www.czasopisma.pan.pl PAN www.journals.pan.pl
POLSKA AKADEMIA NAUK
jet INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2019, VOL. 65, NO. 1, PP. 25–31
Manuscript received August 23, 2018; revised January, 2019. DOI: 10.24425/123561

Weak RSA Keys Discovery on GPGPU

Przemysław Karbownik, Paweł Russek, and Kazimierz Wiatr

***Abstract*—We address one of the weaknesses of the RSA ciphering systems *i.e.* the existence of the private keys that are relatively easy to compromise by the attacker. The problem can be mitigated by the Internet services providers, but it requires some computational effort. We propose the proof of concept of the GPGPU-accelerated system that can help detect and eliminate users' weak keys. We have proposed the algorithms and developed the GPU-optimised program code that is now publicly available and substantially outperforms the tested CPU processor. The source code of the OpenSSL library was adapted for GPGPU, and the resulting code can perform both on the GPU and CPU processors. Additionally, we present the solution how to map a triangular grid into the GPU rectangular grid – the basic dilemma in many problems that concern pair-wise analysis for the set of elements. Also, the comparison of two data caching methods on GPGPU leads to the interesting general conclusions. We present the results of the experiments of the performance analysis of the selected algorithms for the various RSA key length, configurations of GPU grid, and size of the tested key set.**

***Keywords*—cryptography, Internet security, intrusion prevention, accelerated computing, caching methods, big numbers computing**

I. Introduction

CRYPTOGRAPHIC algorithms used in practice become unsafe when they are not properly engaged. In some situations, the vulnerabilities become detected a long time after the algorithm came into common service. Susceptibility of the solution may lay in the cryptographic algorithm or software; nevertheless, providers of the computer infrastructure should commence certain steps to protect users' privacy. Therefore efficient tools should be developed to help mitigate such security deficiencies. In this document, we address above approach and report our efforts to built proof of concept of a system for weak cryptographic keys detection.

The Rivest, Shamir, & Adleman (RSA) algorithm was introduced in 1977, and it is today the most popular asymmetric cryptographic algorithm. To secure communication, it uses a pair of keys: the public key for message coding, and private key for decoding. The protection delivered by RSA comes from the complexity of big numbers factorization. In 2012, Lenstra *et al.* [1] published the results of the check of the openly accessible public RSA keys, and they found that the 0.2% of keys failed to provide the expected level of security. One of the highlighted problems was that the modulus components of some public keys shared common prime factors. Thus, they could be easily factorized compromising the keys. The source of the problem is not in the RSA algorithm but in the fact the keys were generated incorrectly *i.e.* by malfunctioning prime generators that do not have enough entropy and output same numbers repeatedly.

Identifying the keys with moduli that share factors involves collecting a big set of public keys and performing a greatest common divisor (GCD) search for all pairs of the keys. The efficient algorithm to perform bulk GCD computation was, however, proposed by Bernstein [2], and it was used by Heninger *et al.* [3] to identify factorable public keys for $\approx$ 0.50% of network hosts. Heninger's approach invoked CPU that was an apparent approach for the control-intensive algorithm of Bernstein. The naïve approach involve doing GCD search for each distinct key pairs independently, and it more suitable for accelerated computing [4] [5].

Our interest in General-Purpose Computing on Graphics Processing Units (GPGPU) was the main reason we had chosen this platform to propose the solution for the factorable RSA keys detection. The practice of accelerated computing leads to the selection of the algorithms that are regular and display a clear outer loop; therefore, we have identified the naïve procedure as a natural way to attack the problem. A similar concept was earlier exploited by Scharfglass *et al.* [6] and Fujita and Koji [7]. Interestingly, the solutions and conclusions that are presented in this paper unfold beyond the problem of weak RSA key discovery, and it applies to the processing of integer pairs when they belong to the large set of big numbers. It should be also mentioned that authors do not come across any GPU implementation of Bernstein's algorithm in literature, and this method is perhaps their future endeavour for the GPGPU.

II. Problem Statement

In this section, we provide a simple background to the problem of the weak RSA key discovery. The RSA algorithm will be outlined, and the tree versions of the Euclidian algorithm for GCD search will be proposed.

A. *RSA algorithm*

The RSA algorithm is a public-key (asymmetric) cryptographic scheme used to secure communication between the two foreign parties. Before the asymmetric cryptography was first introduced in 1976 [8], a private communication relied on a secret (key) shared by the communication peers. In so-called symmetric cryptography the same key is used for the encryption and decryption process; therefore, the key has to be exchanged in privacy before communication starts. In

Paweł Russek has been supported by AGH University of Science and Technology Statutory Activity, grant no 11.11.230.017. Kazimierz Wiatr has been supported by National Science Centre (NCN), grant no DEC-2011/01/B/ST6/03024.

Przemysław Karbownik graduated from the Faculty of Computer Science, Electronics and Telecommunication at AGH Univeristy of Science and Technology, Cracow, Poland (przemyslaw.karbownik@gmail.com).

Paweł Russek and Kazimierz Wiatr work at AGH UST in Department of Electronics at Faculty of Computer Science, Electronics and Telecommunication AGH; and in Academic Computing Centre 'Cyfronet' AGH, Poland (russek@agh.edu.pl, wiatr@agh.edu.pl).

opposite, the security of the public key cryptography systems rely on mathematical problems that currently admit no efficient solution.

The computational difficulty of a big number factorization is a foundation of the RSA system. The tree basic procedures of RSA are given in Algorithm 1. In RSA, a communication party creates the private and public keys using the *KeysGeneration* procedure. It publishes the public key and keeps the private key in secret. Later, it uses the private key to decrypt data that has been encrypted by other party with the public key; *i.e.* data can be read by the private key owner only.

Mathematical operations in RSA involve modular arithmetic, where numbers "wrap around" upon reaching a certain value. For example, the result of $a * b$ mod N is the residue of $\frac{a*b}{N}$. As it can be seen in the *RSAencode* procedure (Alg.1), message encoding is a modulo exponentiation operation with the public key exponent e and modulus n. The modulo operations are not trivially reversible, as one cannot easily perform the backward trial. Still, the *RSAdecode* procedure allow the private key owner to decode the message m by c^d mod n operation, where c is the cipher text, d is the private key exponent, and n is modulus. The RSA works because d is the modular inverse of e (mod $\varphi(n)$), and both d and $\varphi(n)$ are unknown to the attacker as long as n is not factorized ($n = p * q$). The modulus n is a big number and its bit-width determines the RSA key length *e.g.* 1024-bit keys can be used in practice.

Algorithm 1 RSA algorithm

```
1:  procedure KEYSGENERATION
2:      $p, q \leftarrow$ big prime numbers
3:      $n \leftarrow p * q$
4:      $\varphi(n) \leftarrow (p-1) * (q-1)$ ▷ calculate the value of the Euler function for $n$
5:      $e \leftarrow$ any coprime number of $\varphi(n)$; $1 < e < \varphi(n)$
6:      $d \leftarrow e^{-1} \pmod{\varphi(n)}$ ▷ i.e. $e * d \pmod{\varphi(n)} \equiv 1$
7:      $privateKey \leftarrow (n, d)$; $publicKey \leftarrow (n, e)$
8:      return $\{privateKey, publicKey\}$

9:  procedure RSAENCODE($messageText, publicKey$) ▷ $publicKey : (n, e)$
10:     $m \leftarrow$ convert $messageText$ to a number; $m < n$
11:     $c \leftarrow m^e (\text{mod } n)$ ▷ c is a ciphertext
12:     return $c$

13: procedure RSADECODE($messageText, privateKey$) ▷ $privateKey : (n, d)$
14:     $m \leftarrow c^d (\text{mod } n)$ ▷ $c^d \equiv (m^e)^d \equiv m$ (mod n)
15:     $messageText \leftarrow$ restore $messageText$ from $m$
16:     return $messageText$
```

B. Euclidian algorithm

The use of the poor prime number generator leads to imperfect randomness in the selection of p and q. Consequently, in a large set of the public keys, an attacker can detect pairs of keys whose moduli share the factor. In order to find the common factor of the moduli A and B, one has to use a GCD algorithm. For a pair of numbers, various versions of the Euclidean GCD algorithm exist. In the classic Euclidean method, one performs consecutive subtract operations. It should be noticed that gcd of A and B; where $A > B$, is similar to the gcd of $A - B$ and B. The graphical justification of this property is given in Figure 1. Thus, one can substitute A with $A - B$ and still keep the final result. The Euclidian algorithm is presented in Figure 2a. At each step, we subtract the smaller value from the larger one, and the subtraction continues until one of the values is zero.

The classic GCD performs poorly because it needs many iterations; however, it can speed up significantly if another property of gcd is observed: when A is even and $A > B$, gcd of A and B is also gcd of $\frac{A}{2}$ and B. The division operation makes the algorithm reaches zero faster. Additionally, in processor; the division by two can be substituted by fast and simple arithmetic shift right operation. While designing the efficient GCD procedure, it is also worth to remember that $gcd(\frac{A}{2}, \frac{B}{2}) = \frac{gcd(A,B)}{2}$, if both A and B are even.

The binary Euclidean algorithm, also known as Stein's algorithm [9], exploits all of the presented properties of GCD. We assessed different GCD methods for our experiments with the GPU processor. As a result, we have chosen three schemes for the exhaustive study: the classic Euclidean, binary Euclidean, and fast Euclidean algorithms. The classic Euclidean (Fig. 2a performs badly but it is a good point of reference. Selection of the other two methods was driven by their compelling performance.

The binary Euclidean, presented in Figure 2c, is inspired by Stein's algorithm; however, it features the clear-cut outer loop in our approach. The outer loop makes the candidate a good pick for the hardware acceleration. The diagram (Fig. 2c) plainly explains how the algorithm works. Please note the S variable that is doubled each time both A and B are even and, therefore, halved simultaneously. The S value is used to produce the final algorithm result: $S * A$.

The fast Euclidean algorithm, given in Figure 2b, can be seen as a simplification of the binary algorithm. Only the A variable is tested for its parity. This forces the algorithm to perform more loop iterations to complete when compared to its binary counterpart; but also, it makes its program code simpler and less control-intensive. In GPU's Single Program Multiple Data (SPMD) execution model, the reduction of the conditional statements in a code leads to the more efficient execution of the simultaneous threads. In SPMD, the execution of the conditional body of the threads that meets the condition force the rest of the threads to stall. This is the reason; for GPUs, we prefer programs that do not branch much. Therefore, the fast Euclidean algorithm is a good candidate for our experiments.

III. GPU IMPLEMENTATION

A. Mapping of the problem

In our GPU implementation, we assumed each thread to perform the GCD algorithm for one pair of the key moduli. The key pairs create a 2D grid with elements in rows and columns.

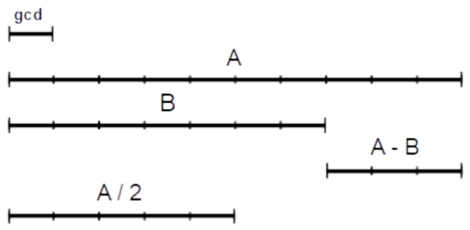


Fig. 1. Graphical justification of the Euclidian algorithm

Naturally, this grid can be seen as the GPU computing grid, and it is enough to assign one grid node to one GPU thread. However, the pairs are a combination of the two keys *i.e.* the order of the keys does not matter, and the pair {key1, key2} is similar to the pair {key2, key1} for example. Therefore, the active nodes form a triangle within the 2D grid. This poses some challenge in GPU implementation where threads form a rectangle grid. The problem is depicted in Figure 3. The pair of keys is denoted as (a, b) in this document, and only pairs where $a > b; a \in \{0, 1, \ldots\}$ are active during computation (they are filled in gray in Figure 3).

A basic approach to cover the triangular key grid with the thread grid is to use the square computing grid of size $n \times n$; where n is a number of keys in the set, and put the key triangle into the thread grid square. In this case, any GPU thread terminates immediately if $a \geq b$ – they are a dummy. This is not a good solution in our opinion, because the dummy threads; even if they do not compute, consume the GPU resources. Every thread requires a certain number of registers, and the GPU compiler has to allocate registers for them. The threads consume the shared memory space for their registers, and it limits the maximum number of the block threads, as the shared memory size is fixed for a given GPU. The higher number of threads per block is the main source of better GPU power utilization (see CUDA occupancy calculator [10]), and it is worth an effort to avoid the dummy threads that take resources but do not compute.

In the proposed method, we fit the key pairs into a rectangle that is half the size of the above square. Please note that the number of the active key pairs is

$$k = \frac{n * (n-1)}{2}. \tag{1}$$

To meet the quantity of the active key pairs (Eq. 1), the thread grid can be formed as the $(n-1) \times \frac{n}{2}$ rectangle, if n is even value; or the $\frac{n-1}{2} \times n$ rectangle, if n is odd value. Following this idea, we introduce the mapping of the key grid into the GPU grid that is given in Figure 4. For generality, let's introduce p variable, and $p = 1$ or $p = 0$ for the even and odd values of n respectively. Accordingly, the GPU grid size is

$$\frac{n-(1-p)}{2} \quad \times \quad n-p.$$

Figures 4a and 4b presents mapping for the even and odd n respectively. In short, the part of the triangle grid that lays outside the rectangle of the width $\frac{n-(1-p)}{2}$ (striped locations) is placed in the free rectangle area (white locations).

Now, we can use the 2D thread grid, where each GPU thread is denoted by its (x, y) coordinates. The formulas are necessary to determine the unique (a, b) pair of keys for the given (x, y) thread identifier. To get its dedicated key pair (a, b), the thread can use simple formulas:

$$a = \begin{cases} x + p & \text{if } (x+p) > y \\ k - x - 1 & \text{if } (x+p) \leq y \end{cases}$$

$$b = \begin{cases} y & \text{if } (x+p) > y \\ k - y - 2 + p & \text{if } (x+p) \leq y. \end{cases}$$

B. *Caching of the keys*

Looking at the grids in Figure 4, one sees the grid blocks that split the thread grid into regions. Along the grid size, the size of the block is an execution parameter of the GPU program. Same block threads can share data that is kept in the block's Shared Memory (SM). Threads read SM very fast; therefore, in the good GPU programming practice, it is used to cache the common block data. Usually during GPU program execution, data is first copied from the GPU Global Memory to SM, and then computation continues efficiently reading data from SM – this is often called arbitrary data caching. Unfortunately, only the threads executed within one block can share data kept in the same SM. Each grid block has its own SM, and its size is limited (*e.g.* GPU, used by the authors, offered 64 kB).

Studying the key pairs (a, b) associated with the nodes of the GPU block (x, y) (Fig.4), we conclude that threads in the same block process only a certain subset of the keys. If we neglect the fact that the part of the grid was moved to reshape the initial triangular grid; we can say that all threads in the same row share a – the first pair's key, and all threads in the same column use in common b – the second key of the pair. Thus, the corresponding key ranges for the block's columns and rows can be transferred to the block's SM prior to the major processing. Our case is a little bit different because the key grid has been re-arranged, and we has to consider the two different keys for each row and the two different keys for each column. This is, however, the case only for the blocks that span the area along the grid binding. One can employ the threads of the first and last block column to transfer two necessary ranges of a, and the threads of the first and last row to transfer two necessary b ranges. This is how the arbitrary caching was implemented in our experiment.

The second scenario, tested in the presented experiments, is the automatic caching. In GPU, the local memory can act also as a conventional L1 cache during program execution. In this case, data read from the Global Memory is automatically stored for reuse. The local memory is configurable to be adapted for Shared Memory, L1 Cache, or both – the local memory is divided between SM and L1 in different proportion (*i.e.* 16/48, 32/32, 48/16 kB, for our GPU).

Quite a lot of the local memory is necessary for the RSA keys analysis. For example; for 4096-bit keys, in the 32×32 blocks configuration, it is necessary to cache 64 kB of data $(2 \times (32+32) \times 4096$ bits) – this exceeds our GPU capabilities,

TABLE I
CHARACTERISTICS OF THE USED CPU AND GPU PROCESSOR

Processor	Intel Xeon E5-2630v3	Nvidia Kepler GK210
Release date	Q3 2014	Q3 2014
Technology	22 nm	28 nm
#Cores	8	2496
TDP	85 W	150 W
Clock freq.	2.4 GHz	875 MHz

as we should remember that the shared memory is also used to keep threads' register data.

IV. Tests and results

A. Experiment setup

Source codes of the OpenSSL library helped to prepare the software that was developed to perform presented experiments. Particularly, the procedures for big number arithmetic become the base for the RSA key processing. Nevertheless, we have introduced some modifications to the original OpenSSL functions to make them less control-intensive and, therefore, better suited for the GPU calculations. Also, scripts from OpenSSL library were used to create three sets of RSA keys (1024-bit, 2048-bit, and 4096-bit key sets).

We have implemented procedures for the three mentioned in Section II-B GDC algorithms: the original, binary, and fast. Significantly, the C-coded source code of the GCD and big number functions is common for the CPU and GPU platforms. The program was compiled by the 'nvcc' v9.0 compiler, and the CUDA framework with Compute Capability 3.5 was used.

The software, developed and used in the performance tests that follow, is available for the reader in the public Git repository [11].

The very first of our performance tests targeted the comparison of the GPU vs. CPU. Our choice of the computing platforms was done to make the judgment as fair as possible; but it was also determined by the particular hardware availability. Consequently, we used the acceleration platform accessible in the Academic Computing Centre 'Cyfronet'[12]. The selected server, based on the Intel Xeon E5-2630v3 CPU, hosted the Nvidia K80 GPU card. The main characteristics of the selected processors are summarized in Table I. Notably, both chips were released to the market at the same time; though, CPU has been fabricated in the more advanced semiconductor technology. The technology advantage of the Intel makes conclusions that favour the CPU architecture over the GPU distorted, but it is not the other way round. On the other hand, the perspective of the user who purchases processor is bond to the launch date rather than to the technology. What is also striking, the clock frequency of the CPU is ~2.7 times higher compared to the GPU, but its Thermal Design Power (TDP) – *i.e.* maximum power a device dissipates, is ~1.7 times lower. This is exactly the opposite one could expect, and this is perhaps a credit to both Intel's newer technology and the higher switching activity of the GPU gates.

V. Results

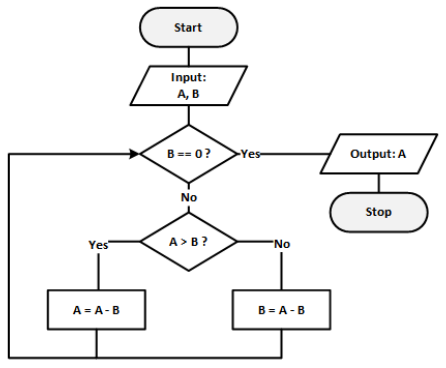
As a performance measure, we have decided to use the average execution time of the GCD for the single key pair. The experiments were conducted for various sizes of the key set: n. Accordingly,d the number of the key pairs p is given by Formula 1. The algorithm execution time T includes the GCD procedure only, and it excludes the key set construction – *i.e.* reading key text files from a disk storage, formatting, and loading data into the memory are not accounted. In the GPU case, we decided to start assessment when the data is already in the GPU global memory. As we copy n keys only, the host to card data transfer time is insignificantly short as compared to the compute time. Thus, the average execution time of the key pair (called *pair time*) is

$$t = \frac{T}{p}.$$

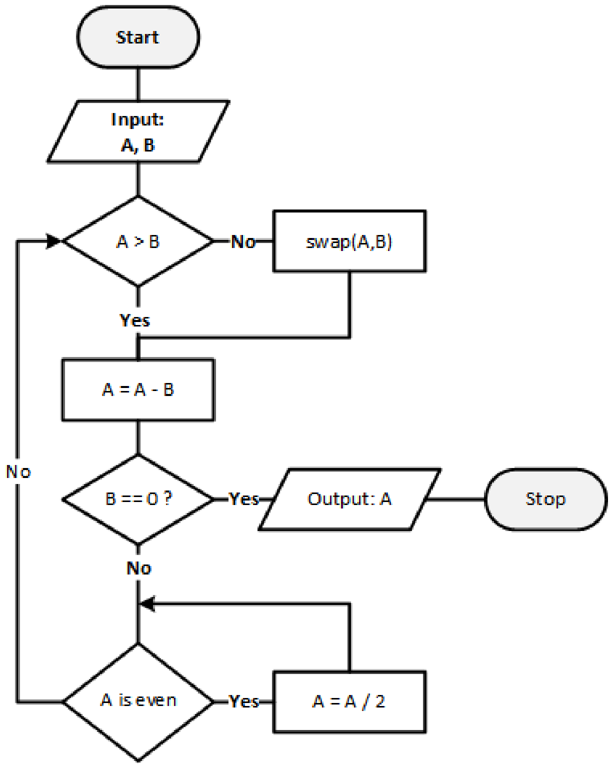
We have started the series of our experiments measuring the performance of the three GCD algorithms on the CPU and GPU; however, the original euclidian algorithm was used as an impractical reference only. Also, according to the GPU good practice, we maximized the block size and selected maximum value allowable for K80 *i.e.* 32 × 32 = 1024 threads per block. Two caching strategies for GPU (see Section III-B) were tried: the GPU-cache and GPU-shmem which denote the automatic and arbitrary caching respectively. The key size in the experiment was 1024 bit. As it can be seen in Figure 5, the binary and fast algorithms significantly outperformed the original one in all cases. Also, they work noticeably faster on the GPU than on the CPU. For the original algorithm, the gain of GPU vs. CPU is not that much. The reason, perhaps, is because no serious attention was paid to the efficient implementation of the original algorithm on GPU. Consequently, as we have found the GPU to be better suited for the weak key discovery, the rest of the presented experiments regard GPU only.

In the next experiment, the an a-priori assumption to maximize the block size was challenged. We performed the variable block size tests for the various algorithms with the three different key sizes (1024, 2048, 4096-bit). The results are given in Figure 6 and 7 for the automatic and arbitrary caching strategies. The fast conclusion can be drawn that the optimal block size is 4 × 4 = 16 threads per block. Only arbitrary caching with the 8 × 8 = 64 block size and 4096-bit key is an exception. The saturation of the execution time at the low value of threads per block is somehow confirmed by [10]. The reason originates in a high register utilization of our algorithms *i.e* 48 registers per thread. Such a high register requirement prevents full GPU utilization because the 'maximum number of active threads' (2048 in the case of CUDA 3.5) cannot be reached due to the local memory shortage – particularly $\#threads * \#registers$ has to fit the local memory register space. According [10], the 'maximum number of active threads' does not exceed 1280 for our kernels, and this value is reached for 98 threads per blocks only. Nevertheless, 16 threads per block proved to perform fastest in practice.

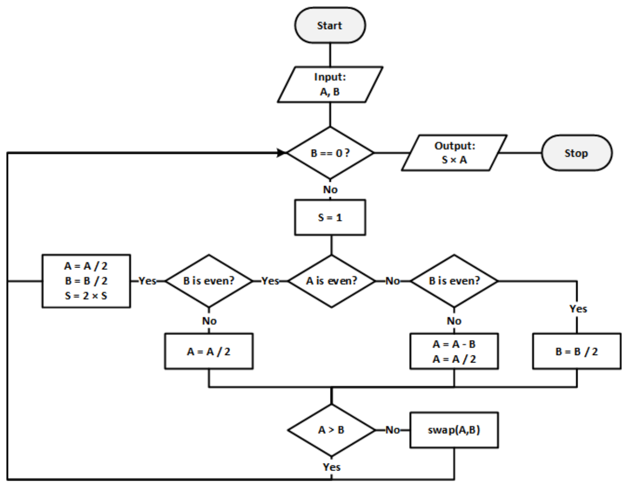
As it can be seen in Figure 7, the arbitrary caching experiment with 4096-bit keys ends at the 24 × 24 block size. The reason is that for the bigger block sizes the necessary space of the aggregated shared and register memory is higher than



(a) Euclidian algorithm



(b) Fast Euclidian algorithm



(c) Binary Euclidian algorithm

Fig. 2. Euclidian algorithms

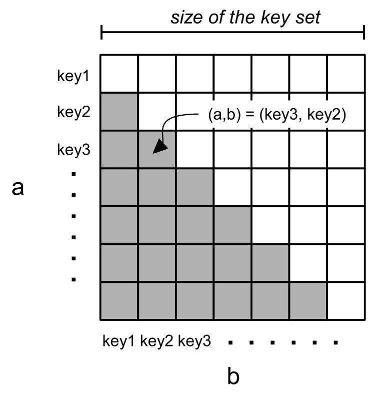
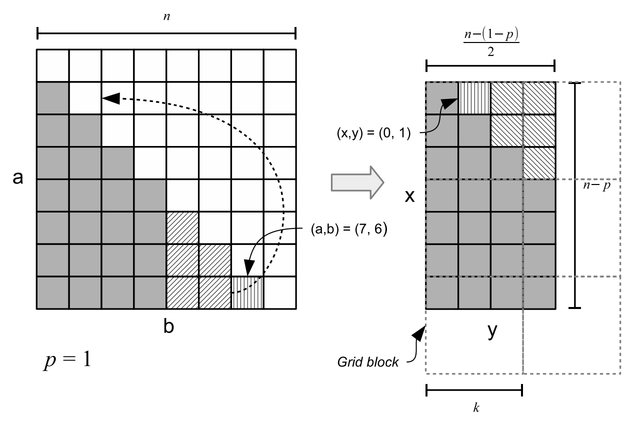
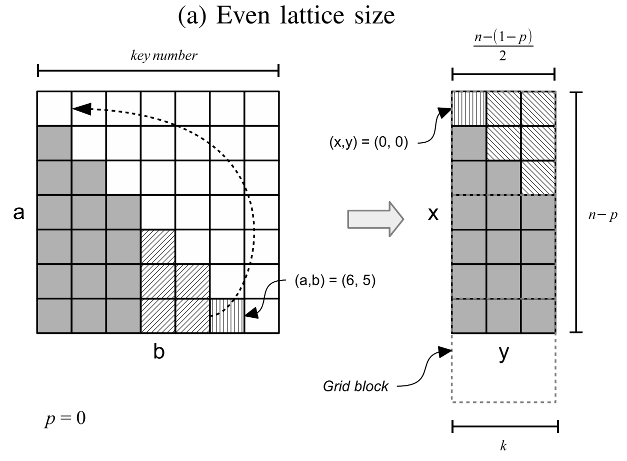


Fig. 3. The grid of the key pairs



(a) Even lattice size

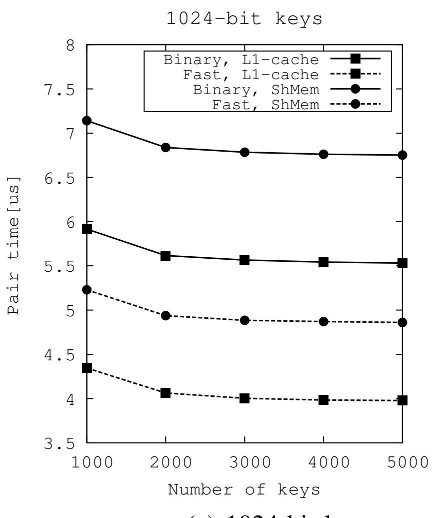


(b) Odd lattice size

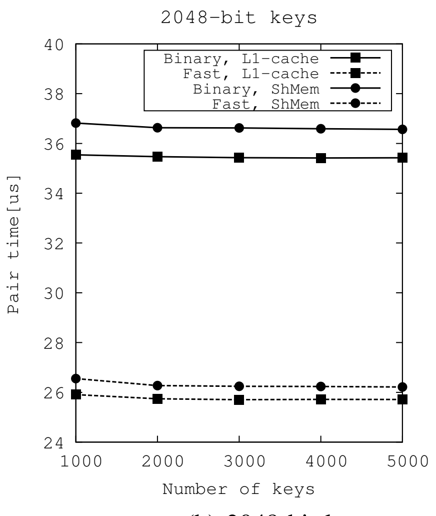
Fig. 4. Performance of the Euclidean algorithms; 16 (4×4) threads per block, different size of the key set

available 4,9152 Bytes of the local memory. The required size of shared memory for arbitrary caching is $2 * k * l$, where k is a block size, and l is a key length in bytes. In our experiment, $2 * 24 * 512 = 24,576$ Bytes of the shared memory is a limit that is reached for the 24 × 24 blocks when the key length is 4096 bits.

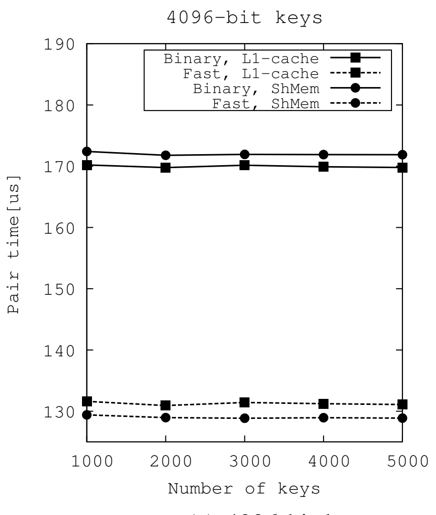
In the last examination, we compared an efficiency of the algorithms for various sizes of the key set. As indicated by the previous tests, the chosen blocks' dimension was 4 × 4. The obtained results, given in Figure 8, show that the maximum performance can be reached when the key set



(a) 1024-bit keys



(b) 2048-bit keys



(c) 4096-bit keys

Fig. 8. Performance of the Euclidean algorithms; 16 (4×4) threads per block, different size of the key set

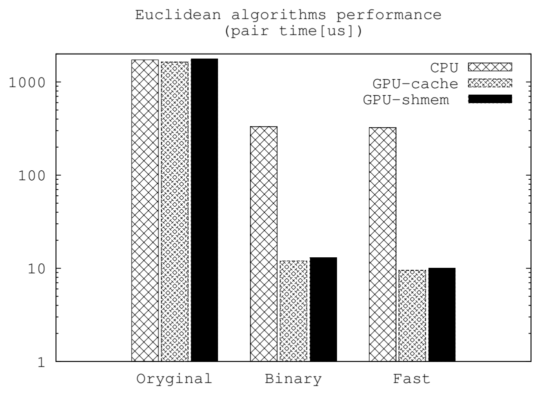


Fig. 5. Performance of the Euclidean algorithms on the GPU and CPU processors; 1024-bit key, 1024 threads per block

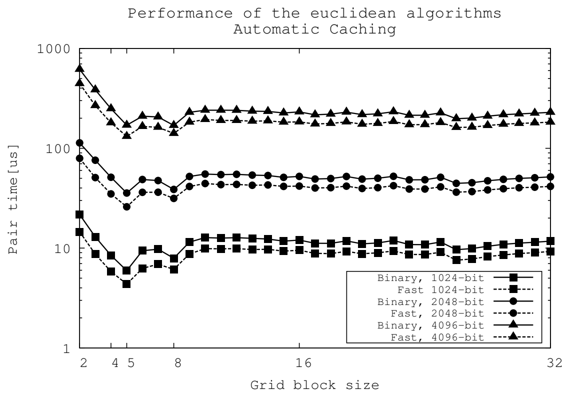


Fig. 6. Performance of the Euclidean algorithms on GPU for a different number of block threads. Automatic caching

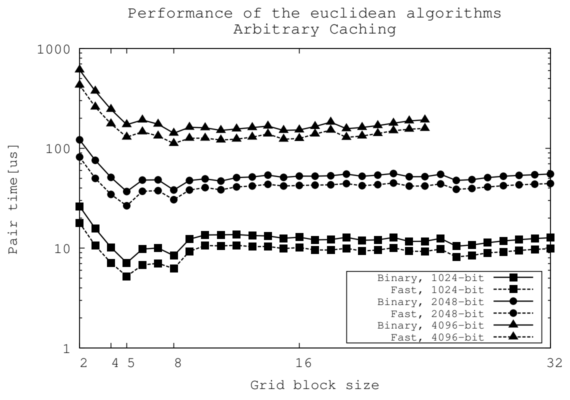


Fig. 7. Performance of the Euclidean algorithms on GPU for a different number of block threads. Arbitrary caching

contains more than 2000 keys. This applies to all tested cases (1024, 2048, and 4096-bit keys; binary and fast algorithms; arbitrary and automatic caching). For the 5000 keys and 4 × 4 blocks, the key times reached the values given in Table II. The configuration in this attempt provided the best pair times measured in all our experiments (denoted in bold).

TABLE II
EXECUTION TIME OF THE KEY PAIR FOR THE 4×4 BLOCKS AND THE SET OF 5000 KEYS (BEST VALUES ARE GIVEN IN A BOLD FONT)

Key length	Algorithm			
	Binary		Fast	
	Shmem	Cache	Shmem	Cache
1024-bit	6.8 μs	5.5 μs	4.9 μs	**4.0 μs**
2048-bit	36 μs	35 μs	26 μs	**25 μs**
4096-bit	171 μs	169 μs	**128 μs**	131 μs

According to Table II, the fast Euclidean algorithm outperformed the binary algorithm for all key lengths. Interestingly, the automatic caching performed significantly better (25% gain) than arbitrary caching for 1024-bit keys, and it performed near equally well for 2048-bit keys. This is a remarkable result because arbitrary caching is known to be the best choice in the GPU computing. To explain this phenomenon, we can notice, that in our arbitrary caching strategy the amount of reserved local memory is not optimal. Instead of two, we assume four subsets of keys for all grid blocks because some of the blocks are located across the seam (where the moved and unmoved nodes are adjacent). This strategy is not optimal for blocks that require only two key subsets (these located in moved or unmoved region only) and put automatic caching in a favorable position. For longer keys, arbitrary planned local memory allocation returns shorter execution times which agree with our educated guess.

VI. State-of-the-art and Contribution

The weak RSA key detection with GPGPU has been previously studied by Scharfglass *et al.* [6] and Fujita *et al.* [7]. Particularly, the second work reports GPU speed that outperforms our results. On the other hand, the former authors do not provide their source code; and, therefore, their solution cannot be further developed or simply used by others. We believe that approach to publish the program sources better serves scientific community. Our code, given in [11], can be further developed and improved by whoever wants to do it. For example, the big number libraries are a subject of an additional effort for GPU optimization in our opinion.

Additionally, our method that fits a triangular grid of pairwise computations into a rectangular GPU-like grid can be exploited beyond RSA key analysis. Also, adopted for the GPU, big number SSH library is also useful in wider context of computational problems *e.g.* prime number generation.

Moreover, our reports unveils interesting phenomenons where the automatic caching outperforms arbitrary one, and the smaller number of thread blocks outperforms the higher one. These results are worth dissemination to the GPU community as they revise the common practice.

VII. Conclusions

In our experiments, GPU has confirmed its dominance, over CPU, for integer data processing in easily parallelized and compute intensive tasks. The fast Euclidean algorithm has performed the best; although, in theory, it required more operations than the binary one. This confirmed the experience that GPU-oriented algorithms should pose the reduced number of conditional and branching instructions.

The GPU automatic caching was found to be an efficient mechanism that provides good performance with a simpler GPU code – no additional programmer effort is necessary to implement arbitrary caching in order to achieve significantly better performance results.

References

[1] A. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter, “Ron was wrong, whit is right,” IACR, Tech. Rep., 2012.
[2] D. J. Bernstein, “How to find smooth parts of integers,” *URL: http://cr. yp. to/papers. html# smoothparts. ID 201a045d5bb24f43f0bd0d97fcf5355a. Citations in this document*, vol. 20, 2004.
[3] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, “Mining your ps and qs: Detection of widespread weak keys in network devices.” in *USENIX Security Symposium*, vol. 8, 2012, p. 1.
[4] M. Wielgosz, G. Mazur, M. Makowski, E. Jamro, P. Russek, and K. Wiatr, “Analysis of the basic implementation aspects of hardware-accelerated density functional theory calculations,” *Computing and Informatics*, vol. 29, no. 6, pp. 989–1000, 2012.
[5] P. Russek and K. Wiatr, “The enhancement of a computer system for sorting capabilities using fpga custom architecture,” *Computing and Informatics*, vol. 32, no. 4, pp. 859–876, 2014.
[6] K. Scharfglass, D. Weng, J. White, and C. Lupo, “Breaking weak 1024-bit rsa keys with cuda,” in *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2012 13th International Conference on.* IEEE, 2012, pp. 207–212.
[7] T. Fujita, K. Nakano, and Y. Ito, “Bulk gcd computation using a gpu to break weak rsa keys,” in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International.* IEEE, 2015, pp. 385–394.
[8] W. Diffie and M. Hellman, “New directions in cryptography,” *IEEE transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.
[9] J. Stein, “Computational problems associated with racah algebra,” *Journal of Computational Physics*, vol. 1, no. 3, pp. 397–405, 1967.
[10] “Cuda occupancy calculator,” https://developer.download.nvidia.com, accessed: 2018-08-16.
[11] “Weak keys discovery git repository,” https://git.plgrid.pl/scm/p̃lgrussek/weak_keys_discovery.git, accessed: 2018-08-17.
[12] “Ack cyfronet agh,” http://www.cyfronet.krakow.pl, accessed: 2018-08-18.