An Efficient Parallel Implementation of a Light-weight Data Privacy Method for Mobile Cloud Users

Mehdi Bahrami, Dong Li, Mukesh Singhal
Cloud Lab
University of California, Merced
{mbahrami;dli35;msinghal}@ucmerced.edu

Ashish Kundu
IBM Thomas J. Watson Research Center
Yorktown Heights, NY, USA
akundu@us.ibm.com

Abstract—**Cloud computing provides an opportunity to users to outsource their data and applications. However, data privacy is one of the key challenges for the users who are outsourcing data on some transparent cloud servers. Data encryption is the best option to protect users' data privacy on the cloud. However, computation overheads of encryption methods could be expensive to some small computing machines, such as mobile or IoT devices with limited resources, such as battery. In our previous study, we developed a light-weight Data Privacy Method (DPM) based on a chaos system that uses a Pseudo Random Permutation (PRP) to scramble the content of original data. Although the nature of PRP is against parallelization, we provide an efficient parallel algorithm to scramble a file while the file splits into multiple chunks. The parallel DPM avoids an adversary to access the original data (e.g., by using a brute-force attack), when the size of each scrambled data is large enough. In this paper, we accelerate DPM on a Graphic Processing Unit (GPU) by using NVIDIA CUDA platform for implementation. We assess the generated shuffle addresses from pseudo-random and the distribution of randomness when the computation on data is parallelized on a multiple GPU-cores. A set of rigorous evaluation results shows that the parallel DPM provides a superior performance over tradition DPM when the most time consuming of native CUDA parallel functions have monitored. We also perform a security analysis of parallel DPM to ensure it is secure and it is a cost effective model to protect users' data privacy in a cloud environment.**

Keywords—***GPU; Cloud Computing; Parallel Computing; Data Privacy; Cloud Privacy;***

I. INTRODUCTION

Cloud Computing and parallel computing paradigms introduce several advantages for processing heavy computation methods. Our study in this paper is based on these two paradigms which are described in the following sub-sections.

A. Cloud Computing

Cloud computing is an emerging technology that combines of distributed systems, and virtualization technology to offer new computing concepts. A cloud computing system shares all available resources with multiple users, and each user is billed base on pay-per-use model or similar payment model. The cloud enables users to increase or decrease their resources on-the-fly Cloud computing has been used in different disciplines, such as mobile computing, robotics when data is outsourced on the cloud. *Mobile Cloud Computing (MCC)* paradigm allows mobile users to outsource their data and applications to the cloud and *MCC* enables mobile users to run complex application on server-side. However, there is a tradeoff in *MCC* between processing faster of data and saving power resources, which is critical for mobile devices. In particular, using cloud computing on mobile devices is a challenge because not all applications are able to efficiently outsource data to the cloud [1]. For instance, if a user outsources data to the cloud by using an application but the application drains the battery due to periodically download/upload files, the application is not efficient to be used. The trade-off between power resource and computation speed are critical in *MCC*. Data privacy is another parameter when users outsource their data to *MCC*. Data security and network security methods can be used in order to protect user privacy in cloud computing systems. However, using complex security methods on mobile devices raise resource limitation challenge [2]. Therefore, not all complex security methods are able to protect user data privacy by providing a balance between resource power and computation speed.

B. Parallel Computing

Parallel computing has been popular since computing machine introduced. Parallel computing allows complex algorithms to be run in parallel in order to increase the computation speed.

Recently, the implementation of parallel computing algorithms has been transferring from massive server machines to small personal computers when open-forums and corporations have introduced new platforms that allows users to have efficient parallel computing on

personal computers. For instance, *CUDA* platform uses *Graphics Processing Unit (GPU)* and it was introduced by *NVIDIA*. This platform enables a user or even a mobile user to process a procedure in parallel that needs intensive computation. The *GPU-based* computing paradigm of parallel computing allows a device to use one or multiple *GPUs* to perform heavy computations where each *GPU* consist of thousands of small and low speed cores. Regularly, each submitted task to a core is a repeated computing process that each instruction does not need heavy computation. *GPU-based* computation improves the performance of processing of complex methods by parallelizing small tasks of a complex method on each *GPU* core. In our previous study, we developed a light-weight data privacy method (*DPM*) [2] that uses a chaos system [6] based on *Pseudo Random Permutation* (*PRP*) to scramble data. *DPM* uses pre-generated arrays that contain random addresses of chunks of data. The *DPM* provides a superior performance over other existing data security methods that needs heavy computation, such as *Advanced Encryption Standard (AES)* [8].

C. Threat Model

In this section, we are describing the specific threats that our proposed technique shall protect the privacy against.

If a mobile cloud user wishes to outsource her photos to a cloud, such as Google Drive, Dropbox, but she would not like to share the original content with the cloud vendor, she may encrypt the content and submit the encrypted photos to the cloud. However, encrypting each file may drain the battery power in short period time. Another option, she might use a *PRP* method to scramble the content of all photos based on bits. Then, submit the scramble data of the photos to the cloud. In this case, the *PRP* generator should be secure and use a lengthy chunk of data (e.g., using a lengthy size of bits) to permute the original content and then submit it to the cloud. As another example, if a user submits a text file or a database file, she might use *PRP* to scramble the original content in order to submit a confusion content to the cloud. In the case of a database, a query also can be performed on a scrambled data without reconstructing the original content [7]. Generally, our proposed method shall use available resources on a mobile device (e.g., cell phone) to scramble the original content. Our proposed method uses *GPU* on mobile device to parallelize the method in order to save device's power resources and process the method faster than device *CPU*. In addition, public analysis tools for data analysis from cloud vendors, or their third-party applications are not able to simply access the original user's photo, text file, or database. For instance, third-party application and bulk data analysis tools are not able to process users' data without reconstructing the original data from scrambled data. If users use a complex model of *PRP*, then reconstructing the original file would be more difficult for cloud vendors or their-third party partners.

In the previous study, we describe how a mobile device access to the shuffle addresses for different chunk sizes of an original file. However, in this paper, we consider implementation of *DPM* on *GPU* to generate *PRP* numbers to permute the original content and outsource scrambled content to a cloud that does not allow bulk data analysis review user's content. The *GPU-based DPM* enables the process to be run on *GPU* core instead of *CPU* in order to improve *DPM* performance.

The rest of the paper is organized as follows: the next section presents the motivation of the study and the major challenges to maintain data privacy while using cloud computing. *Section III* presents the related work. *Section IV* presents the background of this study. *Section V* presents the proposed method on *GPU*. The experimental setup and its results are described in *Section VI*. The security analysis of DPM presents in *Section VII* which shows the security assumptions and the level of security for the proposed method. Finally, *Section VIII* concludes the study.

II. MOTIVATION

The following two options can be considered for shuffling data: (i) using an online *PRP* generator to produce shuffle addresses; (ii) using a set of pre-generated arrays of *PRP* [7] (offline mode). The first option causes an issue on a mobile device because the computation time of generating *PRP* is expensive on mobile devices. The second option is preferred because it removes additional costs for generating *PRP*. However, it uses mobile device storage that could cause indirect issue. In this paper, we consider the first option but we generate arrays of *PRP*s on-the-fly and the process distributes to multiple cores of a *GPU* in order to reduce computation time.

The important challenge which is our focus is data privacy for mobile users because when a user outsources data to the cloud, data privacy can be violated by the cloud vendor, the vendor's partners, hackers, malicious entities or even by other cloud users.

III. RELATED WORK

To the best of our knowledge, no research has been published in the area of implementation of *PRP* on *GPU* because its nature of sequential when it stands against parallelism concept. However, some studies have been published for implementation of other random number generators on *GPU* or *FPGA*. For instance, Thomas et al. [11] compare the performance of three types of random number generators on *CPU*, *GPU* and *FPGA*. In this study, authors use an appropriate algorithm, such as the uniform, Gaussian, and exponential distribution for each hardware platform in order to have efficient power peaks and computations. This study shows that the

performance of the different random-number generators relies on their platform. In this paper, we consider *CUDA* platform on *GPU* in order to optimize the performance and power consumption which is not investigated in [11]. In another study, Tsoi et al. [10] implemented two different random number generators for embedded cryptographic applications on *FPGA*. The first is a true random number generator (*TRNG*) [13] based on oscillator phase noise, and the second is a bit serial implementation of a *Blum Blum Shub (BBS)* [18]. The study shows that *TRNG* is recommended for low-frequency-clock processors. Since *GPU* often consists of thousands of cores and with low speed and smaller than *CPU* cores, we consider this fact for designing small-scale generators in our study which is described in Section V. This consideration makes *PRP* to be highly suited to the target platform. In the similar study by Manssen et al. [15], the authors evaluate different random number generators with different granularity. There are some studies on processing *AES* on *GPU* [3, 4, 5] or using the similar method for the security processing [9] on a *GPU*.

IV. BACKGROUND OF THE STUDY

DPM splits an original file into several chunks. The method uses a pattern to split original file into multiple files when each file consists of random part of the original file. Then, DPM selects a set of bits (chunk) of each split file and finally it scrambles the content of each chunks by using a chaos system [6]. The DPM saves 72% battery power over AES encryption method because *DPM* can be run in *O(1)* time complexity for each chunks and it requires *O(n)* for *n* chunks.

Pseudorandom Permutation (*PRP*) is the key module of many methods, such as encryption and simulations as well as *DPM*. *PRP* is defined as follows:

$$F \text{ is mapping } \{0,1\}^n \times \{0,1\}^s \rightarrow \{0,1\}^n \quad (1)$$

F *is a PRP if*:

(i) $\forall K \text{ where } K \in \{0,1\}^s$, F is a bijection of $\{0,1\}^n \rightarrow \{0,1\}^n$ (2)

(ii) $\forall K \text{ where } K \in \{0,1\}^s$ $F_K(x)$ is an efficient algorithm. (3)

(iii)

$$D: \Pr(D^{F_K}(1^n) = 1) - \Pr(D^{f_n}(1^n) = 1)| < \varepsilon(s), \text{where } K \leftarrow \{0,1\}^s \quad (4)$$

where $\Pr(.)$ is the probability of raising the input event.

As discussed in (4), the *PRP* provides a uniform distribution between all generated elements of *F*. This property of *PRP* passes the important perfect secrecy parameter which was introduced on Shannon's theory [12] for encryption functions [2].

The main function for generating pseudo number is defined as follows:

$$F_{k+1} = \mu F_k(1 - F_K) \quad (5)$$

where $P \in \{0,1\}$ and μ is a parameter of this equation.

In the classic chaos system problem, if μ is selected between $3.569945 \leq \mu \leq 4$, and with an initial value of $F_0 = [0,1]$, F provides a complex chaos model [2]. F uses a set ξ to provide a non-convergent, non-periodic pseudo random numbers [2]:

$$\xi = \{P_k\}_{k=0}^{\omega} \quad (6)$$

where ω is maximum number of an original content.

The *DPM* splits the content of an original content to ω number of chunks. Then, it uses ξ to shuffle the content of chunks. We employed conflict-remover algorithm which is described in [16] to provide a set of unique addresses for each chunks based on input parameter (μ) and selected pattern by a user.

V. THE PROPOSED METHOD

The main challenges of *DPM* are: (i) generating ξ (a set of addresses which is used to permute an original input chunk *(ii)* Applying ξ to the original chunk in order to have permutated data. Both processes need heavy computation and we can accelerate *DPM* by processing both on *GPU*. However, we face several challenges on both processes when we implement DPM in parallel. The following sub-sections explain these challenges as well as a possible solution for each.

A. Generating $\boldsymbol{\xi}$

The original content which is an input to *DPM* comes from different sources and it depends on the type of application where *DPM* is employed. For instance, [7] employed *DPM* for a database system management system where the input is a set of database queries; [2] employed *DPM* for protecting data privacy of *JPEG* files and each chunks is composed of one or multiple Minimum Coded Unit (MCU) blocks; and for healthcare electronic systems [16] where data privacy plays a key role.

The nature of generating a set of ξ is a sequential process that stands against data parallelism. We consider $\mathcal{M}$ as a *2D-array* for generating *PRP* addresses in parallel that allows us to use each GPU core to generate different sets of ξ. Each GPU core is corresponding to partial part of ξ. We map the original input ($\mathcal{D}$) to a *2D-array* ($\mathcal{M}$) to maximize the usage of *GPU* cores. Then, we apply ξ to $\mathcal{M}$ in the next step. Figure 1 shows an example of mapping from the original input ($\mathcal{D}$) to a *2D-*

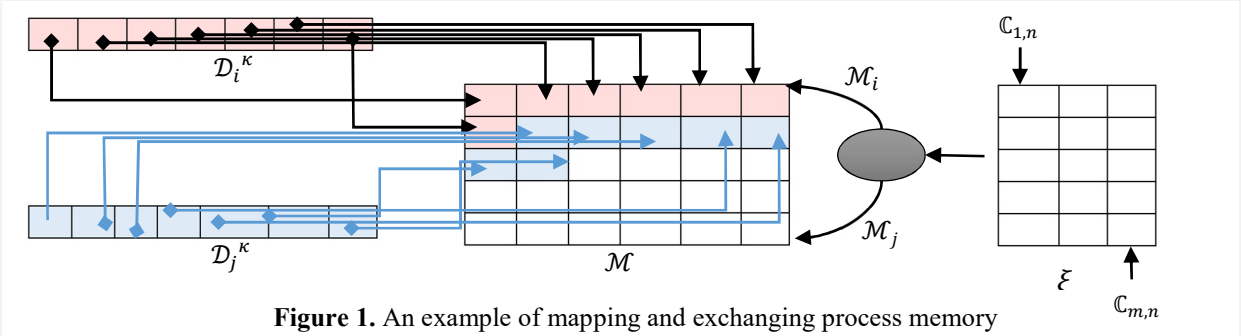


Figure 1. An example of mapping and exchanging process memory

array ($\mathcal{M}$). In this figure, $\mathcal{M} = \bigcup_{i=0}^{\delta} D^{\kappa}$ where δ is the maximum number of chunks for the original content/file and κ is the size of each chunk. ξ generates n set of ξs for m chunks.

Code I shows how each thread get same seed with different sequence numbers. `RowCell` and `ColCell` represent the number of rows and columns of $\mathcal{M}_{m,n}$, respectively. This configuration provides the best performance because each block of threads receives a unique initial seed and each block provides a unique set of ξ [7]. During the initialization (*config* function) `ColCell` is considered as a parameter of application. Let's `RowCell` and `ColCell` be different size of $\mathcal{M}$. In our experiment (see Section 7), we describe different configurations for the application in order to implement different size of $\mathcal{M}$.

Code II shows an implementation of generating of `RowCell` and `ColCell` of the *PRP* generator function in the *main()* function. In this code, *Line 1* allocates space for results on host. Space allocation for results on device is defined in *Line 2-4*. The ξ set is configured in *Line 4* and *PRP* generator is called in *Line 6*.

B. Apply ξ to $\mathcal{M}$

When random addresses have been generated in previous section and it is stored in ξ, then different solutions are available for applying ξ to $\mathcal{M}$ as follows:

- The first option is transferring ξ to the host-memory and shuffle $\mathcal{M}$ on host-memory: This process needs $O(n)$ where n is the length of ξ. However, the computation on shuffling process of data on host-memory cannot be implemented in parallel on device.
- The second option, is transferring $\mathcal{M}$ to device memory, and shuffle $\mathcal{M}$ based on ξon device memory. Although this process still needs $O(n)$ computation time where n is the length of ξ. This process can be implemented in parallel on device that accelerate shuffling process of data.

Let's consider the second option for the implementation of applying ξ to $\mathcal{M}$. In this case, there are two elements of $\mathcal{M}$ that needs to be exchanged when the *PRP* shows that an exchange is required between two κ-bits, $\mathcal{M}_i$where it is the original address and $\mathcal{M}_j$ where it is the destination address for the exchange. If we consider each element of $\mathcal{M}$ as a κ bit element, then minimum memory requirement for the implementation is κ. When *DPM* needs to exchange the content of two bits of $\mathcal{M}$ ($\mathcal{M}_i$ and $\mathcal{M}_j$) based on ξ, one of the following possibilities can raise: (i) $\mathcal{M}_i = \mathcal{M}_j$: In this case, if we assume that $\kappa = 1$ then no exchange is required because the content of both bits are the same and we can save the computation overhead of exchanging the content. If $\kappa > 1$ then the κ-bits of the content from $\mathcal{M}_i$ should be equal

```
// two parameters of the size of Sai (M:
RowCell * ColCell)
const int RowCell = 128;
const int ColCell = 128;
__global__ void config(curandState *state)
{
  int id = threadIdx.x + blockIdx.x * ColCell;
  curand_init(1234, id, 0, &state[id]);
}
```

Code I. The *config* function for initialization of a thread

```
1: hostResults = (unsigned int *)calloc(RowCell * ColCell, sizeof(int));
2: cudaMalloc((void **)&devResults, RowCell * ColCell *sizeof(unsigned int));
3: cudaMemset(devResults, 0, RowCell * ColCell *sizeof(unsigned int));
4: cudaMalloc((void **)&devStates, RowCell * ColCell * sizeof(curandState));
5: config << <RowCell, ColCell >> >(devStates);
6: DPM_PRP << <RowCell , ColCell >> >(devStates, Count, devResults);
```

Code II. Main function for *Calling* the *PRP generator*

to all κ-bits of $\mathcal{M}_j$. (ii) $\mathcal{M}_i \neq \mathcal{M}_j$: The exchange is required. In this case $\mathcal{M}_i'$ and $\mathcal{M}_j'$ are the final value of the exchange process on $\mathcal{M}_i$ and $\mathcal{M}_j$ after exchange process, respectively. Table I. summarizes these two different conditions.

Table I. The summarize of exchange process

$\mathcal{M}_i$	$\mathcal{M}_j$	$\mathcal{M}_i'$	$\mathcal{M}_j'$	Exchange is required
0	0	0	0	N
0	1	1	0	Y
1	0	0	1	Y
1	1	1	1	N

We also optimize the implementation of shuffling process of the two different condition. The following rules are used in order to avoid additional host memory exchange computation:

$$\begin{aligned} \mathcal{M}_i' &= (\mathcal{M}_i \oplus \mathcal{M}_j) \oplus \mathcal{M}_i \\ &= (\mathcal{M}_i \oplus \mathcal{M}_i) \oplus \mathcal{M}_j \\ &= 0 \oplus \mathcal{M}_j \end{aligned} \tag{7}$$

$$\begin{aligned} \mathcal{M}_j' &= (\mathcal{M}_i \oplus \mathcal{M}_j) \oplus \mathcal{M}_j \\ &= (\mathcal{M}_j \oplus \mathcal{M}_j) \oplus \mathcal{M}_i \\ &= 0 \oplus \mathcal{M}_i \end{aligned} \tag{8}$$

Therefore, by using Equation 7 and 8, we do not need to run the exchange function in order to optimize the exchange function.

VI. EVALUATION OF PROPOSED METHOD

A. Experimental setup

We implemented the proposed method on a PC with CPU i7-4790, x64 based processor, device memory of 12 GB and a *GeForce GT 720* GPU with 1,001,000 memory clock rate (kHz), 64 bits Memory Bus, 16.016000 GB/s Peak Memory Bandwidth (GB/s), and 16GB memory. We used *NVIDIA GPU Computing Toolkit v7.0* and we profiled (logging of *NVIDIA* functions) the executions of the proposed method with *NVIDIA profile* v7.0.

In order to record each step of execution, we used *NVIDIA* profiler in the implementation code where it requires to be started or stopped by the following code:

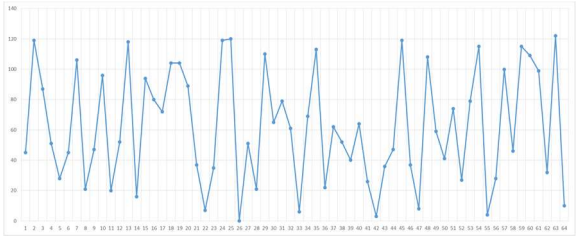
```
// start profiling of part of code
cudaProfilerStart();
// stop profiling of part of code
cudaProfilerStop();
```

By default, the first call of *CUDA API* starts the *profiler* (in this case `cudaGetDevice` initializes the profiler). An example of output of the profiler is shown in the List 1. In this example, the profiler shows that 10384 *API* calls (*CUDA API*) where 98.27% of time is taken in execution of *cudaMemcpy* which includes the following functions:

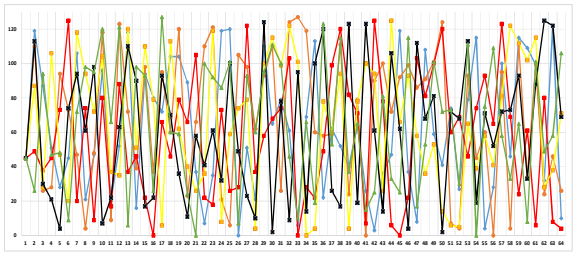
- *cudaMemcpyHostToHost*,
- *cudaMemcpyHostToDevice*,
- *cudaMemcpyDeviceToHost*, or *cudaMemcpyDeviceToDevice*.

B. Experimental Results

First, we consider the generator of ξ sets, and the distribution values for each set over different iterations.



2.a. The result of a set of ξ



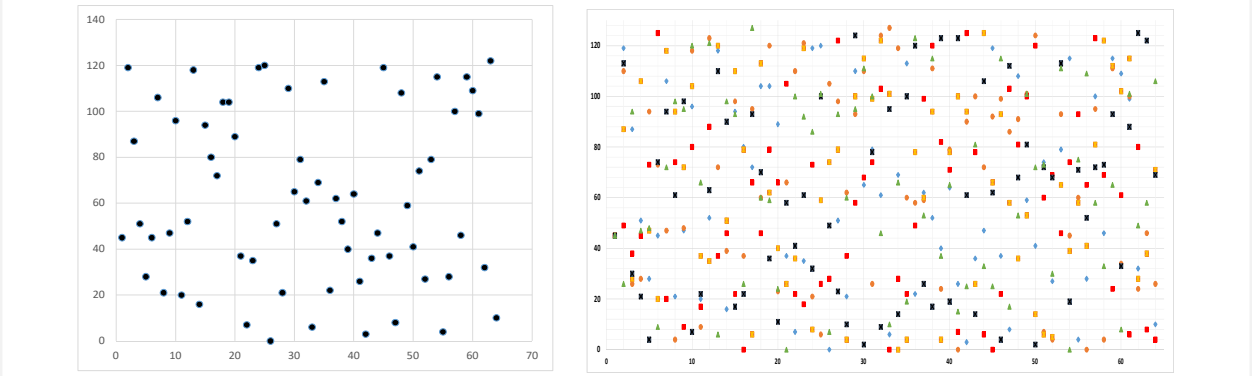
2.b. The result of six sets of ξ with different initial values

Figure 2. The evaluation results

```
==10384== NVPROF is profiling process 10384, command: calltester
==10384== Profiling application: calltester
==10384== Profiling result:
==10384== API calls:
```

Time(%)	Time	Calls	Avg.	Min.	Max.	Name
98.27%	7.18901s	3	2.39s	1.39s	4.12s	cudaMemcpy
1.54%	112.41ms	2	56.20ms	435.92us	111.97ms	cudaMalloc
0.18%	13.240ms	385	34.38us	2.85us	11.74ms	cudaLaunch
0.00%	313.33us	83	3.77us	0ns	147.11us	cuDeviceGetAttribute

List 1. The result of "nvprof calltester"



3.a. A set of ξ **3.b.** Six sets of ξ

Figure 3. Uniform distributions of ξ

These analysis allows us to evaluate the security of the method when the values have been generated with different *GPU*-cores. If there is a conflict between values of different sets, then it shows security issue in the proposed method. In the future, we plan to evaluate the generator with other statistical models, such as chi-square test of independence [17].

Figure 2.a shows a set of ξ and Figure 2.b shows six sets of ξ with different initial values. In these figures, each point represents a value of ξ, *X-axis* represents the iteration number and *Y-axis* represents the value of *PRP* function. These figures illustrate the proposed method generates different sets of ξ when each one does not introduce any conflict to other values of other ξ sets. Figure 3 shows the distribution variances of ξ where each set of ξ values is illustrated in Figure 2. As shown in this figure, the values of each set is uniformly distributed through the range of ξ values. As clearly shown in Figure 2.b and 3.b there is not any spot pattern or a cluster pattern that helps an attacker to estimate values by knowing values (partial values) of one or different ξ sets.

Overlap of different values from different sets of ξ is one of the security challenge for the proposed method. Each ξ provides a permutation model. An overlap between different value of ξ sets, allows an attacker to understand a permutation model by knowing one or multiple permutation models. Another challenge is finding a pattern between different subsets of ξ. As shown in Figure 3, different curves do not show similar patterns in any curves even for one set of ξ.

Second, we considered the performance of the proposed method by profiling the behavior of *DPM* for: (i) generating different sets of ξ on different size of a 2D-array; and (ii) permutation process. As shown in Figure 4 and 5, we assessed the performance of generating ξ values with the following sizes: 32*64, 64*64, 64*128, 128*128. X-axis represents the size of input. Y-axis in Figures 4.a illustrates the number of calls. Y-axis in Figures 4.b-d represent the computation time (*milliseconds*). Figure 5 shows the behavior of *CudaMemcpy* where it is responsible for transferring data from CPU to *GPU*. As shown in this evaluation, the blue curve indicates that it is increased linearly. As a result, the proposed method is capable to increase the size of input data with minimal transferring cost between *CPU* and *GPU*.

The evaluation results show that the 2D-array with the size of 128*128 provides better performance over other input sizes. However, the energy consumption is another parameter that can be assessed [18] in order to provide an overall result for this performance evaluation. This evaluation also provides an overall view of the performance of the proposed method.

VII. SECURITY ANALYSIS

In this section, we describe the security assumption and the level of security for the proposed method.

Let $SC(\mathcal{M})$ be the scramble function of *DPM* on n-core *GPU*. Perfect secrecy as described in Shannon theory [12] is the probability of two different encrypted messages and in our study, $SC(\mathcal{M}_i)$ and $SC(\mathcal{M}_j)$, which is defined as follows:

$$\forall m_0, m_1 \in M \; |m_0| = |m_1| \; and \; c \in C \tag{9}$$

$$(\Pr[SC(\xi_i, m_0) = c]) = (\Pr[SC(\xi_j, m_1) = c]) \tag{10}$$

where ξ_i and ξ_j are defined as different sets of *PRP* with different initial values, μ and P_0 is defined in Equation 5. M is a set of all original messages and C consists of permutated messages based on a set of ξ values.

Lemma 1: *DPM has perfect secrecy.*

To proof Lemma 1, we must proof the following sub-lemmas, *Lemma 1.1* and *Lemma 1.2* as follows:

Lemma 1.1: By a given c (scrambled data), the adversary cannot learn about m_i and m_j (two different original messages). Therefore, we must generate different outputs for all different inputs.

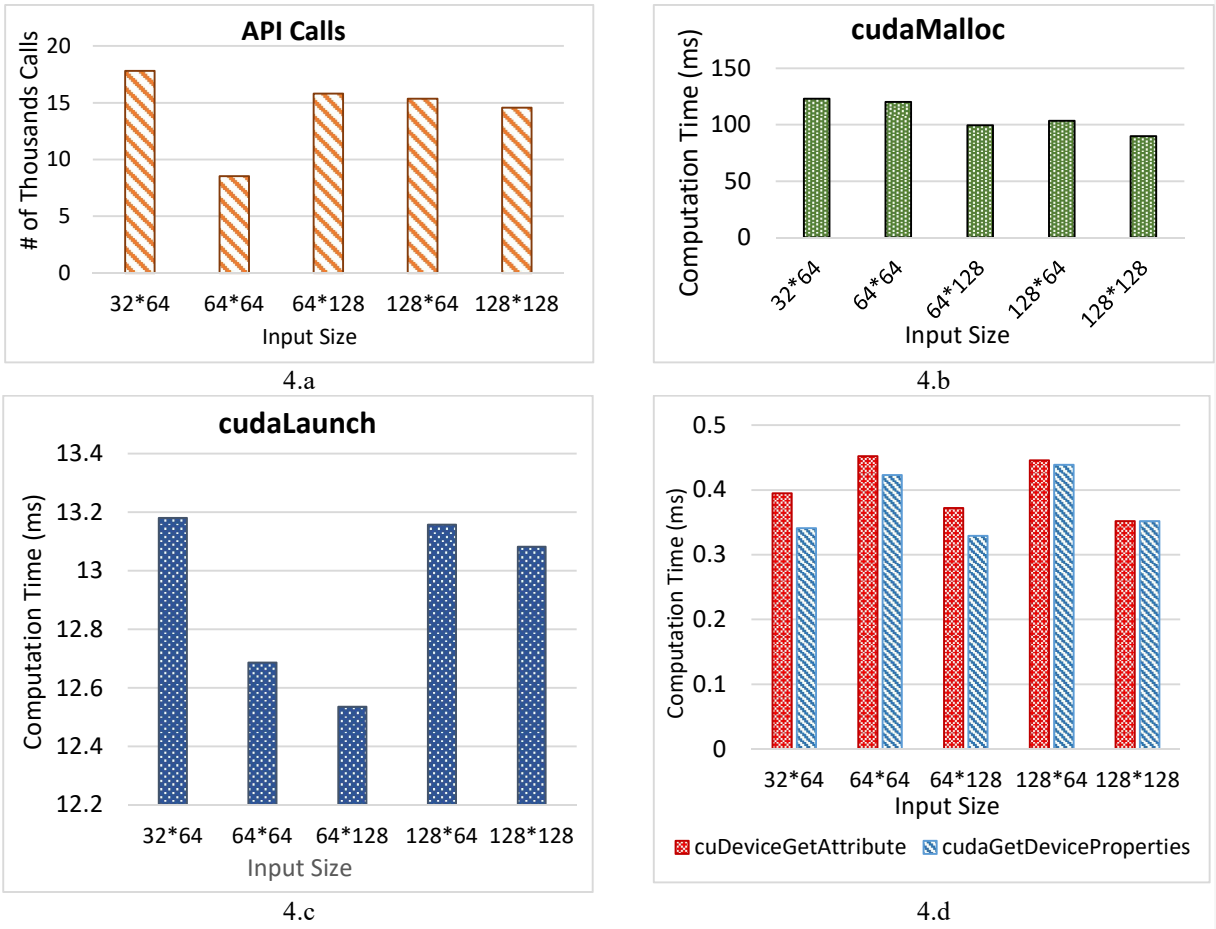


Figure 4. Evaluation Results for different size of ξ

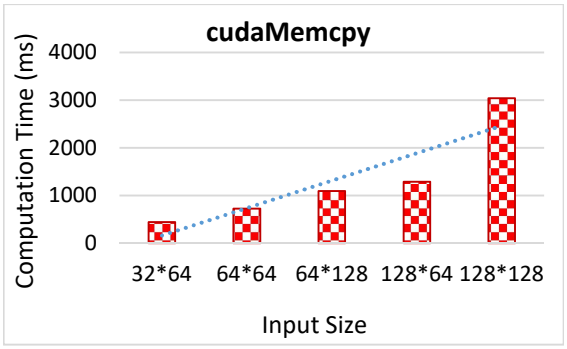


Figure 5. Performance of CudaMemcpy

Proof: Each separate original content in *DPM* should be scrambled with different sets of ξ to avoid similarity between $SC(\xi, m_i)$ and $SC(\xi, m_j)$. Each set of ξ is generated by $\mathbb{C}_m$GPU-core independently. Each core uses different initial values to generates different ξ sets without any conflict with other ξ sets, or with minimal partial conflict to other sets.

$$\forall File_i, c: Pr_{\zeta}[SC(\zeta, Content_i) = c] = \frac{\#\zeta \in \mathrm{Z} \text{ such that } \mathrm{SC}(\zeta, Content_i) = c}{|\mathrm{Z}|}$$

Since the intitalization value of each ξ is different for each GPU-core (the security assumption), then an attacker by accessing to the scrambled content is not able to learn about m_i and m_j, if and only if the attacker cannot learn about sequence of ξ values which means the attacker should not have knowledge of parameters of ξ generator. As our evaluation of generated *PRP* shows in Figure 2 and 3, then the attacker is not able to learn about m_i and m_j by accessing to c. ■

Lemma 1.2: The ξ generator has perfect secrecy for all *GPU* cores.

Proof: The *PRP* must provide a uniform distribution for all entries of *n bits* as follows:

$$P: U \to [0,1] \ \ such\ that \ \sum_{x \in U} P(x) = 1$$

where $U = \{0,1\}^n$.

$$\forall\, x \in U: P(x) = \frac{1}{|U|}$$

Since each GPU-core generates an unique set of ξ values, then the probility of all ξ sets are equal and $P(x) = \frac{1}{|U|}$ for each GPU-cores satisfied the generator condition of perfect secrecy. ■

VIII. CONCLUSION AND FUTURE WORK

Cloud computing offers new oportutanties to users to efficiently outsource data and applications. Data privacy is one of the major issues in cloud computing systems. In our previous study, we introduced a light-weight data privacy method (*DPM*) that allows users to prototect their data before submitting original file to the cloud. *Graphic Process Units (GPU)* allows parallel processes to be run efficiently. *GPU* kernel is able to process computationally intensive tasks on client side by using a *GPU* platform, such as *NVIDIA CUDA Toolkit*.

In this paper, we introduced a solution to mobile cloud users to accelerate *DPM* on multicore *GPUs*. This study shows that *DPM* can be implemented securly and efficiently on multiple independent *GPU*-cores. The proposed method protects users data privacy by processing independent pesudo-random number generator on each core when it is complying with perfect secrecy requirements. We evaluated the proposed method by performing rigorous assessments on performance and the security. On performance side, we ran different number of parallel processes in order to assess the computation time on each input size. We implemented the proposed method when it is being parallelized on a 2D-array of parallel processes where each thread block assigned by different initial values to generate different and unique pesudo-random numbers. The generated numbered are used for permutation of an original file. On security side, we considered the security assumption of the method and we assessed the result of pseudo-random numbers, distribution of this random numbers and perfect security assessments to analysis the security of the proposed method on multiple *GPU* cores.

We plan to asses the performance of the proposed method with different platforms that can be implemented on different *GPU* architectures. For instance, we will implement the proposed method by using *OpenMP* that help us to evaluate and to compare the current performance against other *GPU* architcetures/platforms. We will also invastigate the energy consumption [18] of the method on different *GPU* platforms and architectures.

REFERENCES

[1] Mehdi Bahrami and Mukesh Singhal, “The Role of Cloud Computing Architecture in Big Data”, Information Granularity, Big Data, and Computational Intelligence, Vol. 8, pp. 275-295, Chapter 13, Pedrycz and S.-M. Chen (eds.), Springer, 2015 http://goo.gl/0LxxlH

[2] Mehdi Bahrami and Mukesh Singhal, "A Light-Weight Permutation based Method for Data Privacy in Mobile Cloud Computing" in 2015 3rd Int. Conf. IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (IEEE Mobile Cloud), San Francisco, IEEE, 2015.

[3] Manavski, Svetlin. "CUDA compatible GPU as an efficient hardware accelerator for AES cryptography." Signal Processing and Communications, 2007. ICSPC 2007. IEEE 2007.

[4] Shao, Fei, Zinan Chang, and Yi Zhang. "AES encryption algorithm based on the high performance computing of GPU." Communication Software and Networks, 2010. ICCSN'10. Second International Conference on. IEEE, 2010.

[5] Li, Qinjian, et al. "Implementation and analysis of AES encryption on GPU." High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on. IEEE, 2012.

[6] Han, Zhang, et al. "A new image encryption algorithm based on chaos system." Robotics, intelligent systems and signal processing, 2003. Proceedings. 2003 IEEE international conference on. Vol. 2. IEEE, 2003.

[7] Mehdi Bahrami and Mukesh Singhal, "CloudPDB: A Light-weight Data Privacy Schema for Cloud-based Databases", 2016 IEEE International Conference on Computing, Networking and Communications, Cloud Computing and Big Data, Kauai, Hawaii, Feb 2016.

[8] Katz, Jonathan; Lindell, Yehuda (2007). Introduction to Modern Cryptography: Principles and Protocols. Chapman and Hall/CRC.

[9] Wang, Wei, et al. "Accelerating fully homomorphic encryption using GPU." High Performance Extreme Computing (HPEC), 2012 IEEE Conference on. IEEE, 2012.

[10] Tsoi, Kuen Hung, K. H. Leung, and Philip Heng Wai Leong. "Compact FPGA-based true and pseudo random number generators." Field-Programmable Custom Computing Machines, FCCM 2003. 11th Annual IEEE Symposium on. IEEE, 2003.

[11] Thomas, David Barrie, Lee Howes, and Wayne Luk. "A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation." Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays. ACM, 2009.

[12] C.E. Shannon, “Communication Theory of Secrecy Systems", Bell System Tech. J., Vol. 28, 1949, pp. 656-715.

[13] Killmann, W., Schindler, W.: AIS 31: Functionality Classes and Evaluation Methodology for True (Physical) Random Number Generators, version 3.1, Bundesamt für Sicherheit in der Informationstechnik (BSI), Bonn (2001)

[14] Blum, Lenore, Manuel Blum, and Mike Shub. "A simple unpredictable pseudo-random number generator." SIAM Journal on computing 15.2 (1986): 364-383.

[15] Manssen, Markus, Martin Weigel, and Alexander K. Hartmann. "Random number generators for massively parallel simulations on GPU." The European Physical Journal Special Topics 210.1 (2012): 53-71.

[16] Mehdi Bahrami, and Mukesh Singhal. "A dynamic cloud computing platform for eHealth systems" 2015 17th International Conference on E-health Networking, Application & Services (HealthCom). IEEE, 2015.

[17] McHugh, Mary L. "The chi-square test of independence." Biochemia Medica 23.2 (2013): 143-149.

[18] Huang, Song, Shucai Xiao, and Wu-chun Feng. "On the energy efficiency of graphics processing units for scientific computing." Parallel & Distributed Processing, IPDPS 2009. International Symposium on. IEEE, 2009.