Comparing low and medium cost computer-based technologies suitable for cryptographic attacks

V. Gayoso Martínez, L. Hernández Encinas,
A. Martín Muñoz, and O. Martínez-Graullera

Institute of Physical and Information Technologies (ITEFI)
Spanish National Research Council (CSIC), Madrid, Spain
`{victor.gayoso,luis,agustin}@iec.csic.es,oscar.martinez.graullera@csic.es`

Abstract. Initially developed for tasks related to computer graphics, GPUs are increasingly being used for general purpose processing, including scientific and engineering applications. In this contribution, we have compared the performance of two graphics cards that belong to the parallel computing CUDA platform with two C++ and Java multi-threading implementations, using as an example of computation a brute-force attack on Hitag2, a well known remote keyless entry application. The results allow us to provide valuable information regarding the compared capabilities of the tested platforms and to confirm that such a weak encryption system could be broken in less than a day with medium cost equipment.

Keywords: Cryptography, CUDA, C++, Encryption, Hitag2, Java, OpenMP

1 Introducción

In symmetric encryption algorithms, brute-force attacks consist in checking all possible keys until the correct one is found. In the worst case scenario, all the keys from the entire key space are tested, while in average it is necessary to check only half the number of possible keys.

Modern encryption algorithms are designed so that this kind of attack is infeasible, as the search for the key would take millions of years. Legacy algorithms were also designed with that goal in mind, with the difference that their designers could not anticipate the spectacular increase in computing capability that could be employed by organised groups or even individuals in such a task.

For that reason, we have considered that it is of interest to compare the computing capability provided by several platforms when using a brute-force approach on legacy algorithms such as Hitag2, which was introduced in 1996 but it still can be found in millions of devices [1]. Besides, the simplicity of Hitag2 makes it an ideal candidate for its implementation using CUDA, the parallel computing platform based on GPUs (Graphics Processor Units) created by NVIDIA. In addition to this, we decided to implement other versions of our brute-force application using Java and C++, so we could test their multi-threading capabilities, where in the case of the C++ implementation we have used OpenMP, a well-known multi-threading library. In this way, we have been able to check the latest improvements in Java regarding its performance and analyse how fast it is compared to a native C++ application. The results allow us to state that these implementations allow attackers to complete a brute-force attack against this kind of algorithms in less than a day using publicly available hardware (in our case, a medium tier CUDA card).

It must be clarified that, given the design of Hitag2, this stream cipher has been considered insecure for some years [2–5], and as such it can be attacked using expensive devices such as COPACOBANA [6]. Thus, our goal is not to show that Hitag2 is insecure, but to compare low and medium cost technologies that can be used for obtaining the encryption key with a sole computer in the scope of the protocol used by Hitag2.

In recent years there have been some studies developed by other researchers about implementing or even breaking cryptographic algorithms using CUDA technology (see for example [7–10]). However, none of those studies is focused on Hitag2. Besides, those studies do not compare CUDA implementations with the Java and C++/OpenMP versions of the algorithm being tested.

This contribution represents an extension of the work presented in [11], which studied the legacy encryption algorithm KeeLoq instead of Hitag2. In addition to that, in this study two CUDA cards (GTX 950 and GTX 1070) have been used instead of only one in order to extend the research to medium cost hardware.

The rest of this paper is organised as follows: In Section 2, we present a brief overview of the Hitag2 algorithm. Section 3 describes the CUDA, C++/OpenMP, and Java platforms used in the comparison. In Section 4, we offer a description of our two implementations for those platforms, including relevant code of the optimised CUDA version. The results obtained are located in Section 5. Finally, our conclusions are presented in Section 6.

2 Hitag2

2.1 Algorithm

Hitag2 is a stream cipher that consists of an internal 48-bit Linear Feedback Shift Register (LFSR) and a non-linear filter function f, as it can be observed in Figures 1 and 2. Hitag2 is the successor of Crypto1, another proprietary encryption algorithm created by NXP Semiconductors specifically for Mifare Radio Frequency Identification (RFID) tags.

In addition to the 48-bit key, this cipher uses a 32-bit serial number and a 32-bit Initialization Vector (IV). After a set-up phase of 32 cycles, the cipher works in an autonomous mode where the content of the register defines both the next encryption bit and how the register is updated. Thus, the total number of cycles is defined by the length of the bitstream that needs to be encrypted.

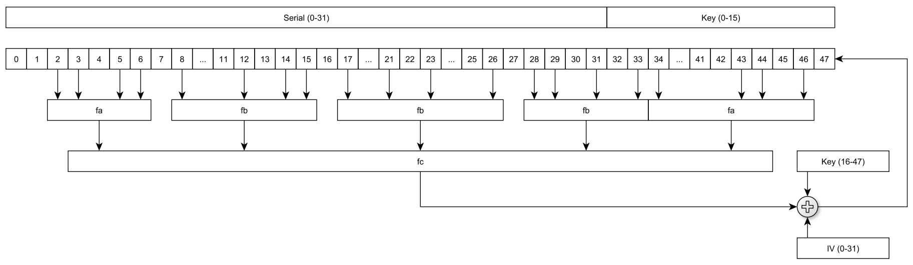

Fig. 1: Hitag2 initialisation phase.

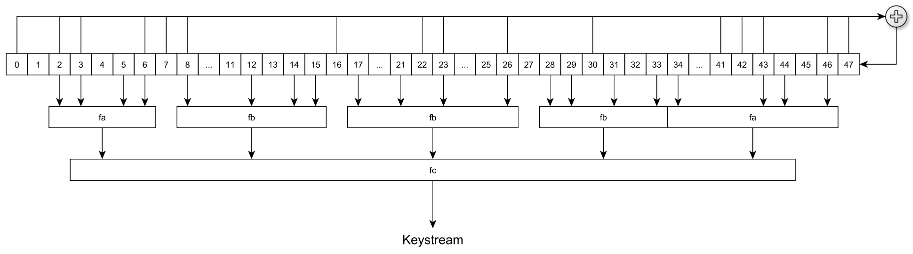

Fig. 2: Hitag2 operation phase.

The filter function f consists of three different functions f_a, f_b and f_c. While f_a and f_b take as input four bits and produce as output one bit, f_c uses five bits in order to generate the final result in the form of a single bit.

The three functions, which are used both in the initialization phase and the operation phase, can be modelled as boolean tables allowing easy implementations, so the output of those functions for the input i is the i-th bit of the values given below:

$$\begin{aligned} f_a(i) &= (\text{0x2C79})_i, \\ f_b(i) &= (\text{0x6671})_i, \\ f_c(i) &= (\text{0x7907287B})_i. \end{aligned}$$

In the initialisation phase (see Figure 1), the register is initially filled with the 32 bits of the serial number and the first 16 bits of the key. If the serial number is expressed as id_i $(0 \leq i \leq 31)$ and the key is expressed as k_i $(0 \leq i \leq 48)$, the register bits r_i $(0 \leq i \leq 47)$ adopt the following initial state:

$$\begin{aligned} a_i &= id_i \ (0 \leq i \leq 31), \\ a_{32+i} &= k_i \ (0 \leq i \leq 15). \end{aligned}$$

In each cycle, the bit generated by f_c is XORed with the corresponding bits of the IV and the key, generating a bit that is inserted in the register at position 47, shifting in the process the register one bit to the left. The new bit is computed according to the expression $f_c \oplus id_i \oplus k_{i+16}$, where $0 \leq i \leq 31$.

In the operation phase (see Figure 2), the new bit of the keystream is directly the output of f_c, while the bit inserted at the register at position 47 in each cycle is the result of the concatenated XOR operations $r_0 \oplus r_2 \oplus r_3 \oplus r_6 \oplus r_7 \oplus r_8 \oplus r_{16} \oplus r_{22} \oplus r_{23} \oplus r_{26} \oplus r_{30} \oplus r_{41} \oplus r_{42} \oplus r_{43} \oplus r_{46} \oplus r_{47} \oplus$.

In order to decrypt a data protected with this algorithm, the receiver needs to know both the key and the IV. Alternatively, decryption can be achieved by using the key and the result of XORing the values f_c and the IV during the initialisation phase, as it can be deduced from Figure 1. This feature will be used in our fine-tuned implementation.

2.2 Protocol

Hitag2 has been widely used in automotive Remote Keyless Entry (RKE) and Passive Keyless Entry (PKE) systems. An RKE system consists of an RF transmitter embedded into a car key that sends a short burst of digital data to a receiver in the vehicle, where it is decoded. In this context, users have to actively initiate the authentication process by pressing a button in their car key. The frequency used by RKE systems is 315 MHz in the US and Japan, and 433 MHz in Europe.

In comparison, in PKE systems users are able to automatically unlock their cars when they approach the vehicle without having to actively press any button, as a bidirectional communication takes place beetween the car key and the vehicle when the transmitter is within the system's range. PKE systems typically operate at the frequency of 125 KHz.

In the PKE protocol analysed in this contribution, which was reversed engineered and published online in 2008 [12], the communication between a reader (the vehicle) and a transponder (typically embedded in the car key) starts with the reader, which sends an authenticate command to the transponder, as illustrated in Figure 3.

Upon reception of this command, the transponder replies with a 32-bit message containing its serial number. Then, the reader generates a 32-bit IV and uses that value, together with the 48-bit key belonging to the transponder, in order to encrypt the hexadecimal value FFFFFFFF, which will be represented as $\overline{\text{FFFFFFFF}}$. Next, the reader sends that encrypted element together with the encrypted IV to the transponder (the encrypted IV, denoted as $\overline{\text{IV}}$, is the result of XORing the IV and the first 32 bits provided by the f_c function during the initialisation phase). If the transponder is able to recover the FFFFFFFF value from the two elements sent by the reader, then it will send as a reply to the

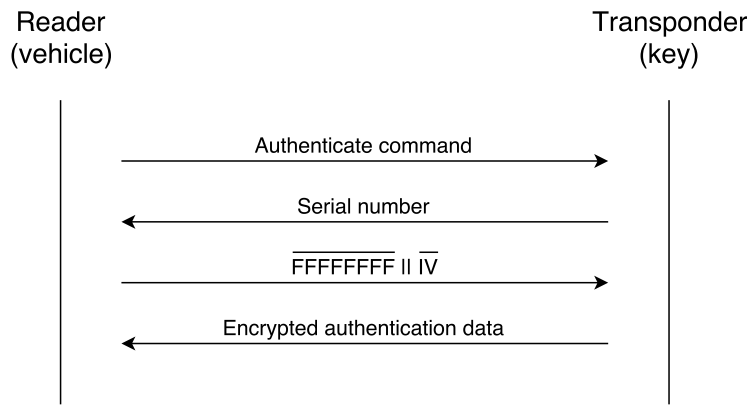
Reader
(vehicle)
Transponder
(key)
Authenticate command
Serial number
$\overline{\text{FFFFFFFF}} \parallel \overline{\text{IV}}$
Encrypted authentication data

Fig. 3: Hitag2 protocol.

reader some configuration bytes only known to both of them in encrypted form [1, 13]. In this way, both the reader and the transponder are authenticated by the other participant in the communication.

This protocol provides an easy attack scheme, as any eavesdropper is able to obtain both the plaintext and the ciphertext from the protocol's operation. Having access to the element $\overline{\text{IV}}$ is not a problem for the attacker, as that value can be directly used in the decryption.

As the number of keys is larger than the number of possible ciphertexts (48 bits versus 32 bits), an attacker will be able to compute many keys which convert the same plaintext into the same ciphertext. Thus, a brute force attack such as the one described in this contribution needs an additional step in order to correlate the keys obtained from several encryption pairs.

3 Programming platforms

3.1 CUDA

In last years, one of the dominant trends in microprocessor architectures has been the continuous increment of the chip-level parallelism and, as a result of that, multi-core CPUs (Central Processing Units) providing 8-16 scalar cores are now commonplace. However, GPUs have been at the leading edge of this drive towards increased chip-level parallelism, GPGPU being the term that refers to the use of a GPU card to perform computations in applications traditionally managed by the CPU. Due to their particular hardware architecture, GPUs are able to compute certain types of parallel tasks quicker than multi-core CPUs, which has motivated their usage in scientific and engineering applications [14]. The disadvantage of using GPUs in those scenarios is their higher power consumption compared to that of traditional CPUs [15].

CUDA is the best known GPU-based parallel computing platform and programming model, created by NVIDIA. CUDA is designed to work with C, C++ and Fortran, and with programming frameworks such as OpenACC or OpenCL, though with some limitations. CUDA organises applications as a sequential host program that may execute parallel programs, referred to as kernels, on a CUDA-capable device. The compute capability specifies characteristics such as the maximum number of resident threads or the amount of shared memory per multi-processor, which can significantly vary from one version to another (and, consequently, from one graphics card to another) [16].

In order to work with CUDA applications, the programmer needs to copy data from host memory to device memory, invoke kernels and then copy data back from device memory to host memory.

3.2 C++ and OpenMP

C++ is a programming language designed by Bjarne Stroustrup in 1983, and that is standardised since 1998 by the International Organization for Standardization (ISO). The latest version is known as C++17 [17].

OpenMP (Open Multi-Processing) is an Application Programming Interface (API) that supports shared-memory parallel programming in C, C++, and Fortran on several platforms, including GNU/Linux, OS X, and Windows. The latest stable version is 4.5, released on November 2015 [18].

When using OpenMP, the section of code that is intended to run in parallel is marked with a preprocessor directive that will cause the threads to form before the section is executed. By default, each thread executes the parallelised section of code independently. The runtime environment allocates threads to processors depending on usage, machine load, and other factors.

3.3 Java

The Java programming language was originated in 1990 when a team at Sun Microsystems was working first in the design and development of software for small electronic devices, and later in the emerging market of Internet browsing. Once the first official version of Java was launched in 1996, its popularity started to increase exponentially.

Currently there are more than 10 million Java developers and, according to [19], more than 15 billion devices (mainly personal computers, mobile phones, and smart cards) run Java. On January 2010, Oracle Corporation completed the acquisition of Sun Microsystems [20], so at this moment the Java technology is managed by Oracle. The latest version, known as Java 9, was launched in September 2017.

Between November 2006 and May 2007, Sun Microsystems released most of the Java components under the GNU (GNU's Not Unix!) GPL (General Public License) model through the OpenJDK project [21], so virtually all the pieces of the Java language are currently free open source software.

4 Implementations

4.1 First implementation

The first implementation of the brute-force attack is a direct implementation, in the sense that it completes all the steps needed to be performed by the attacker. This means that, taking as input the encrypted data and the IV, the implementation performs the 32 steps of the initialisation phase and the 32 steps of the operation phase for all the keys that are tests. It is important to point out that this implementation has been used primarily as a comparison element with regards to the second implementation.

4.2 Second implementation

The second implementation takes advantage of some peculiarities of the encryption algorithm and the protocol that allow to increase the performance of the brute-force attack.

The first improvement consists in directly using the $\overline{\text{IV}}$ element (instead of using the original IV), as that data can be easily obtained by the attacker during one of the steps of the protocol. By using the $\overline{\text{IV}}$, the attacker does not need to complete the initialisation phase for each key, replacing that operation by the XOR of the $\overline{\text{IV}}$ and a 32-bit portion of the key (bits 16 to 47) in order to produce the 32 bits that are located in the right-hand side of the register at the start of the operation phase. Using this improvement, the running time is practically halved.

The second improvement derives from the fact that the plaintext is known to the attacker and its value is the aforemention hexadecimal value FFFFFFFF. As in the operation phase each round of the algorithm generates a bit of the keystream, it is possible to discard a candidate key as soon as the keystream bit generated by one round does not generate a bit 1 when XORed with the corresponding bit of the encrypted value. This means that, in most cases, only a few rounds of the operation phase are completed in comparison with the full 32 rounds that were completed for each key in the first implementation. By introducing this feature, we have been able to roughly divide the running time by 16.

The code displayed in Listing 1.1 contains the details of second version of the CUDA kernel, where one key is tested by each thread.

5 Results

The tests whose results are presented in this section were completed using the following equipment [22]:

- A PC with an Intel Core i7 processor model 3370 at 3.40 GHz.
- A GeForce GTX 950 card, which is a low tier GPU with 768 processor cores, a base clock of 1024 MHz, a memory bandwith of 105.6 GB/s, and a floating point performance of 1.85 TeraFLOPS [23].

```
#define bit(x,n) (((x)>>(n))&1)
#define g4(x,a,b,c,d) (bit(x,a) + bit(x,b)*2 + bit(x,c)*4 + bit(x,d)*8)
#define f5(a,b,c,d,e) (a + b*2 + c*4 + d*8 + e*16)
#define fa 0x2C79
#define fb 0x6671
#define fc 0x7907287B

__global__ void hitag2_en(uint32_t *ciphertext, uint64_t *key,
uint32_t *plaintext, uint64_t *numtot, uint64_t *serial, uint32_t *enciv)
{
  uint64_t index = blockIdx.x*blockDim.x + threadIdx.x;
  uint64_t z = *key + index;
  uint32_t b1, b2, b3;

  bool fail = false;

  uint64_t LFSR = 0;
  uint32_t bstream = 0;
  uint32_t result = 0;

  // Phase 1: Initilisation

  LFSR = (((z & 0xFFFF00000000)>>32) + ((*serial)<<16)) & 0xFFFFFFFFFFFF;
  LFSR = ((LFSR<<32)^(((uint64_t)*enciv)&0xFFFFFFFF)^(z&0xFFFFFFFF))
        & 0xFFFFFFFFFFFF;

  // Phase 2: Decryption

  for (uint32_t i = 0; i < 32; i++)
  {

    b1 = bit(fc, f5(bit(fa, g4(LFSR, 45, 44, 42, 41)),
             bit(fb, g4(LFSR, 39, 35, 33, 32)),
             bit(fb, g4(LFSR, 30, 26, 24, 21)),
             bit(fb, g4(LFSR, 19, 18, 16, 14)),
             bit(fa, g4(LFSR, 13, 4, 3, 1))));
    b2 = (*ciphertext>>(31-i))&0x01;
    b3 = (*plaintext>>(31-i))&0x01;

    if((b1^b2) != b3)
    {
      fail = true;
      break;
    }

    bstream <<= 1;

    bstream += b1;

    LFSR = (LFSR << 1) +
           ((bit(LFSR, 47)) ^ (bit(LFSR, 45)) ^ (bit(LFSR, 44)) ^
            (bit(LFSR, 41)) ^ (bit(LFSR, 40)) ^ (bit(LFSR, 39)) ^
            (bit(LFSR, 31)) ^ (bit(LFSR, 25)) ^ (bit(LFSR, 24)) ^
            (bit(LFSR, 21)) ^ (bit(LFSR, 17)) ^ (bit(LFSR, 6)) ^
            (bit(LFSR, 5)) ^ (bit(LFSR, 4)) ^ (bit(LFSR, 1)) ^
            (bit(LFSR, 0)));
  }

  result = bstream ^ (*plaintext);

  __syncthreads();

  if(!fail)
  {
    *key = z;
  }

}
```

Listing 1.1: Portion of code belonging to the CUDA implementation.

– A GeForce GTX 1070 card, which is a medium tier GPU with 1920 processor cores, a base clock of 1506 MHz, a memory bandwith of 256.3 GB/s, and a floating point performance of 6.46 TeraFLOPS [24].

While the CUDA and C++/OpenMP applications have been compiled with Visual Studio 2010 and 2017 (the application for the older GeForce GTX 950 was compiled with Visual Studio 2010), the Java application has been compiled with NetBeans 8.0 using the JDK (Java Development Kit) version 1.8.0-141.

In all the tests that have been performed, each application had to check the first 2^{34} possible keys (an arbitrary value large enough in order to obtain valid conclusions) using an encryption/decryption pair generated with the following values:

– Serial number: 0x87654321.
– IV: 0x75B5DE65.
– Plaintext: 0xFFFFFFFF.
– Ciphertext: 0x1CE18551.

5.1 Results of the first implementation

Table 1 shows the running time in seconds of the C++/OpenMP and Java implementations when using a different number of concurrent threads. Tables 2 and 3 show the running time of the fist CUDA application when executed on the GeForce GTX 950 with different grid sizes and a constant block size of 512 and 1024, respectively. Finally, Tables 4 and 5 include the running time of the fist CUDA application when executed on the GeForce GTX 1070 with different grid sizes and a constant block size of 512 and 1024, respectively.

Table 1: Running time in seconds using the C++ and Java multi-threaded first implementation.

Number of threads	1	2	4	8	16	32
C++	18126.60	9084.68	4625.80	3749.45	3748.61	3747.32
Java	17548.88	8461.70	4496.55	3744.72	3694.46	3817.03

Table 2: Running time in seconds using the first CUDA implementation with a block size of 512 on the GeForce GTX 950 card.

Grid size	512	1024	2048	4096	8192	16384
Running time	197.62	196.72	194.92	193.97	193.65	193.43

Table 3: Running time in seconds using the first CUDA implementation with a block size of 1024 on the GeForce GTX 950 card.

Grid size	512	1024	2048	4096	8192	16384
Running time	194.93	193.68	192.72	192.22	191.96	191.87

Table 4: Running time in seconds using the first CUDA implementation with a block size of 512 on the GeForce GTX 1070 card.

Grid size	512	1024	2048	4096	8192	16384
Running time	53.33	50.68	49.39	48.63	48.30	48.12

Table 5: Running time in seconds using the first CUDA implementation with a block size of 1024 on the GeForce GTX 1070 card.

Grid size	512	1024	2048	4096	8192	16384
Running time	51.38	49.60	48.90	48.51	48.32	48.21

5.2 Results of the second implementation

Table 6 shows the running time in seconds of the C++/OpenMP and Java implementations when using a different number of concurrent threads in the second implemenation. Tables 7 and 8 show the running time of the second CUDA application when executed on the GeForce GTX 950 with different grid sizes and a constant block size of 512 and 1024, respectively. Finally, Tables 9 and 10 include the running time of the second CUDA application when executed on the GeForce GTX 1070 with different grid sizes and a constant block size of 512 and 1024, respectively.

Figure 4 shows a graphic representation of the best running time obtained with the second implementation by each platform.

6 Conclusions

In this contribution we have compared the computer capability of several hardware and software technologies using as an example a cryptographic brute-force attack on the legacy algorithm Hitag2. More specifically, we have compared two versions of a CUDA application, a C++ implementation using the OpenMP library, and a Java application that uses the multi-threading capabilities provided by the language.

These tests have shown that, with the best configuration in each case, the native C++/OpenMP application provides a performance only slightly better than the performance of the interpreted Java code. Given that the code of each language was very similar, the most probable explanation is the use of basic data types in both cases, which allowed us to avoid slow-performance Java classes

Table 6: Running time in seconds using the C++ and Java multi-threaded second implementation.

Number of threads	1	2	4	8	16	32
C++	543.08	277.27	167.64	127.85	123.57	119.97
Java	525.89	268.76	161.10	132.67	123.96	121.29

Table 7: Running time in seconds using the second CUDA implementation with a block size of 512 on the GeForce GTX 950 card.

Grid size	512	1024	2048	4096	8192	16384
Running time	11.20	8.96	8.31	7.68	7.40	7.25

Table 8: Running time in seconds using the second CUDA implementation with a block size of 1024 on the GeForce GTX 950 card.

Grid size	512	1024	2048	4096	8192	16384
Running time	9.29	8.37	7.75	7.45	7.34	7.25

Table 9: Running time in seconds using the second CUDA implementation with a block size of 512 on the GeForce GTX 1070 card.

Grid size	512	1024	2048	4096	8192	16384
Running time	5.16	3.81	3.03	2.73	2.54	2.43

Table 10: Running time in seconds using the second CUDA implementation with a block size of 1024 on the GeForce GTX 1070 card.

Grid size	512	1024	2048	4096	8192	16384
Running time	3.90	3.32	2.77	2.58	2.45	2.38

such as `BigInteger`. In addition to that, it is important to take into account that Java's JIT (Just-In-Time) compiler improves the performance by compiling Java bytecodes into native machine code at run time [25], which would also help to explain the similarity of its performance to the one obtained with the C++/OpenMP code. As the multi-threading capabilities are available in Java by default, without having to add any third-party library, it can be stated that Java is a viable alternative to C++ for this kind of developments.

Regarding the increase in the performance when commanding more threads, the tests show that the improvement is tightly related to the number of physical cores, not to the number of logical cores (the processor used in the tests has 4 physical cores and 8 logical cores).

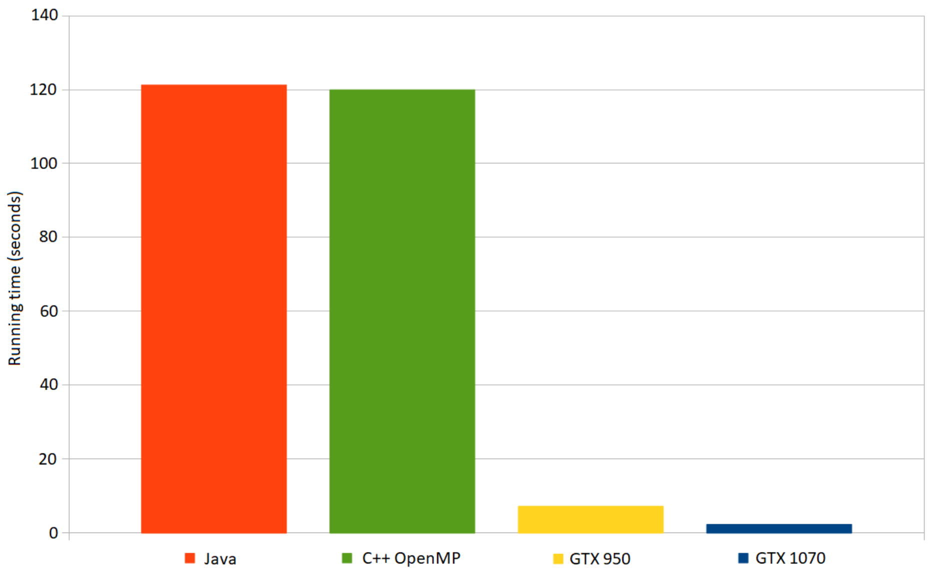

Fig. 4: Running time comparison.

Considering all the results, it is clear the superiority of CUDA cards with respect to advanced CPUs for certain intensive computing tasks. The best result obtained with the GeForce GTX 1070 provides a performance almost 50 times better than that of the C++/OpenMP implementation when using the full capacity of the i7 3370 processor.

Even in the case of using some of the most powerful CPUs avaliable today, such as Intel's i9-7980XE (18 physical cores with a price around 2,000 dollars) or AMD's Ryzen Threadripper 1950X (16 physical cores with a price around 1,000 dollars), the performance would not achieve by far the levels obtained with a CUDA card comparatively cheaper.

Regarding the comparison between the two CUDA devices, the results are aligned with the technical capabilities of the cards such as the number of cores, the memory bandwidth and the floating-point performance rate. In both cases, the best results are obtained when using a bigger grid size, which allows to perform less calls to the CUDA code and avoid some latency issues.

Using the best result obtained with the CUDA versions, it can be extrapolated that the whole set of 2^{48} keys could be tested in less than half a day. This result could be vastly improved when using other CUDA cards such as the Tesla P100 (which has 3584 processor cores and a floating point performance of 10.6 TeraFLOPS), GeForce GTX 1080 Ti (3584 processor cores and 11.3 TeraFLOPS) or TITAN V (5120 processor cores and 14.9 TeraFLOPS) [26, 27].

Acknowledgements

This work has been partially supported by Ministerio de Economía, Industria y Competitividad (MINECO), Agencia Estatal de Investigación (AEI), and Fondo Europeo de Desarrollo Regional (FEDER, UE) under project COPCIS, reference TIN2017-84844-C2-1-R, and by Comunidad de Madrid (Spain) under project CIBERDINE, reference S2013/ICE-3095-CM, also co-financed by European Union FEDER funds.

References

1. Verdult, R., Garcia, F.D., , Balasch, J.: Gone in 360 seconds: Hijacking with Hitag2. In: 21st USENIX Security Symposium (USENIX Security 2012). (2012) 237–252
2. Courtois, N.T., O'Neil, S., Quisquater, J.J.: Practical algebraic attacks on the Hitag2 stream cipher. In: Information Security: 12th International Conference (ISC 2009). (2009) 167–176
3. Courtois, N.T., O'Neil, S., Quisquater, J.J.: Cube cryptanalysis of Hitag2 stream cipher. In: International Conference on Cryptology and Network Security (CANS 2011). (2011) 15–25
4. Stembera, P., Novotny, M.: Breaking Hitag2 with reconfigurable hardware. In: 14th Euromicro Conference on Digital System Design (DSD 2011). (2011) 558–563
5. Garcia, F.D., Oswald, D., Kasper, T., Pavlidès, P.: Lock it and still lose it–On the (in)security of automotive remote keyless entry systems. In: 25th USENIX Security Symposium (USENIX Security 2016). (2016) 929–944
6. Guneysu, T., Kasper, T., Novotny, M., Paar, C., Rup, A.: Cryptanalysis with COPACOBANA. IEEE Transactions on Computers **57** (2008) 1498–1513
7. Li, Q., Zhong, C., Zhao, K., Mei, X., Chu, X.: Implementation and analysis of AES encryption on GPU. In: IEEE 14th International Conference on High Performance Computing and Communication and IEEE 9th International Conference on Embedded Software and Systems. (2012) 843–848
8. Li, C., Wu, H., Chen, S., Li, X., Guo, D.: Efficient implementation for MD5-RC4 encryption using GPU with CUDA. In: 2009 3rd International Conference on Anti-counterfeiting, Security, and Identification in Communication. (2009) 167–170
9. Agosta, G., Barenghi, A., Pelosi, G.: High speed cipher cracking: the case of Keeloq on CUDA. In: 3rd Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures (PARMA 2012), 2012. (2012) 1–7
10. Scharfglass, K., Weng, D., White, J., Lupo, C.: Breaking weak 1024-bit RSA keys with CUDA. In: 2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies. (2012) 207–212
11. Gayoso Martínez, V., Hernández Encinas, L., Martín Muñoz, A., Martínez-Graullera, Ó., Villazón-Terrazas, J.: A comparison of computer-based technologies suitable for cryptographic attacks. In: International Joint Conference SOCO'16-CISIS'16-ICEUTE'16. (2016) 621–630
12. Wiener, I.: Philips/NXP Hitag2 PCF7936/46/47/52 stream cipher reference implementation (2008) `https://web.archive.org/web/20080105114835/http://cryptolib.com/ciphers/hitag2/hitag2.c`.
13. Verdult, R.: The (in)security of proprietary cryptography. Radboud University Nijmegen, Nijmegen (Nederlands) (2015)

14. NVIDIA Corp.: What is GPU computing? (2016) https://www.nvidia.com/object/what-is-gpu-computing.html.
15. Mittal, S., Vetter, J.S.: A survey of methods for analyzing and improving GPU energy efficiency. ACM Computing Surveys **47**(2) (2014) 1–23
16. NVIDIA Corp.: Programming Guide (2016) http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities.
17. ISO/IEC: ISO/IEC 14882:2017. (2017) https://www.iso.org/standard/68564.html.
18. OpenMP: The OpenMP API specification for parallel programming (2016)
19. Oracle Corp.: Go Java. (2018) https://go.java/index.html.
20. Oracle Corp.: Oracle Completes Acquisition of Sun. (2010) http://www.oracle.com/us/corporate/press/044428.
21. Oracle Corp.: OpenJDK. (2018) http://openjdk.java.net.
22. NVIDIA Corp.: CUDA Legacy GPUs (2018) https://developer.nvidia.com/cuda-legacy-gpus.
23. NVIDIA Corp.: GeForce GTX 950 - Specifications (2017) https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-950/specifications.
24. NVIDIA Corp.: GeForce GTX 1070 - Specifications (2017) https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1070/.
25. Oracle Corp.: Understanding Just-In-Time Compilation and Optimization (2011) https://docs.oracle.com/cd/E15289_01/doc.40/e15058/underst_jit.htm.
26. NVIDIA Corp.: The World's Most Powerful PC GPU (2018) https://www.nvidia.com/en-us/titan/titan-v/.
27. TechPowerUp: NVIDIA TITAN V (2018) https://www.techpowerup.com/gpudb/3051/titan-v.