Breaking a Hitag2 Protocol with Low Cost Technology

V. Gayoso Martínez[1], L. Hernández Encinas[1], A. Martín Muñoz[1] and J. Zhang[1,2]
[1]*Institute of Physical and Information Technologies (ITEFI), Spanish National Research Council (CSIC), Madrid, Spain*
[2]*The Honk Kong University of Science and Technology (HKUST), Kowloon, Hong Kong*
{*victor.gayoso, luis, agustin*}*@iec.csic.es, jzhangcf@connect.ust.hk*

Keywords: Cryptography, CUDA, C++, Hitag2, Java, OpenMP, Stream Cipher.

Abstract: Hitag2 is an encryption algorithm designed by NXP Semiconductors that is used in electronic vehicle immobilizers and anti-theft devices. Hitag2 uses 48-bit keys for authentication and confidentiality, and due to that feature it is considered an insecure cipher. In this contribution we present a comparison of low cost technologies able to break a known protocol based on this cipher in a reasonable amount of time. Building on top of these solutions, it is possible to create an environment able to obtain Hitag2 keys in almost negligible time. The procedure can be easily expanded in order to consider other protocols based on the same cipher.

1 INTRODUCTION

Hitag2 is a 48-bit stream cipher used widely in both 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 this contribution, we have focused on the usage of Hitag2 as a PKE system in a publicly known protocol (Verdult et al., 2012). Given the short length of Hitag2's keys, this stream cipher has been considered insecure for some years, and as such it can be attacked by using expensive devices such as COPACOBANA (Guneysu et al., 2008). In addition to that, Hitag2 suffers from more elaborated cryptographic attacks (Courtois et al., 2009; Courtois et al., 2011; Stembera and Novotny, 2011; Verdult et al., 2012; Garcia et al., 2016).

Thus, our goal is not to show that Hitag2 is insecure, but to compare low cost technologies that can be used to obtain the transmitter's key with a sole computer in the scope of the aforementioned protocol. In this sense, we have developed three implementations, two of them using an only-software approach (Java and C++/OpenMP), and the other one based on a CUDA-capable graphics card.

The rest of this paper is organized as follows: In Section 2, we present a brief overview of the Hitag2 algorithm. Section 3 describes the Java, C++/OpenMP, and CUDA platforms, including part of the code used in the CUDA implementation. In Section 4, we offer to the readers the experimental results obtained with our implementations. Finally, our conclusions are presented in Section 5.

2 HITAG2

2.1 Algorithm

Hitag2 is a stream cipher which 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

Martínez, V., Encinas, L., Muñoz, A. and Zhang, J.
Breaking a Hitag2 Protocol with Low Cost Technology.
DOI: 10.5220/0006271905790584
In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy (ICISSP 2017)*, pages 579-584
ISBN: 978-989-758-209-7

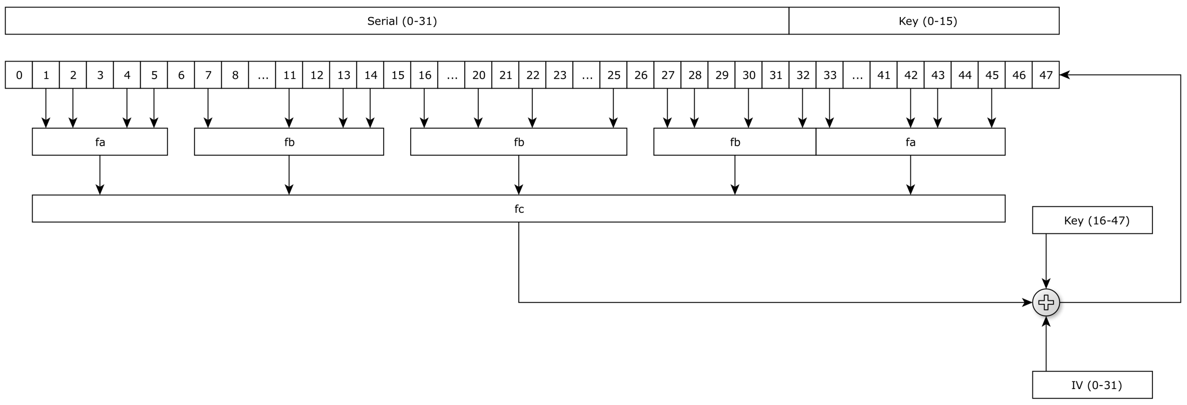

Figure 1: Hitag2 initialization phase.

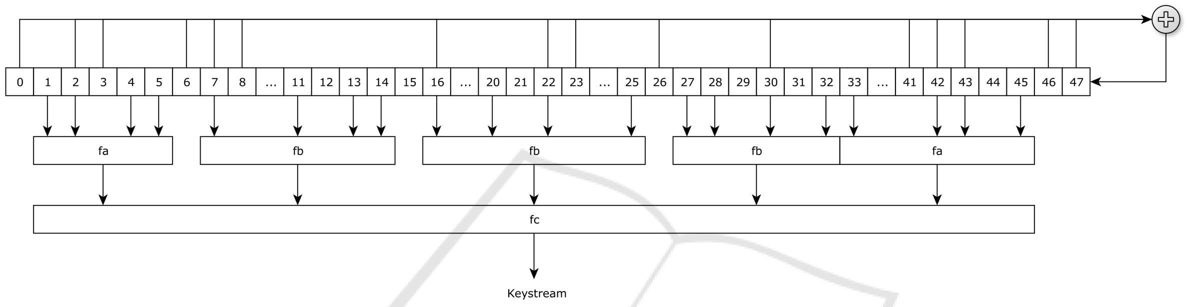

Figure 2: Hitag2 encryption phase.

the registry defines both the next encryption bit and how the registry is updated. Thus, the total number of cycles is defined by the length of the bitstream that needs to be encrypted.

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 encryption 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:

$$f_a(i) = (0\text{x}2\text{C}79)_i$$
$$f_b(i) = (0\text{x}6671)_i$$
$$f_c(i) = (0\text{x}7907287\text{B})_i$$

In the initialization 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:

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

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 the position 47, shifting the register one bit to the left. The new bit is computed according to the following expression:

$$f_c \oplus id_i \oplus k_{i+16} \quad (0 \leq i \leq 31)$$

In the encryption 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 $r_0 \oplus r_2 \oplus r_3 \oplus r_6 \oplus r_7 \oplus r_8 \oplus r_{16} \oplus r_{22} \oplus r_{23} \oplus r_{26} \oplus r_{30} \oplus r_{41} \oplus r_{42} \oplus r_{43} \oplus r_{46} \oplus r_{47} \oplus$.

2.2 Protocol

In the PKE protocol analysed in this contribution, which was reversed engineered and published online in 2008 (Wiener, 2008), the communication between a reader (vehicle) and a transponder embedded in the car key starts with the reader, which sends an authenticate command to the transponder. 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 value 0xFFFFFFFF. If the transponder validates the reader by recovering the 0xFFFFFFFF value, it will send to the reader in encrypted form some configuration bytes only known to both of them (Verdult et al., 2012; Verdult, 2015).

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. As the number of keys is larger than the number of possible ciphertexts (48 bits vs 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.

In this phase of our study, we have focused on the implementations that are able to compute those potential keys. In the next phase, we will focus on improving the retrieval step by including Field Programmable Gate Array (FPGA) devices in the comparison of technologies, and on determining the average number of pairs needed to isolate the correct key.

3 IMPLEMENTATION PLATFORMS

3.1 C++ and OpenMP

C++ is a programming language designed by Bjarne Stroustrup in 1983, and that is standardized since 1998 by the International Organization for Standardization (ISO). The latest version is known as C++14 (ISO/IEC, 2014).

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 (OpenMP, 2016). 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 parallelized section of code independently. The runtime environment allocates threads to processors depending on usage, machine load, and other factors.

3.2 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 (Oracle Corp., 2016), the figure of Java enabled devices (mainly personal computers, mobile phones, and smart cards) is numbered in the thousands of millions. On January 2010, Oracle Corporation completed the acquisition of Sun Microsystems (Oracle Corp., 2010), so at this moment the Java technology is managed by Oracle. The latest version, known as Java 8, was launched in 2014.

3.3 CUDA

GPGPU is the term that refers to the use of a Graphics Processor Unit (GPU) card to perform computations in applications traditionally managed by a Central Processing Unit (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 (NVIDIA Corp., 2016). The disadvantage of using GPUs in those scenarios is their higher power consumption compared to that of traditional CPUs (Mittal and Vetter, 2014).

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 organizes applications as a sequential host program that may execute parallel programs, referred to as kernels, on a CUDA-capable device.

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.

The code displayed in Listing 1 contains the details of the CUDA kernel, where only one key is tested by each thread.

As one of the goals of our study was to determine if the amount of time copying elements back and forth between host and device memories was to some extent comparable to the running time of the kernel, we developed a second version of the CUDA application which is able to request each thread to test a specified number of keys before it finishes its execution.

```
#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 *last_key, uint32_t *last_encrypted, uint64_t *numtot, uint64_t *serial, uint32_t *iv)
{
   uint64_t index = blockIdx.x*blockDim.x + threadIdx.x, a = *serial;
   uint32_t x = *plaintext;
   uint32_t y = *ciphertext;
   uint64_t a = *serial;
   uint32_t b = *iv;
   uint64_t z = *key + index;

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

   // Phase 1: Initilization

   LFSR = (((z & 0xFFFF00000000) >> 32) + (a << 16)) & 0xFFFFFFFFFFFF;

   for (int r = 0; r < 32; r++)
   {
      func = 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))));
      LFSR = (LFSR << 1) + ((bit(z, (31 - r)) ^ bit(b, (31 - r)) ^ func));
   }

   // Phase 2: Encryption

   for (uint32_t i = 0; i < 32; i++)
   {
      bstream <<= 1;
      func = 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))));
      bstream += func;

      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 ^ x;

   __syncthreads();

   if (result == y)
   {
      *key = z;
   }

   if (index == *numtot - 1)
   {
      *last_encrypted = result;
      *last_key = z;
   }
}
```

Listing 1: Portion of code belonging to the CUDA application.

4 TESTS

All the tests whose results are presented in this section were completed using a PC with an Intel Core i7 processor model 3370 at 3.40 GHz. The CUDA-capable graphics card used in the tests is a GeForce GTX 950 card with 768 processor cores, a base clock of 1024 MHz, a memory bandwith of 6.6 GB/s, a floating point performance of 1,572.9 GFLOPS, and a texture rate of 49.2 GTexels per second (GT/s). The GTX 950 is a graphics card that can be purchased by approximately 175 euros. In comparison, the most powerful Nvidia card, the GTX 1080 Ti, uses 3,328 processor cores and can be obtained by 900-1,000 euros.

While the CUDA and C++/OpenMP applications have been compiled with Visual Studio 2010, the Java application has been compiled with NetBeans 8.0 using the JDK (Java Development Kit) version 1.8.0-101.

In all the tests that have been performed, each application has 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.

Table 1 shows the running time in seconds of the C++/OpenMP and Java implementations when using a different number of concurrent threads. Table 2 includes the running time of the CUDA application when executed with different grid sizes but a constant block size of 512. Table 3 presents the results when using the second version of the CUDA application when using different grid sizes but the same block size of 512. Table 4 includes the running time of the CUDA application when executed with different grid sizes but a constant block size of 1024. Table 5 presents the results when using the second version of the CUDA application when using different grid sizes but the same block size of 1024.

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

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

Table 2: Running time in seconds using the first CUDA implementation with a block size of 512.

Grid size		
512	1024	2048
180.66	175.90	173.78

Table 3: Running time in seconds using the second CUDA implementation with a block size of 512 and the kernel loop.

Iterations in the kernel loop			
1	2	4	8
174.88	173.11	172.35	171.88

Table 4: Running time in seconds using the first CUDA implementation with a block size of 1024.

Grid size		
512	1024	2048
175.38	172.56	171.80

Table 5: Running time in seconds using the second CUDA implementation with a block size of 1024 and the kernel loop.

Iterations in the kernel loop			
1	2	4	8
175.02	172.58	171.40	170.82

5 CONCLUSIONS

The tests presented in the previous section provide an interesting result, in the sense that the multithread Java application slightly outperforms the C++/OpenMP application in most of the tests. Given that both implementations are almost identical, the most probable explanation is the use of basic data types in both cases, which allowed us to avoid slow-performance Java classes such as `BigInteger`. Besides, as the Java compiler used in the tests was released in 2016 while the C++ compiler belonged to Visual Studio 2010, it is reasonable to expect that the Java compiler contained the latest advances when executing interpreted code.

Even though we decided to use in the Java and C++/OpenMP tests a number of concurrent threads that surpasses the theoretical limit provided by the i7 processor (which has four physical cores and eight logical ones), and as such the C++ implementation does not improve its performance, the Java application provided better results when requesting a higher number of concurrent threads. We assume that this is due to optimizations of the Java virtual machine, which apparently manages more efficiently a higher number of threads when communicating with the operating system.

Regarding the CUDA implementations, when comparing the version which tries one key in each thread with the version that tries several keys, it is possible to detect a slight improvement when using the second version of the CUDA application. However, the difference is not significant, which implies that the delays created by the passing of data elements between the host and device memories are not a bottleneck in this kind of applications.

When comparing the results of the Java and C++/OpenMP vesions and the results of the CUDA versions, it is clear that, even when using the 8 logical cores of the i7 processor, the non-GPU implementations are not a match for the GPU application. 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 approximately one month.

As a work-in-progress study, in the next phase we are planning to include in the comparison an implementation using a low cost FPGA. In addition to that, we will work on the determination of the number of plaintext/ciphertext pairs needed to correctly isolate the correct key in the analysed protocol as well as in other protocols also based on Hitag2.

ACKNOWLEDGEMENTS

This work has been supported by the European Union FEDER funds distributed through Ministerio de Economía y Competitividad (Spain) under the project TIN2014-55325-C2-1-R (ProCriCiS), and through Comunidad de Madrid (Spain) under the project S2013/ICE-3095-CM (CIBERDINE).

The work associated to this contribution has been performed during J. Zhang's stay at CSIC in 2016.

REFERENCES

Courtois, N. T., O'Neil, S., and Quisquater, J.-J. (2009). Practical algebraic attacks on the Hitag2 stream cipher. In *Information Security: 12th International Conference (ISC 2009)*, pages 167–176.

Courtois, N. T., O'Neil, S., and Quisquater, J.-J. (2011). Cube cryptanalysis of Hitag2 stream cipher. In *International Conference on Cryptology and Network Security (CANS 2011)*, pages 15–25.

Garcia, F. D., Oswald, D., Kasper, T., and Pavlidès, P. (2016). Lock it and still lose it–On the (in)security of automotive remote keyless entry systems. In *25th USENIX Security Symposium (USENIX Security 2016)*, pages 929–944.

Guneysu, T., Kasper, T., Novotny, M., Paar, C., and Rup, A. (2008). Cryptanalysis with COPACOBANA. 57:1498.

ISO/IEC (2014). *ISO/IEC 14882:2014*. http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=64029&ICS1=35&ICS2=60.

Mittal, S. and Vetter, J. S. (2014). A survey of methods for analyzing and improving GPU energy efficiency. *ACM Computing Surveys*, 47(2):1–23.

NVIDIA Corp. (2016). What is GPU computing? https://www.nvidia.com/object/what-is-gpu-computing.html.

OpenMP (2016). The OpenMP API specification for parallel programming.

Oracle Corp. (2010). *Oracle Completes Acquisition of Sun*. http://www.oracle.com/us/corporate/press/044428.

Oracle Corp. (2016). *Go Java*. https://go.java/index.html.

Stembera, P. and Novotny, M. (2011). Breaking Hitag2 with reconfigurable hardware. In *14th Euromicro Conference on Digital System Design (DSD 2011)*, pages 558–563.

Verdult, R. (2015). *The (in)security of proprietary cryptography*. Radboud University Nijmegen, Nijmegen (Nederlands).

Verdult, R., Garcia, F. D., , and Balasch, J. (2012). Gone in 360 seconds: Hijacking with Hitag2. In *21st USENIX Security Symposium (USENIX Security 2012)*, pages 237–252.

Wiener, I. (2008). Philips/NXP Hitag2 PCF7936/46/47/52 stream cipher reference implementation. https://web.archive.org/web/20080105114835/http://cryptolib.com/ciphers/hitag2/hitag2.c.