from DK. The desired behavior is that a one bit change in DK should result in a different set of cryptographic primitives, and thus, different MAC values. To confirm this, we conducted a sensitivity test of DK, for 1,000 random keys by computing the Hamming distance between the generated MAC values for the same message when two keys differing in one bit are used. This distance is computed as follows:

$$KS_w = \frac{\sum_{k=1}^{T} MAA_{DK_w}(M) \oplus MAA_{DK'_w}(M)}{Tb} \times 100\% \quad (4)$$

where KS_w is the obtained percentage of Hamming distance between the MAC value obtained by using the w^{th} dynamic key DK_w and the w^{th} modified dynamic key DK'_w with the same input message M_w. In addition, $MAC_w = MAA_{DK_w}(M_w)$ and $MAC'_w = MAA_{DK'_w}(M_w)$ are the obtained MAC values using the dynamic keys DK_w and DK'_w, respectively. We considered a difference between DK_w and DK'_w in the Least Significant Bit (LSB) of a randomly chosen byte.

The key sensitivity test results are included in Fig. 14 and statistical results are presented in Table 4. The obtained results are similar to the obtained ones with CMAC and HMAC. The plotted results of KS with respect to 1,000 random keys show that, overall, KS is approximately 50% and that it follows a normal distribution. Therefore, it is clear from the results that any minor change in the secret key or nonce will result in a very different MAC value for the same message, with a difference around 50%.

5.3 Collision Resistance

Collision resistance is another security requirement for any MAA. It aims at assessing the probability of having two distinct inputs with the same MAC value. Actually, hash functions are designed to resist collision, and to avoid having two distinct inputs with the same MAC. The collision resistance test was applied for 1,000 dynamic keys, and each time, a randomly-selected bit of a random byte is modified in the secret key and/or in the input message. Then, we apply DKEMA and compare the obtained MAC values for the initial and modified data. The number of identical ASCII characters (bytes) at the same positions is computed based on the following equation.

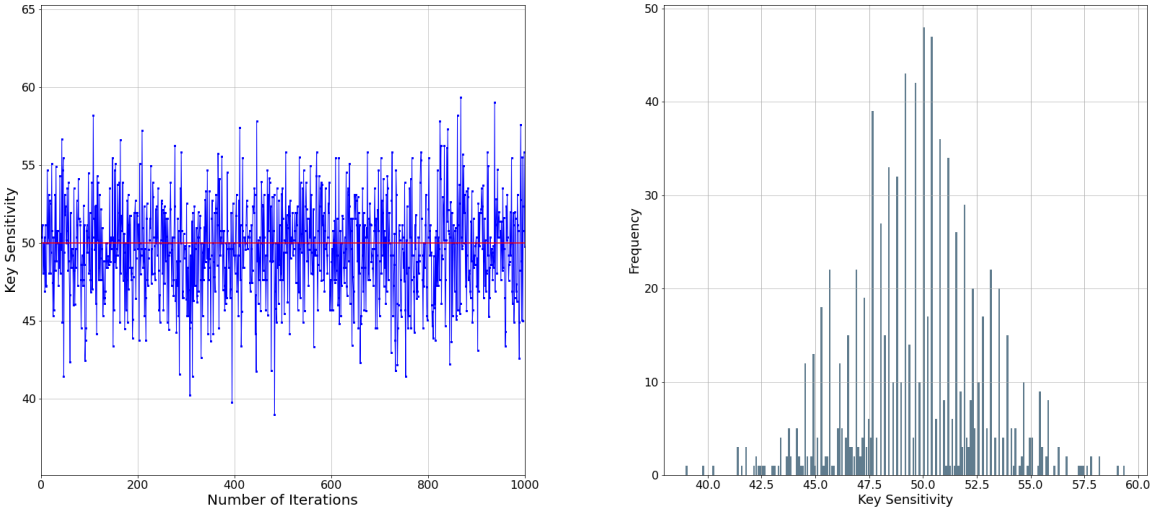
$$Diff = \sum_{i=1}^{Tb} D\{MAC(i), MAC'(i)\}, \quad (5)$$

where $D(x, y) = 1$ if $x = y$, else $= 0$; MAC and MAC' represents the MAC values of the original and modified message, respectively; MAC(i) represents the i^{th} ASCII character of the MAC.

The results, shown in Table 5, show that only 3 characters are equal for $Tb = 256$ in case of a modified secret key, and 2 characters are equal for a modified message. This indicates that there is a very small percentage of similar characters in the obtained MAC values: 1) when considering slightly different inputs with the same key, 2) when considering slightly different secret keys applied

Table 4: Statistical results for message and key sensitivity (message avalanche effect) MS

Sensitivity Test	**Scheme**	**Minimum**	**Mean**	**Maximum**	**Standard Deviation**
MS	Proposed (Tb=256)	39.453	49.7493	58.2031	3.122
	HMAC	39.0625	49.949	63.671	3.1258
	CMAC	39.453	50.026	62.109	3.1331
KS	Proposed (Tb=256)	38.976	49.638	59.375	3.0908
	HMAC	38.6719	50.004	62.891	3.1602
	CMAC	37.891	50.026	62.109	3.0708



(a) (b)

Fig. 14: Variation of the key sensitivity (KS) versus 1,000 random dynamic secret keys (changed random bit of the secret key) for DKEMA, (b) and its corresponding distribution, respectively.

to the same input data, 3) or when considering different secret keys and messages. As such, the proposed scheme is immune against collision and it can resist statistical attacks such as birthday attack, meet-in-the-middle and differential attacks [29].

The obtained results in this section indicates clearly that DKEMA reaches the desired cryptographic properties. In the following section, we discuss its resistance against existing cryptanalysis attacks.

6 Cryptanalysis

In the previous section, the security of DKEMA and the proposed stream-cipher has been tested and discussed in terms of randomness, uniformity, high sensitivity and collision resistance. In this section, the cryptanalysis is presented to validate that the proposed solution is resistant to well-known message authentication attacks. In this section, the cryptanalysis of DKEMA is discussed by considering key- and MAC-related attacks.

6.1 Resistance against Statistical Attacks

In contrast to the majority of existing MAAs, DKEMA is based on the dynamic key approach, with dynamic substitution and seeds for each input data. Previous statistical tests, especially TestU01 and Practrand [30, 31], have confirmed the robustness of DKEMA and its high resistance against statistical attacks.

6.2 Resistance against Brute Force Attacks

The secret-key space of the proposed authentication algorithm is very large, 2^{Tb}, with $Tb =$ equals to 256 bits at least and it can be equal to 512, or 1024 bits. To resist brute force attacks, according to Schneier [2],

Table 5: Percent distribution of the number of ASCII characters with the same value at the same location in the MAC value for the LSB of a random byte in the secret key or message (number of hits) with $Tb = 256$.

Modified input \ hits	0	1	2	3
Secret key	86.9	12.7	0.3	0.1
Message	87	12.1	0.9	-

the key space should be larger than 2^{192}. In our case, the key space is sufficiently large to make brute-force attacks unfeasible. The same is true for the key space of the dynamic key, which is 2^{512}. Therefore, having a large key space, DKEMA can resist against brute force attacks.

6.3 Pseudo-collision Resistance

In a pseudo-collision attack, the attacker tries to modify the message and the associated MAC value, by relying on weaknesses in the compression function [32, 33]. However, DKEMA, which is based on dynamic key-related cryptographic primitives, consists of a parallel substitution-diffusion compression function of message blocks ensuring the avalanche effect. Moreover, the compression function is non-linear. This makes it impossible for attackers to retrieve useful information about a message from the obtained MAC or to obtain another message that can have the same MAC. In addition, a different key-stream sequence is produced for each new input message.

6.4 Resistance against Birthday Attacks

Birthday attacks are traditional attacks that can target the MAA in the aim at finding two messages with same MAC values in less than $2^{\frac{N}{2}}$ trials (here N is equal to Tb, which is the size of the MAC) [34]. In DKEMA, the smallest MAC size is 256, which imposes 2^{128} trials for a brute force attack. As stated in [35], "*if an appropriate padding scheme is used and the compression function is collision-resistant, then the hash function will also be collision resistant*". Indeed, DKEMA, having a large MAC value, is collision resistant, and thus, it is immune against birthday attacks.

6.5 Resistance against Meet-in-the-Middle Attacks

Having a data content with nb blocks $m_1||m_2||\dots||m_{nb}$, the meet-in-the-middle attack aims at finding a block m_i that can replace one of the nb blocks without changing the final MAC value [36,37]. Since DKEMA uses variable cryptograhic primitives for each input data, this renders such an attack impossible. To validate this, we replaced a randomly chosen data block m_i by a random block and we computed the MAC values. This test was repeated $N = 1,000$ times. The results, presented in Fig. 15, show that the difference between the obtained MAC values is at least $Tb/2$ bits (50% of MAC bits are modified) for Tb =256 and similar results were obtained for higher values of Tb.

6.6 Chosen/known Plain-text/cipher-text Attacks

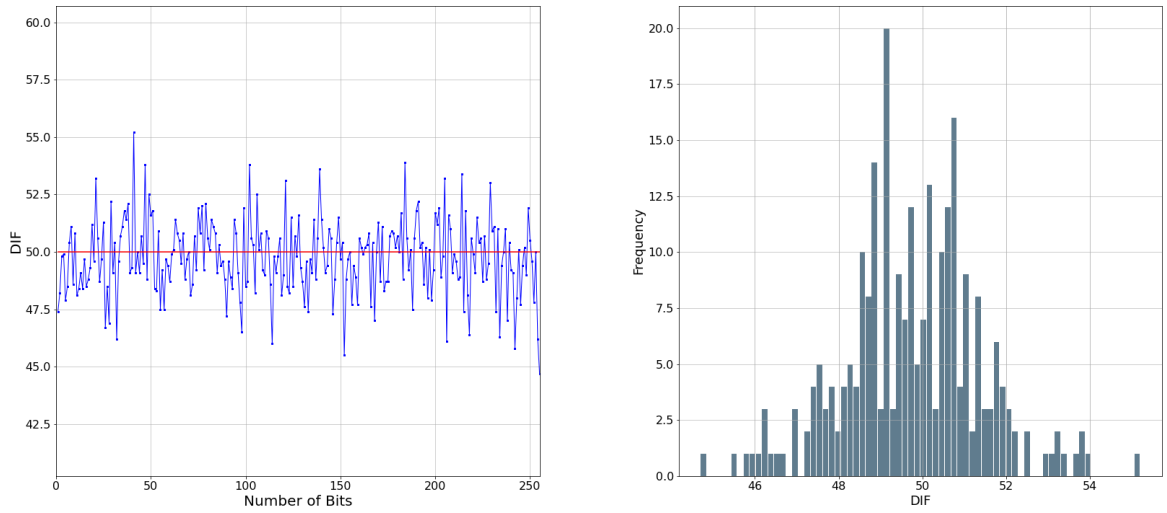
On the other hand, the resistance against chosen/known message attacks is verified due to the dynamic key approach, which drastically complicates the attacker's task. As such, the problems of a single message failure and accidental key disclosure are avoided. Furthermore, modern attacks are ineffective since any change in the dynamic key leads to a significant difference in the produced cryptographic primitives and in the MAC as well. Moreover, the key sensitivity analysis demonstrated a high sensitivity against key-related attacks. These results are sufficient to conclude that no useful information can be inferred from the MAC or the produced key-stream.

7 Performance Analysis

This section evaluates DKEMA and compares it to various encryption algorithms on CPU and compares its performance on different GPU devices. This assessment has been done on a Linux/Debian system. The computational complexity of CF in addition to the average execution time, and speedup between CPU and GPU implementations are presented in this part. Furthermore, the Cuda version 11.3 is employed to implement the required cryptographic algorithm kernels of DKEMA.

7.1 Computation Complexity

In the following, the computation delay of CF is presented. It was designed in order to reach a high degree of security with single round and with a minimum number of operations to decrease the computational complexity



(a) (b)

Fig. 15: Percent of the distribution of changed bit number between $H_n^{''}$ and H versus N tests (a) and its corresponding distribution (b), respectively.

and the associated delay. To compute the compression delay, the following components are used:

1. T_S denotes the required byte substitution time for a block of one word .
2. T_{xor} denotes the required logical XOR execution time between two words.
3. T_{ROTL} denotes the required rotation left execution time for a word.
4. $T_{Splitmix64}$ represents the required delay to iterate the Splitmix64 PRNG once.
5. T_D represents the required delay of the proposed simple binary diffusion process between Nw words.

The Computational Delay (CD) of the proposed scheme to process one block of Nw words is:

$$CD_{MAA} = (Nw+1) \times (T_S + T_{ROTL} + T_{Splitmix64}) + 2 \times (Nw+1) \times T_{xor} + T_D \quad (6)$$

Furthermore, the computation time overhead of the dynamic key and cryptographic primitives derivation (initialization time), that are done at the CPU level depend on the duration of the session. Moreover, these processes are flexible and can be configured for a sub-session time. Besides, the computation time of the dynamic key generation depends only on two operations, which are:

1. The simple "exclusive or" between the nonce N_0 and the secret key
2. The hash of the output of the previous step

In addition, the required computation delay to construct the cryptographic primitives for each session is :

$$CD_{CCP} = T_{PRNG}(N) + 2 \times T_{KSA} \quad (7)$$

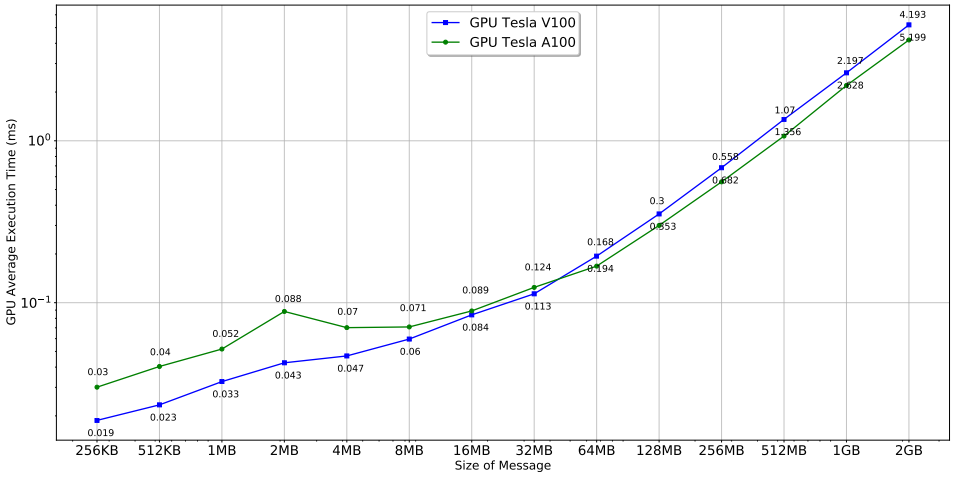
where:

1. $T_{PRNG}(N)$ is the required delay to construct N seeds. In this step, a PRNG or stream cipher will be iterated to produce a keystream with N words.
2. $T_{KSA}(n)$ is the required delay to construct a substitution table by using the KSA of RC4.

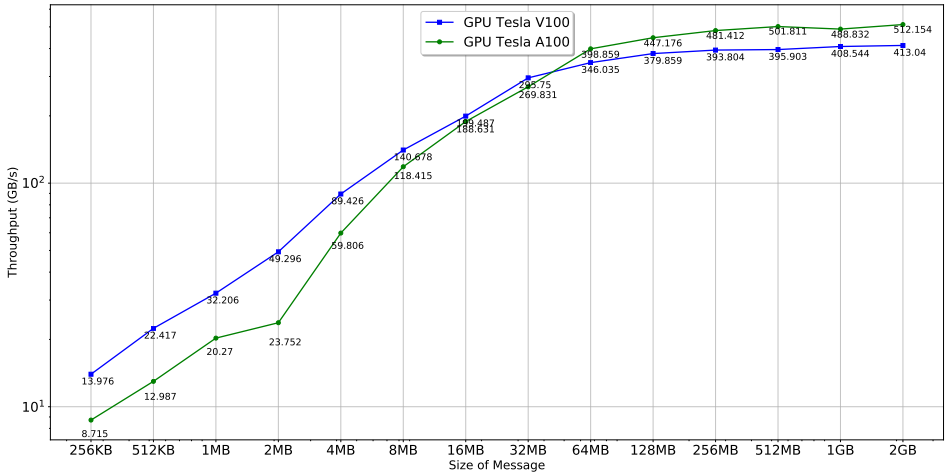
Note that the overhead imposed by the cryptographic primitives derivation is low, since these primitives will be produced once during a session or sub-session. Besides, in contrast to existing compression functions, which require several rounds and multiple operations per round, CF requires only a single round to compress one input block, and the blocks are processed in parallel.

Besides, the complexity of DKEMA is based on 2 steps:

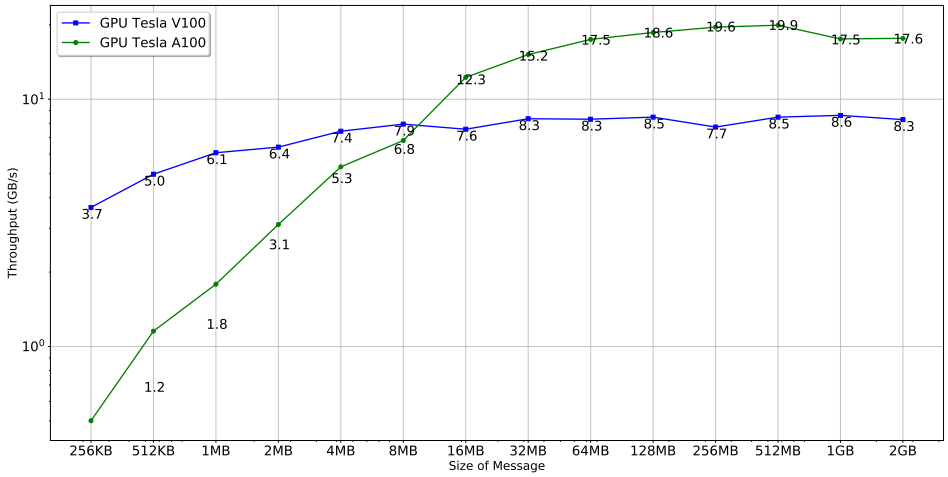
– The first step (first for loop in Algorithm 4) consists of computing the compression function of all message blocks and computing the reduction in parallel on all threads. Thus, the complexity of this step is $O(n)$.



(a) Execution Time(ms)

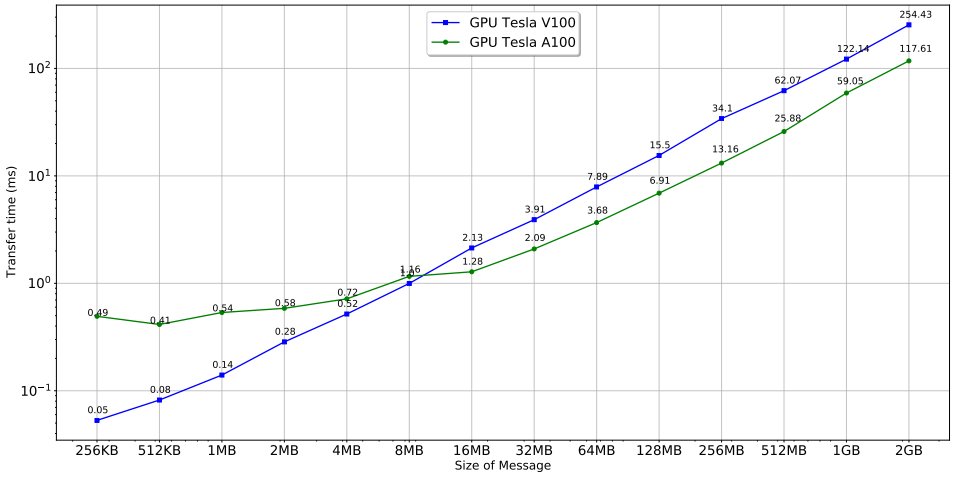


(b) Throughput (without data transfer)

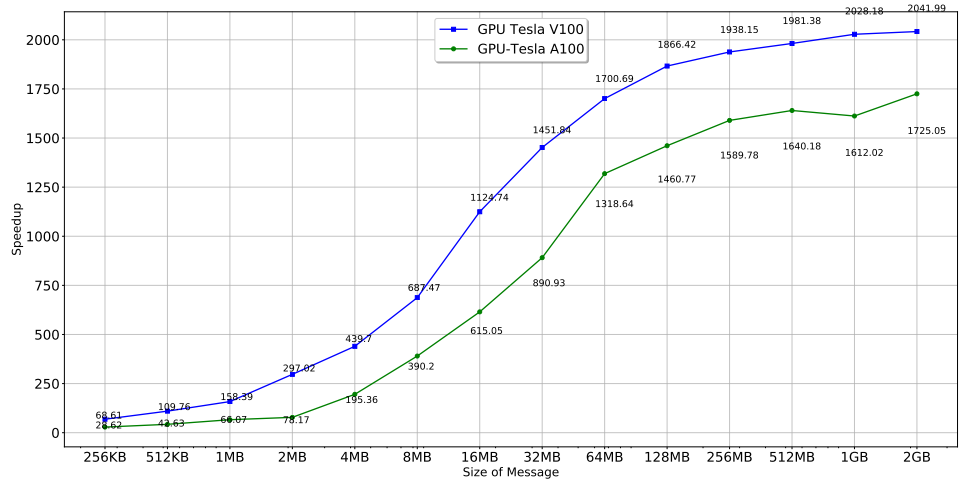


(c) Throughput (with data transfer)

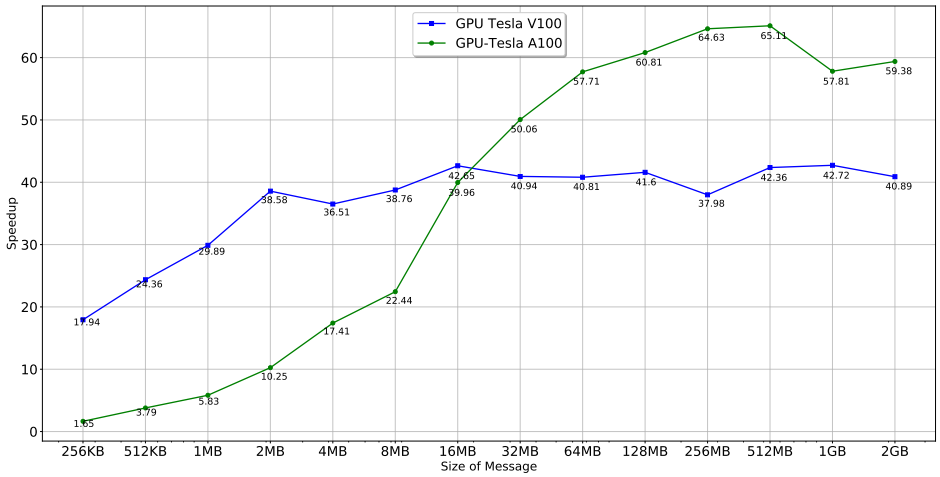
Fig. 16: Variation of the average GPU execution time (a), GPU throughput without (b) and with (c) data transfer between CPU and GPU on Tesla V100 and Tesla A100 versus message size.



(a) Data transfer time



(b) Without data transfer time



(c) With data transfer time

Fig. 17: Variation of the data transfer time (a) in addition to the speedup of DKEMA on Tesla V100 and Tesla A100 versus message size without (b) and with (c) taking in consideration the transfer time between host and device.

– Then, the second step (in Algorithm 6) consists of applying the reduction, which has a complexity of $O(logn)$. In the proposed implementation, multiple compressed blocks will be xored per thread sequentially.

7.2 Experimental Results over CPU

To compare the performance of the proposed solution on GPU to existing MAAs (CMAC and HMAC) that can be applied on CPU, we run tests on an Intel (R) Xeon(R) CPU E5-2698 v4. Each core of this CPU operates at a frequency of 2.20 GHz. The optimized implementation of these MAAs with OpenSSL [38] are executed on this processor and their throughput results are presented in Fig. 18-(a). This result indicates clearly that the size of message has no effect on throughput. This is logical as existing MAAs are based on the chaining mode and cannot benefit from the parallel computing. This means that the throughput is more or less constant with large messages on CPU. It is obvious since the parallelism capability is quite limited.

7.3 Experimental Results over GPUs

To the best of our knowledge, there is no existing MAA that has been designed and implemented on GPU. Therefore, to justify the efficiency of the proposed solution, the speedup results are highlighted to compare the throughput of DKEMA with GPU compared to CPU. The results indicate that the speedup can reach greater than 2040 if data is located on GPU. Otherwise, if data is located on CPU, taking into consideration the required transfer time of message from CPU to GPU, the speedup becomes close to 59 at maximum. Note that the performance of DKEMA is evaluated on two GPUs (Tesla V100 and Tesla A100), which are described in Table 6.

In this part, the required GPU average execution time (milliseconds) and throughput (GigaBytes/s) versus message length of DKEMA are presented in Fig. 16. Fig. 16 shows that the average execution time and throughput increase when the message length increases. It is clear from these results that using a powerful GPU provides better speedup since the best GPU throughput (512 GigaBytes/s) is achieved with the Tesla A100 if no need to data transfer. In addition, the maximum throughput reaches by using Tesla V100 is 413 GB.

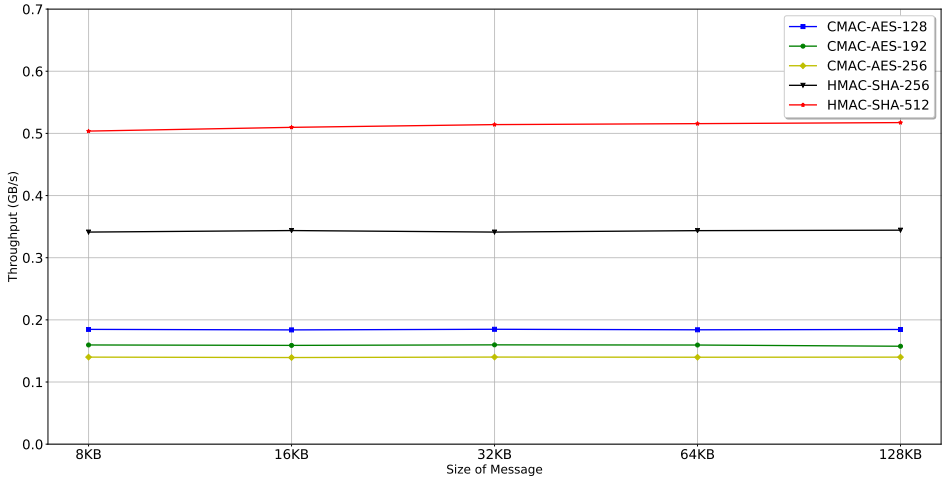
On the other hand, the variation of data transfer time versus message size is presented in Fig. 17-(a) for two different GPUs. This result shows that the variation of data transfer time is approximately linear in function of message length. In addition, for message length $\leq$ 8MB, Tesla V100 requires less transfer time compared to Tesla A100. While for message length $>$ 8MB, Tesla A100 requires less transfer time compared to Tesla V100.

Besides, the GPU throughput decreases when a data transfer is needed, as shown in Fig. 16-(c), and the maximum throughput become 8.3GB and 17.6 GB for Tesla V100 and Tesla A100, respectively.

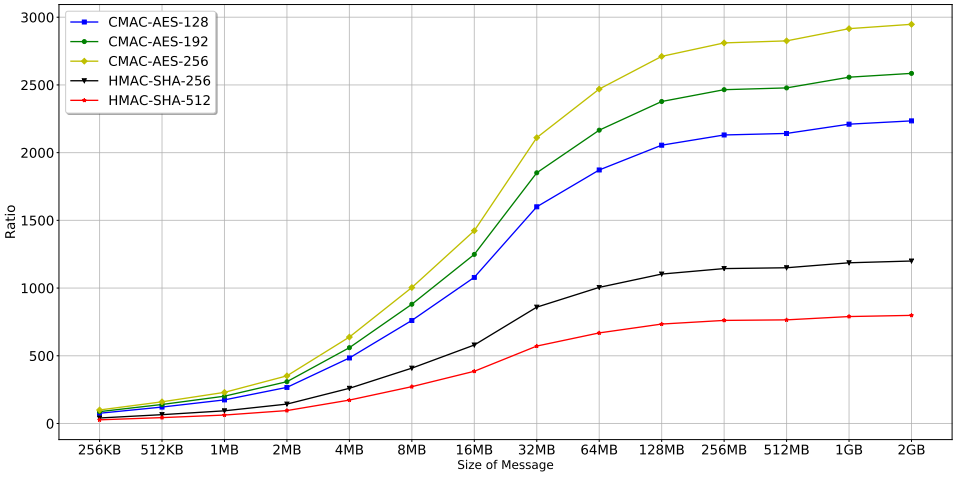
It should be noticed that we consider two possible scenarios, where the first one assumes that input data to be authenticated are already in the GPU. This case can be considered as the fine-grained parallel processing time in the GPU. In this case, the best throughput and consequently speedup is reached (see Fig. 17-(b)) with a factor up to 2K for Tesla A100. In practice, this scenario is realistic with an application dedicated to GPU. Therefore, only the computing times on the GPU are taken into account. The second scenario is when data are in the CPU, and consequently the data transfer time from CPU to GPU must be taken into account but this time depend on the employed hardware and the PCI bus in addition to message length. This case is the most real scenario of heterogeneous-based systems since they are always based on the interaction between the host (CPU) and the device (such as the GPU). Furthermore, the main purpose of a device (hardware accelerator) is to operate on specific workloads under the supervision of the host of the system. Unfortunately, which this scenario, less throughput and consequently less speedup is reached compared to the first scenario as shown in Fig. 17-(c). Although, DKEMA, in the case of complete system interaction (Host and GPU), improves speedup with a factor close to 59 for Tesla A100.

Moreover, comparing the obtained results in Fig. 18-(a) to Fig. 16-(b) and (c), the CPU result shows that the message size has no effect on the throughput, which is constant for data length $\geq 8KB$. In contrast, the GPU results show that the throughpu increases in function of data length. The GPU implementation of DKEMA reaches the maximum throughput for 256MB in case there is no need to transfer message from CPU to GPU.

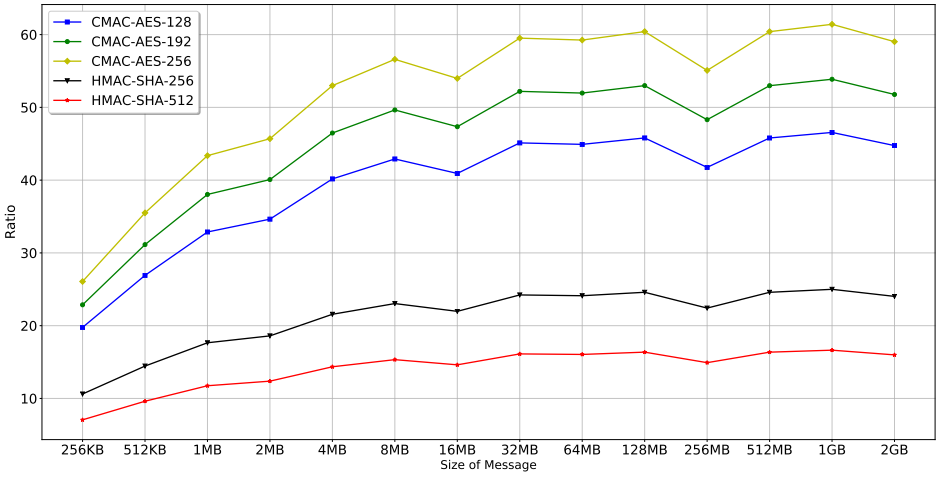
Therefore, we can conclude that DKEMA has higher throughput compared with the existing standard MAAs (less than 1GB with CPU). In addition, we present the ratio throughput between GPU (proposed solution) and CPU of existing chaining MAA with or without data transfer in Fig. 18-(b) and (c), respectively. These results



(a) CPU throughput



(b) Throughput ratio between DKEMA (GPU) and existing standard MAAs (CPU) without data transfer



(c) Throughput ratio between DKEMA (GPU) and existing standard MAAs (CPU) with data transfer time

Fig. 18: The throughput of existing standard MAAs (CMAC and HMAC) implemented by OpenSSL on Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz. In addition, the throughput ratio between DKEMA (GPU Tesla V100) and existing standard MAAs (CPU) without (b) and with (c) data transfer for message length varying between 8KB and 128KB.

Table 6: GPU devices during bench-marking

Tesla A100	**Tesla V100**
– Compute capability: 8.0 – Global memory: 40,000 MB – GPU frequency: 1.41 GHz – Memory frequency: 1,215 MHz – Number of Cuda cores: 6,912	– Compute capability: 7.0 – Global memory: 16,152 MB – GPU frequency: 1.53 GHz – Memory frequency: 877 MHz – Number of Cuda cores: 5,120

show that the ratio is higher and increase in function of message length. Furthermore, ratio Fig. 18-(c) is reduces as GPU throughput is reduced when data transfer is needed.

All these results validate that DKEMA on GPU is faster compared to well-used non-parallel MAAs on CPU, and best performance is reached when no need to data transfer.

8 Conclusion

In this paper, we propose a new MAA solution, DKEMA, designed for GPU implementation. This solution, being dynamic key-dependent, and one-round based, can ensure a high level of efficiency and robustness. For the best of our knowledge, we are the first to design a parallel MAA targeting a GPU implementation, which makes it preferable for limited latency applications. Moreover, DKEMA scheme offers a high degree of randomness, which was validated by a set of statistical tests. Moreover, the implementation of DKEMA is very simple compared to other existing MAAs (no chaining operation mode and one round single compression function). The robustness of DKEMA and stream cipher have been assessed and confirmed via cryptanalysis along with different benchmark tests. Note that other existing cryptanalysis techniques are designed to target static structures, which is not the case of DKEMA. Finally, the execution time and throughput tests show that, compared to existing MAAs such as CMAC and HMAC, DKEMA has a lower cost in terms of processing and execution time. As future work, the design of an efficient parallel message authentication-encryption algorithm will be investigated based on the GPU characteristics.

Acknowledgement

This paper is funded by the EIPHI Graduate School (contract "ANR-17-EURE-0002"). We also thank the supercomputer facilities of the Mésocentre de calcul de Franche-Comté.

References

1. Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Advanced Encryption Standard*. Alpha Press, 2009.
2. William Stallings. *Cryptography and Network Security: Principles and Practice*. Pearson Upper Saddle River, NJ, 2017.
3. Qinjian Li, Chengwen Zhong, Kaiyong Zhao, Xinxin Mei, and Xiaowen Chu. Implementation and Analysis of AES Encryption on GPU. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, pages 843–848. IEEE, 2012.
4. Guang-liang Guo, Quan Qian, and Rui Zhang. Different Implementations of AES Cryptographic Algorithm. In *High Performance Computing and Communications (HPCC), IEEE 7th International Symposium on Cyberspace Safety and Security (CSS)*, pages 1848–1853. IEEE, 2015.
5. Rone Kwei Lim, Linda Ruth Petzold, and Çetin Kaya Koç. Bitsliced High-performance AES-ECB on GPUs. In *The New Codebreakers*, pages 125–133. Springer, 2016.
6. Raphaël Couturier. *Designing Scientific Applications on GPUs*. Numerical Analysis & Scientific Computating. Chapman & Hall/CRC, 2013.
7. Nvidia, CUDA. A C Programming Guide, version 9.0. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.
8. Jacques Bahi, Raphaël Couturier, Christophe Guyeux, and Pierre-Cyrille Héam. Efficient and Cryptographically Secure Generation of Chaotic Pseudorandom Numbers on GPU. *The Journal of Supercomputing*, 71(10):3877–3903, 2015.
9. Wai-Kong Lee, Hon-Sang Cheong, Raphael C-W Phan, and Bok-Min Goi. Fast Implementation of Block Ciphers and PRNGs in Maxwell GPU Architecture. *Cluster Computing*, 19(1):335–347, 2016.
10. Biagio Peccerillo, Sandro Bartolini, and Çetin Kaya Koç. Parallel Bitsliced AES through PHAST: a Single-Source High-Performance Library for Multi-Cores and GPUs. *Journal of Cryptographic Engineering*, pages 1–13, 2017.
11. Like Chen and Runtong Zhang. A Key-dependent Cipher DSDP. In *Electronic Commerce and Security, 2008 International Symposium on*, pages 310–313. IEEE, 2008.
12. Runtong Zhang and Like Chen. A Block Cipher using Key-dependent S-box and P-boxes. In *Industrial Electronics, 2008. ISIE 2008. IEEE International Symposium on*, pages 1463–1468. IEEE, 2008.
13. Hassan Noura, Ali Chehab, Lama Sleem, Mohamad Noura, Raphaël Couturier, and Mohammad M Mansour. One Round Cipher Algorithm for Multimedia IoT Devices. *Multimedia Tools and Applications*, pages 1–31, 2018.
14. Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. Simon

and speck: Block ciphers for the internet of things. *IACR Cryptology ePrint Archive*, 2015:585, 2015.
15. Hassan N Noura, Mohamad Noura, Ali Chehab, Mohammad M Mansour, and Raphaël Couturier. Efficient and secure cipher scheme for multimedia contents. *Multimedia Tools and Applications*, pages 1–30, 2018.
16. Hassan N Noura, Ali Chehab, Mohamad Noura, Raphaël Couturier, and Mohammad M Mansour. Lightweight, dynamic and efficient image encryption scheme. *Multimedia Tools and Applications*, pages 1–35, 2018.
17. Hassan Noura, Lama Sleem, Mohamad Noura, Mohammad M Mansour, Ali Chehab, and Raphaël Couturier. A New Efficient Lightweight and Secure Image Cipher Scheme. *Multimedia Tools and Applications*, 77(12):15457–15484, 2018.
18. Zeinab Fawaz, Hassan Noura, and Ahmed Mostefaoui. An Efficient and Secure Cipher Scheme for Images Confidentiality Preservation. *Signal Processing: Image Communication*, 42:90–108, 2016.
19. Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.
20. Andrew Lauritzen and Summed-Area Variance Shadow Maps. Chapter 36. aes encryption and decryption on the gpu | nvidia developer. `https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-36-aes-encryption-and-decryption-gpu`, 2007.
21. Ahmed Awadalla Abdelrahman, Mohamed Mahmoud Fouad, and Hisham Dahshan. Analysis on the aes implementation with various granularities on different gpu architectures. *Advances in Electrical and Electronic Engineering*, 15(3):526–535, 2017.
22. Ahmed A Abdelrahman, Mohamed M Fouad, Hisham Dahshan, and Ahmed M Mousa. High performance cuda aes implementation: A quantitative performance analysis approach. In *2017 Computing Conference*, pages 1077–1085. IEEE, 2017.
23. Ahmed A Abdelrahman, Hisham Dahshan, and Gouda I Salama. Enhancing the actual throughput of the aes algorithm on the pascal gpu architecture. In *2018 3rd International Conference on System Reliability and Safety (ICSRS)*, pages 97–103. IEEE, 2018.
24. Hassan N Noura, Ola Salman, Raphaël Couturier, and Ali Chehab. Lorca: Lightweight round block and stream cipher algorithms for iov systems. *Vehicular Communications*, page 100416, 2021.
25. Hassan Noura, Ola Salman, Raphael Couturier, and Ali Chehab. Novel one round message authentication scheme for constrained iot devices. *Journal of Ambient Intelligence and Humanized Computing*, 2021.
26. Hassan N Noura, Ali Chehab, and Raphael Couturier. Efficient & secure cipher scheme with dynamic key-dependent mode of operation. *Signal processing: Image communication*, 78:448–464, 2019.
27. Hassan Noura. *Conception et simulation des générateurs, crypto-systèmes et fonctions de hachage basés chaos performants*. PhD thesis, université de Nantes, 2012.
28. Pseudo-random numbers/splitmix64 - rosetta code. `https://rosettacode.org/wiki/Pseudo-random_numbers/Splitmix64`.
29. Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In *In EUROCRYPT*. Springer-Verlag, 2005.
30. Guy L Steele Jr and Sebastiano Vigna. Lxm: better splittable pseudorandom number generators (and almost as fast). *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–31, 2021.
31. Augusto Parisot, Lucila MS Bento, and Raphael CS Machado. Testing and selecting lightweight pseudo-random number generators for iot devices. In *2021 IEEE International Workshop on Metrology for Industry 4.0 & IoT (MetroInd4. 0&IoT)*, pages 715–720. IEEE, 2021.
32. Amir Akhavan, Azman Samsudin, and Afshin Akhshani. A novel parallel hash function based on 3d chaotic map. *EURASIP Journal on Advances in Signal Processing*, 2013(1):1–12, 2013.
33. B. Yang, Z. Li, S. Zheng, and Y. Yang. Hash function construction based on coupled map lattice for communication security. In *Global Mobile Congress 2009*, pages 1–7, Oct 2009.
34. Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.
35. Ivan Damgård. A design principle for hash functions. In *Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '89, pages 416–427, London, UK, UK, 1990. Springer-Verlag.
36. Mohamed Amin, Osama S Faragallah, and Ahmed A Abd El-Latif. Chaos-based hash function (cbhf) for cryptographic applications. *Chaos, Solitons & Fractals*, 42(2):767–772, 2009.
37. A Kanso and M Ghebleh. A structure-based chaotic hashing scheme. *Nonlinear Dynamics*, 81(1-2):27–40, 2015.
38. Viega John, Matt Messier, and Pravir Chandra. Network security with openssl: Cryptography for secure communications. *O'Reilly Media, Inc.*, 2002.

Appendices

Algorithm 4 kernel reduceSinglePassMultiBlockCG

```
__global__ void reduceSinglePassMultiBlockCG(const ulong * __restrict__ g_idata,
ulong * __restrict__, g_odata, unsigned int n, const uchar * __restrict__ Sbox8,
const uchar * __restrict__ DK)
{
    cg::thread_block block = cg::this_thread_block();
    cg::grid_group grid = cg::this_grid();
    extern ulong __shared__ sdata3[];
    ulong *sdata = (ulong*)sdata3;
    ulong *sdata2 = (ulong*)&sdata3[block.size()];
    uchar *ssbox =(uchar*)&sdata3[block.size()*2];
    sdata[block.thread_rank()] = 0;
    sdata2[block.thread_rank()] = 0;
    if(block.thread_rank()<256) {
      ssbox[block.thread_rank()]=Sbox8[block.thread_rank()];
    }
    uint64_t r1;
    for (int i = grid.thread_rank(); i < n; i += grid.size()) {
      r1=(i)^(((uint64_t*)DK)[i&63])^g_idata[i];
      r1=rotl(r1,ssbox[i&255]&63);
      apply_sub_and_prng(&r1,ssbox);
      sdata2[block.thread_rank()]= r1;
      if ((block.thread_rank()&3) == 0) {
        r1=sdata2[block.thread_rank()]^sdata2[block.thread_rank()+1]^
            sdata2[block.thread_rank()+2]^sdata2[block.thread_rank()+3];
        apply_sub_and_prng(&r1,ssbox);
        sdata[block.thread_rank()]^=sdata2[block.thread_rank()]^r1;
        sdata[block.thread_rank()+1]^=sdata2[block.thread_rank()+1]^r1;
        sdata[block.thread_rank()+2]^=sdata2[block.thread_rank()+2]^r1;
        sdata[block.thread_rank()+3]^=sdata2[block.thread_rank()+3]^r1;
      }
    }
    cg::sync(block);
    reduceBlock(sdata, block);
    if (block.thread_rank() == 0) {
      g_odata[blockIdx.x*szblock] = sdata[0];
      g_odata[blockIdx.x*szblock+1] = sdata[1];
      g_odata[blockIdx.x*szblock+2] = sdata[2];
      g_odata[blockIdx.x*szblock+3] = sdata[3];
    }
    cg::sync(grid);
    if (grid.thread_rank() == 0) {
        for (int i = 1; i < gridDim.x; i++) {
            g_odata[0] ^= g_odata[i*4];
            g_odata[1] ^= g_odata[i*4+1];
            g_odata[2] ^= g_odata[i*4+2];
            g_odata[3] ^= g_odata[i*4+3];
        }
        sdata[0]=g_odata[0]^g_odata[1]^g_odata[2]^g_odata[3];
        apply_sub_and_prng(&sdata[0],ssbox);
        g_odata[0]=g_odata[0]^sdata[0];
        g_odata[1]=g_odata[1]^sdata[0];
        g_odata[2]=g_odata[2]^sdata[0];
        g_odata[3]=g_odata[3]^sdata[0];
    }
}
```

Algorithm 5 kernel *apply_sub_and_prng*

```
__device__ void  apply_sub_and_prng(ulong *r1, uchar *ssbox) {
    uchar *rr = (uchar*)r1;
    rr[0]=ssbox[rr[0]];
    rr[1]=ssbox[rr[1]];
    rr[2]=ssbox[rr[2]];
    rr[3]=ssbox[rr[3]];
    rr[4]=ssbox[rr[4]];
    rr[5]=ssbox[rr[5]];
    rr[6]=ssbox[rr[6]];
    rr[7]=ssbox[rr[7]];
    *r1=splitmix64(*r1);
}
```

Algorithm 6 kernel reduceBlock

```
__device__ void reduceBlock(ulong *sdata, const cg::thread_block &cta)
{
    const unsigned int tid = cta.thread_rank();
    cg::thread_block_tile<32> tile32 = cg::tiled_partition<32>(cta);
    ulong beta  = sdata[tid];
    ulong temp;

    for (int i = tile32.size()/ 2; i >= szblock; i >>= 1) {
        if (tile32.thread_rank() < i) {
            temp       = sdata[(tid+i)];
            beta       ^= temp;
            sdata[tid] = beta;
        }
        cg::sync(tile32);
    }
    cg::sync(cta);
    if (cta.thread_rank() <4) {
      beta  = 0;
      for (int i = tid; i < blockDim.x; i += tile32.size()) {
        beta  ^= sdata[i];
      }
      sdata[tid] = beta;
    }
    cg::sync(cta);
}
```