Breaking Hitag 2 Revisited

Vincent Immler

Ruhr University Bochum, Germany
vincent.immler+space2012@rub.de

Abstract. Many Radio Frequency IDentification (RFID) applications such as car immobilizers and access control systems make use of the proprietary stream cipher Hitag 2 from the company NXP. Previous analysis has shown that the cipher is vulnerable to different attacks due to the low complexity of the cipher and its short 48-bit secret key. However, all these attacks either rely on expensive reconfigurable hardware, namely the Field Programmable Gate Array (FPGA) cluster COPACOBANA, or are impractical. In this paper we develop the first bit-sliced OpenCL implementation for the exhaustive key search of Hitag 2 that runs on off-the-shelf hardware. Our implementation is able to reveal the secret key of a Hitag 2 transponder in less than 11 hours on a single Tesla C2050 card in the worst case. The speed of our approach can be further improved due to its scalability, i.e., we estimate a speed of less than one hour on a heterogeneous platform cluster consisting of CPUs and GPUs that can be realized with a budget of less than 5,000 €. This result enables anyone to obtain the secret key with only two sniffed communications in shorter time and with significantly less cost compared to systems such as the COPACOBANA.

Keywords: Brute-force Attacks, HITAG2, GPU, Cryptanalysis, OpenCL.

1 Introduction

Since a strong increase of car theft in the early nineties all major car manufacturers incorporated electronic immobilizers. These systems usually consist of an RFID transponder in the car key and a reader around the ignition lock. Each time the car is started, the transponder is being read by the reader and verified if it is a genuine key. Early systems just used plain messages for authentication, while later systems started to have encryption schemes implemented.

One particular immobilizer system makes use of the proprietary stream cipher Hitag 2 from NXP. According to the NXP website, it is still in production and deployed in many areas which emphasizes the importance of this system. Previous analysis has shown the vulnerability of the cipher, nevertheless it is still being advertised to offer "main stream security" [8].

All existing attacks focus on obtaining recorded communications to determine the key via an analytical attack or exhaustive key search. Obtaining these recorded communications is fairly simple, as anyone being in possession of the car and the key can record the communication. This is even of interest for legitimate car owners in case they want to have a cheap extra key for their car.

Due to that, some companies offer key copy machines and sniffing devices such as SILCA's P-Box and SNOOP[10], which may eventually copy keys using the Hitag 2 cipher, though the inner working mechanism is not known publicly.

1.1 Previous Work

Hitag 2 was kept secret by the manufacturer but it was reverse engineered and presented in 2008 [5]. Together with the description of the cipher, a reference program was made available and allowed further research. On that score, several attacks evolved. A type of successful attack is based on algebraic attacks.

One such attack was carried out by Nicolas Courtois, Sean O'Neil, and Jean-Jacques Quisquater in their paper "Practical Algebraic Attacks on the HITAG2 Stream Cipher" which was presented at the Information Security Conference (ISC) in 2009 [6]. The first part of their work details the weaknesses of Hitag 2 with regard to algebraic attacks and explains the principle on how to transform the problem into a Satisfiability (SAT) problem, which may then be solved using the SAT solver MiniSat 2.0. The second part of their paper presents different attacks of which the following was the most practical: 14 bits of the key are fixed/guessed; for four known IVs the problem is then represented in equations solvable by a SAT solver. According to their paper, the attack takes 2 days which was recently confirmed by Mate Soos, developer of CryptoMiniSat2 [11].

Another algebraic attack by Karsten Nohl and Henryk Plötz presented during the HAR2009 lacks detailed information about the restrictions of their attack, which is successful in 6 hours on a standard PC [7]. Hence, we did not take their result into further consideration.

An attack by using the brute-force technique was implemented by Petr Stembera on the FPGA cluster COPACOBANA [12]. The runtime of the attack is less than 2 hours (103.5 minutes) in the worst case and requires two recorded authenticator values. All successful attacks are summarized in Table 1 and compared to the estimation of a brute-force attack implemented in software, running on a single CPU requiring approximately 800 operations per key [12].

Other attacks by time-memory-trade-off (TMTO) [2] are not feasible in real world attacks (because of the amount of required keystream) and protocol weaknesses as shown in [4] cannot be applied to recover the key (plus, the memory page containing the key can be locked against reading).

Table 1. Summary of known attacks on the Hitag 2 cipher

Publication	Method	Runtime	Prerequisites	Platform
[6]	Algebraic attack	2 days	4 authenticators	PC (2 GHz)
[12]	Brute-force	2 hours	2 authenticators	FPGA cluster
[12]	Brute-force	4 years (est.)	2 authenticators	PC (1.8 GHz)

1.2 Contribution of This Paper

Our paper explains the steps necessary to realize an OpenCL implementation to break Hitag 2 by exhaustive key search. We show how to efficiently compute and verify all possible key candidates by using a bit-sliced approach, hence breaking the stream cipher in less than 11 hours on a single Tesla C2050 graphics card.

We also demonstrate the benefit of using a heterogenous computing environment consisting of CPUs and GPUs by estimating the runtime of our approach on heterogenous cluster systems. Our analysis shows that our implementation outperforms any previous attack. It runs on off-the-shelf hardware and can be realized with a budget of less than 5,000 €, thus setting a new lower bound in cost, speed, and practicability for cloning car keys.

1.3 Outline

At first, we introduce the fundamentals of bit-slicing and the cipher itself in Section 2. Afterwards, in Section 3, we sketch the important properties of OpenCL and the available hardware architectures. We then pinpoint the implementation details in Section 4 and explain how to efficiently map the attack to the hardware.

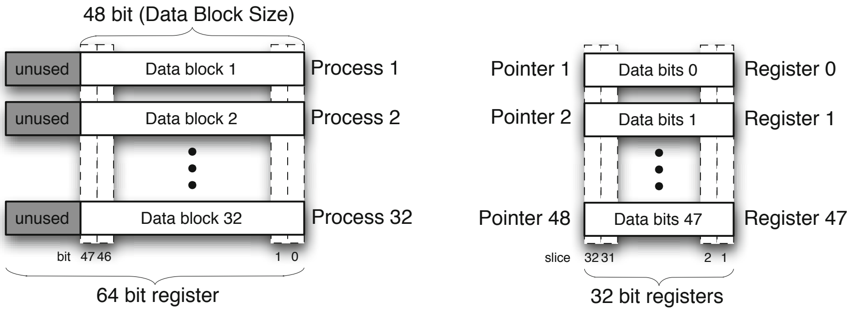
Finally, in Section 5 we conduct practical experiments and measure the execution time of our implementation on different hardware platforms. In addition to that, we evaluate the practical feasibility of realizing a heterogenous cluster in comparison to the costs and speed of the COPACOBANA.

2 Fundamentals

In the following section we introduce the bit-slicing technique that enables us to efficiently implement the cipher, which is explained thereafter. In addition to that, we describe the authentication protocol and the messages that need to be recorded for a successful attack.

2.1 Bit-Slicing

Many algorithms, especially stream ciphers, operate on bits instead of bytes or words, which would be optimal for n bit wide registers of general purpose processors. To allow an efficient implementation of these hardware-oriented algorithms, Eli Biham introduced a concept named bit-slicing [3]. The idea is to perform a chunk-wise serial to parallel bit transposition of the input blocks, thus obtaining a matrix where the n-th column represents the n-th input block and each n-th row containing all bits of the n-th bit position of each input block. By doing this bit-slicing, one achieves a Single Instruction Multiple Data (SIMD) processing of the input data, hence processing more than one data block at a time per computation. The rearrangement can be fitted to the native register width of the underlying processing unit. For many algorithms the bit-slice implementation



(a) Non-sliced input data (b) Bit-sliced input data

Fig. 1. Comparison of the different data representations. To the left is the non-sliced form, to the right the bit-sliced form. Each slice represents one block of data and each bit position is put into a single register.

is the fastest, even with the extra cost of the rearrangement. For illustration purposes, we assume a data block size of 48 bit as can be seen in Figure 1a.

Since the register width of current processors is either 32 or 64 bit[1], it is necessary to partition all input data blocks accordingly to the targeted platform. For n bit registers this means packing all bits of the same input position within the n input blocks together in one register. This process is then repeated for all bit positions and results in a representation as depicted in Figure 1b.

For algorithms using functions other than bitwise operations, it is necessary to modify these functions accordingly to only use bitwise operations (if possible). Moreover, the rearrangement must be done for the plaintext, as well as the key (for cryptographic schemes). Furthermore, it is necessary to transpose the ciphertext to ensure compatibility with non-sliced implementations (if required).

2.2 HITAG 2 Stream Cipher and Protocol

Stream Cipher: The HITAG 2 cipher is a bit-oriented stream cipher and used to encrypt transmissions within the HITAG 2 protocol. The cipher, as shown in Figure 2, consists of a 48 bit secret key, a 48 bit Linear Feedback Shift Register (LFSR), and a non-linear output function with 20 input bits and 1 output bit per clock.

This output function can be considered as two different levels of multiplexors. The four input bits to the functions f_a and f_b serve as address bits and select one of the contained data bits. The function f_c works respectively. To generate key stream, one has to run through an initial processing which is now explained:

1. *LFSR initialization*: At first, the LFSR is being initialized by loading the 32 bit serial number (SN) and bits 0..15 of the key into the LFSR.

[1] In this paper, we always assume a native register width of 32 bit.

2. *State randomization*: Subsequently, the LFSR is clocked 32 times, each time using the output bit of the non-linear output function as a feedback bit Exclusive OR (XOR) the remaining bits of the key XOR the Initialization Vector (IV) (one bit each cycle). This process is illustrated in Figure 2. Please note that the feedback function of the LFSR is not used during this stage.
3. *Keystream generation:* To generate keystream, the feedback function (which is an XOR of all taps) of the LFSR is now in use, in addition to the output function. All other parts of the cipher remain inactive. The first 32 output bits are inverted and represent an authenticator used during a protocol run. The consecutively generated keystream is then used for encryption.

Alternatively, the output function can be described as an S-box, in Algebraic Normal Form (ANF) or in boolean logic. This ultimately leads to the description for the bit-sliced filter functions[2], as given in [13]:

– Filter function f_a: `(~(((a|b)&c)^(a|d)^b))`
– Filter function f_b: `(~(((d|c)&(a^b))^(d|a|b)))`
– Filter function f_c: `(~((((((c^e)|d)&a)^b)&(c^b))^(((d^e)|a)&((d^b)|c))))`

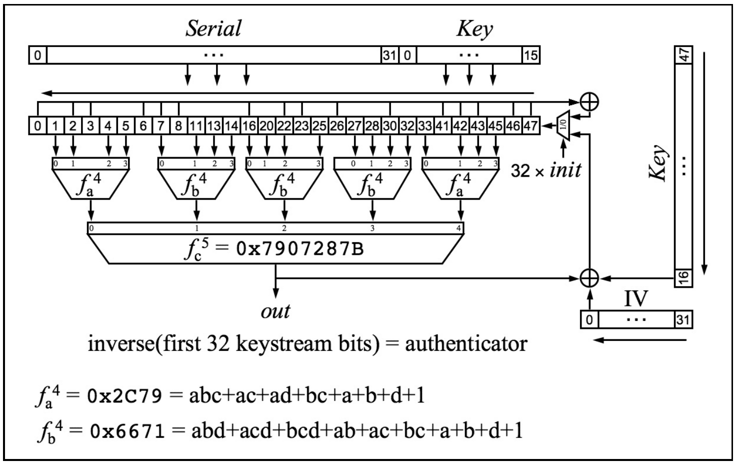


Fig. 2. Graphical description of the HITAG 2 cipher comprising the different components, namely the LFSR and the filter functions f_a, f_b, f_c[13]

Protocol: Within the HITAG 2 protocol, there are two modes for authentication: password and crypto [9]. During the password based authentication, a password in plaintext is being transmitted. Obviously, this mode offers no protection

[2] Here, the operands ~, &, | and ^ represent the equivalent bitwise operands of the C programming language. The variables a, b, c, and d denote the input values that are determined by the tap position of the LFSR.

against eavesdropping. We therefore concentrate our effort on the crypto mode authentication, as it is sketched in Figure 3.

This authentication mode works as follows: If a HITAG 2 transponder is placed within the interrogation zone of a reader, the transponder needs a certain time to power up and is in a wait state. The reader now issues a `Start_Auth` command which consists of the 5 bits: '`11000`'. The transponder responds with its 32 bit serial number (SN) and a preceding 5 bit header, where all bits are set to '1'. The base station then chooses an IV (32 bit) and generates the authenticator which consists of the first 32 bitwise inverted keystream bits.

During the reception of the IV and authenticator Auth_B, the transponder computes the same authenticator (denoted as Auth_T) and checks if Auth_B matches the one that was computed. If they match, the transponder responds with a header and the encryption of Page 3 (a configuration memory page).

A successful authentication implies an entity authentication of the reader towards the transponder and causes the transponder to go in an authorized state. This state permits to read/write the tag using encrypted communication.

To break the authentication protocol, we only need to record two pairs of IV and authenticator and may even use an emulation device to trigger the output of a valid IV and authenticator pair (for a given serial number). If the key is successfully recovered, it is easy to decrypt previously recorded communication. In case no memory read/write locks are set, one may read/write the transponder memory or emulate the complete device using the recovered key.

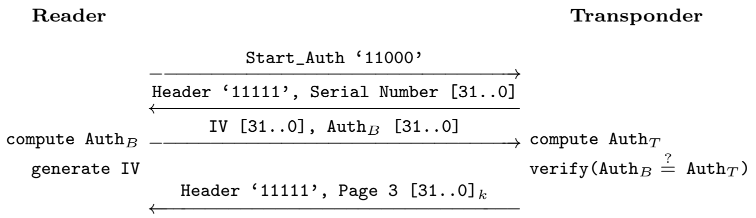

Reader
Transponder
Start_Auth '11000'
Header '11111', Serial Number [31..0]
IV [31..0], Auth$_B$ [31..0]
compute Auth$_B$
generate IV
compute Auth$_T$
verify(Auth$_B \overset{?}{=}$ Auth$_T$)
Header '11111', Page 3 [31..0]$_k$

Fig. 3. Authentication messages of the HITAG 2 protocol in crypto mode [9]

3 Overview of OpenCL

In the past decade, multicore processors emerged and both CPUs as well as GPUs have undertaken a swift development, especially GPUs in adding massively parallel computational power. In contrast to the programming language CUDA which is only geared towards GPU platforms, is OpenCL a framework that executes across heterogeneous platforms consisting of CPUs and GPUs.

Therefore, it is beyond scope to cover all details of all platforms that are able to run OpenCL. However, they share a common abstraction layer as it is depicted in Figure 4. In this architecture, a host device is the control instance for several compute devices, for example, a CPU or GPU.

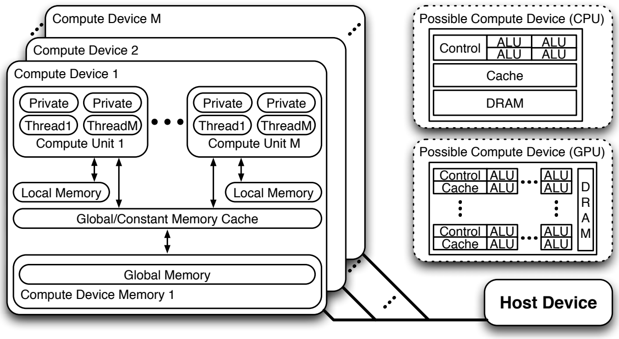


Fig. 4. This figure illustrates the generic OpenCL architecture and the different memory address spaces

These compute devices can be organized in different sets, namely a platform or a context, to ease the management and identify the correct device. The memory of the compute devices is partitioned into four different address spaces: `private`, `local`, `constant`, `global` which is primarily based on recent GPU architectures. To run a program on these devices, a pre-compiled function called kernel is loaded onto the device and executed by a user-specified number (which is the global work size). Each thread is assigned a global identification number (GlobalId). The global work size can be partitioned into blocks by a parameter named local work size that (should) evenly divide the global work size. These blocks, called work-group (GroupId or BlockId), contain threads with a local identification number (LocalId) that are called work-item.

Work-items or sub-sets (so called warps) of work-groups are executed on the compute units provided by the compute device. The actual parallelism is realized by the device and depends on its architecture and kernel code. Within a work-group, inter-thread communication may take place using `local` memory. Each work-item uses its own `private` memory space for computations. Globally available memory is the read-only `constant` memory and the read-write `global` memory.

Limitations of OpenCL may vary due to the different hardware architectures. For GPU devices, memory size and bandwidth, inefficient memory access patterns and cache size are common limitations. Path divergencies of work-items under branching also result in performance penalties. Due to that, it is important to tailor the kernel to a specific device family to address these problems.

4 Implementation

At first, we reason about the techniques used to implement the cipher in Section 4.1. Afterwards, an overview of the system used for our implementation is given in Section 4.2. A difficulty to overcome is the cost of the bit transposition because of the bit-slicing, especially for the key candidates, which is shown in Section 4.3. In a next step, we describe the computation that is carried out by our kernel in Section 4.4.

4.1 Implementation Considerations

To efficiently implement a brute-force attack, we make the following important observation about the cipher: Keys that have a common part (from their 'beginning' on) may share steps of the cipher setup. As an example, keys that have their 16 least significant bit in common may operate on a memory copy of the LFSR that is initialized only once. By taking this idea to the maximum extent, we could possibly use many nested for-loops, each time using a memory copy of the upper loop and using each loop index as next portion of the key.

Alternatively, instead of having (too) many loops, we can also make use of the general idea to initialize the LFSR of the cipher up to a certain point, where we can load this partially initialized LFSR onto a compute device and finish the computation (thus, using parallelization). On our compute device, we can then simply use the identification numbers of each work-item or work-group as potential key guess.

Also relevant for the efficient implementation is to have fast building blocks in software. This can be achieved by using bit-slicing. The LFSR of the cipher is denoted as the array `lfsr[48]`, representing 32 LFSR states of the cipher in parallel. Furthermore, we need to implement the bit-sliced non-linear output function and the feedback function of the LFSR. The latter is simply implemented by 15 XOR operations (cf. function `hitag2bs_round`, Appendix D), whereas the former is realized by the bit-sliced filter functions f_a, f_b, f_c, and results in 46 boolean operations (cf. function `f20bs`, Appendix D).

Also necessary to implement is the 'clocking' of the LFSR. In hardware, this causes a shift of each bit to the next position. In software, an array shift fulfills the same purpose if operating on bit-sliced data. However, by unrolling all subsequent function calls one can omit the array shift by manually adjusting the indices to imitate the shift, thus saving a significant amount of operations[3].

4.2 System Overview

Our system consists of one host device and an arbitrary count of compute devices. On the host device, we perform the necessary steps to access the compute devices and partially initialize/randomize the LFSR to finish the computation on the compute device. This process is described as pseudo-code in Algorithm B.1.

Together with the partially randomized LFSR data, we load the kernel onto the compute device that finishes the cipher computation up to the point, where the output is compared to the sliced authenticator. The result of the computation is then retrieved by the host device and evaluated, i.e., the result contains the

[3] Due to that, by referring to 'clocking' we mean the processing of the LFSR (and inputs) accordingly to the cipher description.

number of possible key candidates found (a counter) and information to recover the key on the host to verify it. The computation carried out by the kernel is described in detail in Section 4.4.

We highlight the fact that with one authenticator and IV pair, one may find up to $2^{48}/2^{32} = 2^{16}$ key candidates. These candidates are therefore verified against a second authenticator and IV pair. This verification takes place in parallel to the continuous execution of the kernel(s) and causes only negligible overhead.

4.3 Key Space Partitioning

To circumvent the transformation overhead of the key challenges, we carefully partitioned the key space of 2^{48} based on the ideas given in [1]. Moreover, we are able to distribute the load between host and compute device to benefit from the shared computational cost during the initialization phase.

The details of this process are given in Appendix A and result in a partitioned key space as it is illustrated in Figure 5. We emphasize that the computational cost on the host is minimal by using memory copies.

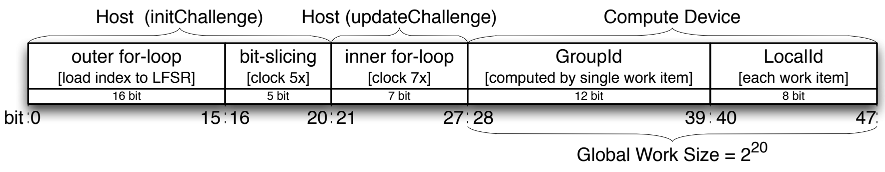


Fig. 5. This figure illustrates the partitioned key space and how the different parts of the systems are used to create the key challenges

4.4 Kernel Implementation

The purpose of the kernel is to finalize the state randomization and then to interleave the authenticator generation with a verification to stop execution if all key candidates have been shown invalid.

The input arguments of the kernel are in sequential order: precomputed LFSR, remaining bits of IV, precomputed key guess XOR IV for each work group, sliced authenticator, result data. For a detailed analysis, the actual (annotated) code is given in Appendix D with only obvious gaps to keep it short.

Finalize State Randomization: As a first step, only the first work item of each work group further randomizes the LFSR by processing the key and IV bits in addition to performing the bit-sliced non-linear output function. The resulting 12 output slices are stored in `local` memory. Consistency is guaranteed by using a synchronization barrier to allow other work items in the same group to access the previously computed output slices.

Afterwards, the last portion of the key is loaded from `constant` memory (precomputed on the host) and processed by each work item of a group. This puts the LFSR in a state ready for keystream generation.

Keystream Generation and Output Verification: The keystream generation consists of the round function and is also fully unrolled. Each time the round function is carried out we get one output slice which is a 32 bit value. This value represents 32 times 1 output bit per key candidate per cipher clock cycle.

To verify this output and track the valid key candidates, we make use of the `Not Exclusive Or (NXOR)` logical function by applying this operation to the output of the round function and one register of the authenticator. Afterwards, we `AND` the result with a bit mask which is initialized with `0xFFFFFFFF`. By doing so, we keep ones in all the bit positions where a valid key candidate remains.

After 8 times 'clocking' the LFSR, we check if any key candidates are left (bit mask must be non-zero). If not, we terminate. If key candidates are left, we repeat the process until we generated 32 slices of output, each time checking after 8 outputs if any key candidates are left by checking the bit mask. By checking only every 8^{th} mask, we reduce path divergencies (by minimizing if-else clauses) that are costly on GPU hardware.

The concept of the bit mask was also introduced to directly verify the sliced output by only causing 3 extra operations per slice. Finally, if the mask is still non-zero after all steps, we increment a counter and return the mask(s) and the globalId(s) of the thread, to be able to recover the possible key candidate.

4.5 Analysis of Our Kernel Implementation

To rate our kernel and verify the efficiency, it is our goal to have very few operations per key. To estimate this, we carry out the following simplified calculation.

The state randomization within the kernel is split in two stages. The first contributes only $12{\cdot}46/256 = R_1 \approx 2$ to the overall operations (first thread of a group only). The second is executed by each thread resulting in $8{\cdot}46 = R_2 = 368$. The subsequent key stream generation results in $32 \cdot (46 + 3 + 15) = C = 2048$, completely disregarding the fact that with each if clause the amount of threads actually running significantly reduces (but possibly inducing serialization). Taking the bit-slicing into consideration, we can divide the sum of $R_1 + R_2 + C = 2418$ by 32 which yields approximately 75.56 operations per key.

While running on SIMT architectures (e.g., Tesla C2050), we could possibly further divide this number by the number of processors (e.g., 448), resulting in hypothetical 0.169 operations per key. Besides the fact that we ignored the operations contributed by the host, we did not consider amongst other factors: architectural properties (possible serialization on GPU, dual dispatch units on Fermi), memory operations, and cache size.

However, this estimation indicates that our implementation is highly efficient in terms of operations per key. All this can be implemented using 63 registers (C2050) with negligible register spilling, resulting in an occupancy of each streaming processor (SM) of 33 % (with 2 active thread blocks per SM).

5 Experimental Results and Costs

In this section we conduct practical experiments on different hardware platforms and present the obtained results. We evaluate the practical feasibility of realizing a heterogeneous cluster running on off-the-shelf hardware to break the cipher.

5.1 Throughput Evaluation on a Single Device

Our evaluation aimed at comparing the speed of our OpenCL implementation on the different architectures offered by Nvidia, AMD and Intel. For each GPU platform, minor modifications to the kernel were applied to optimize the result. For CPUs, more significant changes were necessary, e.g., we changed the data type from 32 bit to 64 bit and as a consequence, the overall distribution between host and compute device. The platform details are given in Appendix C.

It was not our goal to write native assembly code (e.g., PTX, IL, x86 assembly), even though we expect a certain performance benefit from that. Instead, we analyzed the speed by using our generic OpenCL kernel as starting point, to find out how well the attack performs on the different architectures.

Further changes were manual loop unrolling on AMD hardware, as we noticed a significant speedup by doing so. Due to the manually unrolled code, it is neither possible to change the global nor local work size on the fly.

Our obtained results are shown in Table 2. The first three columns specify the brand, device, and name (type) of the architecture followed by the number of compute units and cores provided by each platform. The last column shows the kernel runtime and already includes the overhead to set, write, and read the kernel arguments as well as the time to execute the function `updateChallenge`. Any other computational overhead is negligible.

As one can see, a single Tesla C2050 is the fasted compute device, however the keys per € ratio of the GTX295 is much better. In terms of keys per watt, the mobile graphics chipset 6750M offers a competitive ratio of $2^{25.7}$, even compared to the COPACOBANA ($2^{26.17}$). Of course, the overall runtime is different.

Table 2. Comparison of the evaluated compute devices, their technical specifications and the obtained runtimes per kernel execution (including overhead)

Chipset Brand	Device	Architecture Name (Type)	#CUs[a]	#Cores[b]	Clock [MHz]	TDP [W]	Release [Date]	Kernel [ms]
Nvidia	GTX295	GT200 (SIMT)	30	$30 \cdot 16 = 480$	1242	289	Jan 2009	7.5
Nvidia	C2050	GF100 (SIMT)	14	$14 \cdot 32 = 448$	1150	238	Nov 2009	4.5
Amd	6750M	TeraScale2 (VLIW)	6	$6 \cdot 16 \cdot 5 = 480$	600	25	Jan 2011	24.5
Intel	2820QM	Sandy Bridge (SIMD)	8	4	2300	45	Jan 2011	50[c]
Intel	2xE5540	Nehalem (SIMD)	$2 \cdot 8$	$2 \cdot 4 = 8$	2530	80	Q1' 2009	30[c]

[a] Compute Unit (CU): NVIDIA: Streaming Multiprocessor (SM), AMD: SIMD Engine, Intel: Siblings.
[b] Cores: NVIDIA: Streaming Processor (SP), AMD: Stream Core (SC), Intel: Core.
[c] Runtime of 100 ms resp. 60 ms scaled with 1/2 for comparison reasons (due to 64 bit registers).

5.2 Cost Evaluation of a Heterogeneous Platform Cluster

A great advantage of using OpenCL is the possibility to realize a heterogeneous platform cluster consisting of CPUs and GPUs. Hence, one is able to better utilize the given hardware in comparison to GPU only clusters. We now consider the financial effort to break the cipher by realizing such a cluster with a budget of less than 5,000 €.

By considering the throughput results for a single device, we estimate the runtime of a particular cluster to show that we can easily outperform the COPACOBANA with less budget using off-the-shelf hardware. For our cost estimation, we assume that one workstation contains two video cards and one CPU.

Table 3 shows the results of our cost analysis. The first two columns report the models of both GPU and CPU, followed by two columns reporting the price of each device. The next column is a cost analysis and presents the number of workstations affordable within the given price range.

Table 3. Overview of two different heterogeneous cluster systems with cost and runtime estimation to break HITAG 2

Device		**Cost/Device**		**# of Workstations Affordable**	**Estimated**
GPU	**CPU**	**GPU**	**CPU**	**with budget of 5000 €**	**Runtime [hours]**
GTX295	i7-860	299	229	5	0.5
GTX295	i3-540	299	76	6	0.3

For our analysis, we suppose an overhead of 150 € per workstation, plus the costs for two graphics cards and the CPU. Because we lacked suitable hardware, we assumed a moderate performance gain by 10% when using the i7-860 and a gain by 5% when using the i3-540. We emphasize that no communication between the workstations is required, thus no performance penalty affects the scalability.

6 Conclusion

In this paper, we propose the first bit-sliced OpenCL implementation of HITAG 2 to perform an exhaustive key search on heterogeneous platforms. We present results of our experimental evaluation and show that breaking the cipher is a matter of hours on a single GPU. We emphasize that our hardware setup was rather outdated and more recent hardware would perform even more promising.

Due to the nature of our approach, we can easily make use of more hardware resources and provide data on building a heterogeneous cluster to break the cipher within an hour using off-the-shelf hardware. Given the fact that we did not use platform dependable code, we can assume that future versions may run even faster by using hand-written IL or PTX assembly code. In order to improve the results on CPUs, vector data types could be used to exploit the width of SSE or AVX registers.

References

1. Agosta, G., Barenghi, A., De Santis, F., Pelosi, G.: Record Setting Software Implementation of DES using CUDA. In: Proceedings of the 2010 Seventh International Conference on Information Technology: New Generations, ITNG 2010, pp. 748–755. IEEE Computer Society, Washington, DC (2010)
2. Bachani, R.: Time-memory-data cryptanalysis of the HiTag2 stream cipher. Master's thesis, Technical University of Denmark (2008), `http://code.google.com/p/cryptanalysis-of-hitag2/`
3. Biham, E.: A Fast New DES Implementation in Software, pp. 260–272. Springer (1997)
4. Bogdanov, A., Paar, C.: On the Security and Efficiency of Real-World Lightweight Authentication Protocols (2008)
5. Courtois, N.T., O'Neil, S.: FSE Rump Session – Hitag 2 Cipher (2008), `http://fse2008rump.cr.yp.to/00564f75b2f39604dc204d838da01e7a.pdf` (August 30, 2012)
6. Courtois, N.T., O'Neil, S., Quisquater, J.-J.: Practical Algebraic Attacks on the Hitag2 Stream Cipher. In: Samarati, P., Yung, M., Martinelli, F., Ardagna, C.A. (eds.) ISC 2009. LNCS, vol. 5735, pp. 167–176. Springer, Heidelberg (2009)
7. Henryk Plötz, K.N.: Breaking Hitag 2 (August 2009), `https://har2009.org/program/attachments/113_breaking_hitag2_part1_hardware.pdf`
8. NXP. HITAG 2 transponder IC, `http://www.nxp.com/products/identification_and_security/smart_label_and_tag_ics/hitag/series/HITAG_2_TRANSPONDER_IC.html#overview` (August 30, 2012)
9. NXP. HT2 Transponder Family Communication Protocol – Revision 2.1, `http://www.proxmark.org/files/index.php?dir=Documents%2F125+kHz+-+Hitag%2F&download=HT2protocol.pdf` (August 30, 2012)
10. SILCA. Die neueste Silca Transponder Technologie, `http://www.silca.de/media/112090/v4/File/silca-id46-solution-rw4.pdf` (August 30, 2012)
11. Soos, M.: Cracking Industrial Ciphers at a Whim (April 2011), `http://50-56-189-184.static.cloud-ips.com/wordpress/wp-content/uploads/2011/04/hes2011.pdf` (August 30, 2012)
12. Stembera, P.: Cryptanalysis of Hitag-2 Cipher. Master's thesis, Czech Technical University in Prague (2011), `https://dip.felk.cvut.cz/browse/pdfcache/stembpe1_2011dipl.pdf` (August 30, 2012)
13. Unknown. Hitag2 Stream Cipher C Implementation and Graphical Description (2006-2007), `http://cryptolib.com/ciphers/hitag2/` (June 14, 2012)

A Details of Key Space Partitioning

We continue with the description of the key space partitioning and make use of the C syntax. The array `keyChallenge[48]` shall (fictively) contain the key candidates in bit-sliced representation. In detail, we start with:

– The *outer-for-loop* (from 0 to $2^{16} - 1$): This loop contains the function `initChallenge` that takes the index of the loop as key candidate. Each candidate is sliced and used as the quantity of the key that is directly loaded into the LFSR together with the serial number. Of course, loading the serial number can be done once and a memory copy used for all loop iterations.

```
for(i=0; i<32; i++) { // slice serial number and load into LFSR
   if(serial&1) { lfsr[i] = ~lfsr[i]; }
   serial >>= 1;
}
// ... keyGuess is the index of the outer-for-loop ...
for(i=0; i<16; i++) { // slice index and load into LFSR
   if(keyGuess&1) { lfsr[32+i] = ~lfsr[32+i]; }
   keyGuess >>= 1;
}
// ... keyGuess represents keyChallenge[0..15]
```

– The next 5 bits are all possible values in this range (read column-wise), due to bit-slicing and the native register width of 32 bit ($2^5 = 32$, for GPUs):

```
fixedBits[0]=0x55555555; // 01010101010101010101010101010101 //=keyChallenge[16]
fixedBits[1]=0x33333333; // 00110011001100110011001100110011 //=keyChallenge[17]
fixedBits[2]=0x0F0F0F0F; // 00001111000011110000111100001111 //=keyChallenge[18]
fixedBits[3]=0x00FF00FF; // 00000000111111110000000011111111 //=keyChallenge[19]
fixedBits[4]=0x0000FFFF; // 00000000000000001111111111111111 //=keyChallenge[20]
```

These bits are fixed and can be directly clocked into the LFSR on the host as part of the function `initChallenge`. This step represents the first 5 out of 32 clock cycles to randomize the state of the LFSR. We later take advantage of the fact that we can check for the valid keys by verifying a bit mask.

– The *inner-for-loop* (from 0 to $2^7 - 1$): These 7 bit represent the number of kernel calls within each outer loop and are clocked into the LFSR by applying the function `updateChallenge`.

– For the remaining 20 bit, the kernel is sent to the compute device. Hence, our global work size is 2^{20}, whereas:
  - 12 bit are represented by the blockId and
  - 8 bit by the localId (the maximum size of a work group on some devices).

  Together, they specify the unique key challenges of each work item. We stress that the global work size was limited to 2^{20} due to the fact that some compute devices refused to work with larger values (e.g., AMD 6750M).
  We notice that the 8 bit determining the localId are a fixed index for each thread and thus can be precomputed by XORing all (sliced) indices with the corresponding (sliced) IV bits. This precomputed data is then loaded onto the device as array `workGroupSlices` (size of $256 \cdot 8 \cdot 32$ bit).

B Pseudocode of the Host Attack Algorithm

To implement this algorithm, we wrote two functions, namely `initChallenge` and `updateChallenge`. These functions are incorporated in the basic structure of our program which consists of the set-up stage and two for-loops.

The first for-loop named *outer-for-loop* represents the LFSR initialization and calls the function `initChallenge`. The *inner-for-loop* creates a memory copy of the data provided by the outer-for-loop and continues the state randomization (clocking the LFSR) by calling the function `updateChallenge`.

The pseudo-code implementation can be further improved by using the idea of nested for-loops, each time operating on a memory copy of the upper loop.

Algorithm B.1: Host Attack Algorithm

```
   Input  : $SN$, $(IV_1, Auth_1)$, $(IV_2, Auth_2)$, fixedBits[5]
   Output: Key
 1 Discover and set up compute device;
 2 Create sliced representation of $IV_1$ and $Auth_1$;
 3 Precomputation of workGroupSlices;
 4 for $i \leftarrow 0$ to $2^{16} - 1$ do
 5     lfsr$_i$ ← initChallenge($i$, fixedBits, $SN$);
 6     for $j \leftarrow 0$ to $2^7 - 1$ do
 7         lfsr$_j$ ← memcpy(lfsr$_i$);
 8         lfsr$_j$ ← updateChallenge(j, lfsr$_j$);
 9         Send data to compute device;
10         $counter$ ← Retrieve result data if kernel finishes;
11         if $counter > 0$ then
12             verify data with $(IV_2, Auth_2)$ in parallel and continue execution;
13         end
14     end
15 end
16 return key;
```

C Platform Specifications

The specifications of the platforms used for testing are as follows:

- MacBook Pro: Intel i7 2820QM, AMD 6750M, OS X 10.7.4, Apple LLVM compiler 3.1
- Server #1: 2x Intel Xeon X5680 3.33 GHz, Nvidia Tesla C2050, Ubuntu 11.10, Driver 290.10
- Server #2: 2x Intel Xeon E5540 2.53 GHz, Nvidia GTX295, Ubuntu 10.04, Driver 295.40

D Kernel Implementation

The following source code is a working kernel to be executed on GPUs:

```
/* C preprocessor defines for the filter functions */
#define ht2bs_4a(a,b,c,d)  (~(((a|b)&c)^(a|d)^b))
#define ht2bs_4b(a,b,c,d)  (~(((d|c)&(a^b))^(d|a|b)))
#define ht2bs_5c(a,b,c,d,e) (~((((((c^e)|d)&a)^b)&(c^b))^(((d^e)|a)&((d^b)|c))))

__kernel void
hitag_worker(__constant const unsigned int *preSlicedLFSR,   // 48*32 bit
             __constant const unsigned int *ivInput,         // 12*32 bit
             __constant const unsigned int *workGroupSlices, // 256*8*32 bit, __global on GF100
             __constant const unsigned int *cmpSlicedArray,  // 32*32 bit
             __global unsigned int *numCollisions)           // [0]=cnt, [>0]=data
{
const unsigned int lid = get_local_id(0);     // 0 ... 255
unsigned int keyChallenge[20];                // store key challenges (note: index is shifted)
unsigned int lfsr[48];                        // store LFSR state
unsigned int bitMask = 0xFFFFFFFF;            // check for valid key candidates

__local unsigned int groupPreComp[12];

if (lid==0) { // Execute the following code only for the first work item of each work group
    unsigned int blockId = get_group_id(0);   // 0 ... ((2^20)/(2^8))

    /* Assign key challenges based on blockId (0 ... 2^12) */
    keyChallenge[ 0] = (blockId&1) ? 0xFFFFFFFF : 0x00000000;
    blockId >>= 1;
    // [...] unrolled code [...]
    keyChallenge[10] = (blockId&1) ? 0xFFFFFFFF : 0x00000000;
    blockId >>= 1;
    keyChallenge[11] = (blockId&1) ? 0xFFFFFFFF : 0x00000000;

    /*
     Continue state randomization and store result in local memory.
     The following code is basically the unrolled version of:

     // for (j = 0; j < 47; j++) lfsr[j] = lfsr[j+1]; // extra shift done on host
     for(i=0; i<12; i++) {
        for (j = 0; j < 47; j++) lfsr[j] = lfsr[j+1];
           lfsr[47] = ivInput[i] ^ keyChallenge[i] ^ f20bs(lfsr);
     }
     */

    groupPreComp[11] = ivInput[ 0] ^ keyChallenge[ 0] ^ ht2bs_5c(
           ht2bs_4a(preSlicedLFSR[ 1],preSlicedLFSR[ 2],preSlicedLFSR[ 4],preSlicedLFSR[ 5]),
           ht2bs_4b(preSlicedLFSR[ 7],preSlicedLFSR[11],preSlicedLFSR[13],preSlicedLFSR[14]),
           ht2bs_4b(preSlicedLFSR[16],preSlicedLFSR[20],preSlicedLFSR[22],preSlicedLFSR[25]),
           ht2bs_4b(preSlicedLFSR[27],preSlicedLFSR[28],preSlicedLFSR[30],preSlicedLFSR[32]),
           ht2bs_4a(preSlicedLFSR[33],preSlicedLFSR[42],preSlicedLFSR[43],preSlicedLFSR[45]));
    groupPreComp[ 0] = ivInput[ 1] ^ keyChallenge[ 1] ^ ht2bs_5c(
           ht2bs_4a(preSlicedLFSR[ 2],preSlicedLFSR[ 3],preSlicedLFSR[ 5],preSlicedLFSR[ 6]),
           ht2bs_4b(preSlicedLFSR[ 8],preSlicedLFSR[12],preSlicedLFSR[14],preSlicedLFSR[15]),
           ht2bs_4b(preSlicedLFSR[17],preSlicedLFSR[21],preSlicedLFSR[23],preSlicedLFSR[26]),
           ht2bs_4b(preSlicedLFSR[28],preSlicedLFSR[29],preSlicedLFSR[31],preSlicedLFSR[33]),
           ht2bs_4a(preSlicedLFSR[34],preSlicedLFSR[43],preSlicedLFSR[44],preSlicedLFSR[46]));
    // [...] unrolled code [...]
    groupPreComp[10] = ivInput[11] ^ keyChallenge[11] ^ ht2bs_5c(
           ht2bs_4a(preSlicedLFSR[12],preSlicedLFSR[13],preSlicedLFSR[15],preSlicedLFSR[16]),
           ht2bs_4b(preSlicedLFSR[18],preSlicedLFSR[22],preSlicedLFSR[24],preSlicedLFSR[25]),
           ht2bs_4b(preSlicedLFSR[27],preSlicedLFSR[31],preSlicedLFSR[33],preSlicedLFSR[36]),
           ht2bs_4b(preSlicedLFSR[38],preSlicedLFSR[39],preSlicedLFSR[41],preSlicedLFSR[43]),
           ht2bs_4a(preSlicedLFSR[44],groupPreComp[ 5],groupPreComp[ 6],groupPreComp[ 8]));
    }
    barrier(CLK_LOCAL_MEM_FENCE); // Synchronize all work items of work group
```

```
/* Hint: Use for-loops for NVIDIA, manually unrolled loops for AMD-GPU. */
for(unsigned int i=0; i<11; i++) { lfsr[i] = groupPreComp[i]; }
lfsr[47] = groupPreComp[11];
for(unsigned int i=11; i<47; i++) { lfsr[i] = preSlicedLFSR[i]; }

keyChallenge[12] = workGroupSlices[lid+(256*0)];  // coalesced if global mem
keyChallenge[13] = workGroupSlices[lid+(256*1)];  // coalesced if global mem
// [...] unrolled code [...]
keyChallenge[19] = workGroupSlices[lid+(256*7)];  // coalesced if global mem

lfsr[11] = keyChallenge[12] ^ ht2bs_5c(ht2bs_4a(lfsr[13],lfsr[14],lfsr[16],lfsr[17]),
                                       ht2bs_4b(lfsr[19],lfsr[23],lfsr[25],lfsr[26]),
                                       ht2bs_4b(lfsr[28],lfsr[32],lfsr[34],lfsr[37]),
                                       ht2bs_4b(lfsr[39],lfsr[40],lfsr[42],lfsr[44]),
                                       ht2bs_4a(lfsr[45],lfsr[ 6],lfsr[ 7],lfsr[ 9]));
lfsr[12] = keyChallenge[13] ^ ht2bs_5c(ht2bs_4a(lfsr[14],lfsr[15],lfsr[17],lfsr[18]),
                                       ht2bs_4b(lfsr[20],lfsr[24],lfsr[26],lfsr[27]),
                                       ht2bs_4b(lfsr[29],lfsr[33],lfsr[35],lfsr[38]),
                                       ht2bs_4b(lfsr[40],lfsr[41],lfsr[43],lfsr[45]),
                                       ht2bs_4a(lfsr[46],lfsr[ 7],lfsr[ 8],lfsr[10]));
// [...] unrolled code [...]
lfsr[18] = keyChallenge[19] ^ ht2bs_5c(ht2bs_4a(lfsr[20],lfsr[21],lfsr[23],lfsr[24]),
                                       ht2bs_4b(lfsr[26],lfsr[30],lfsr[32],lfsr[33]),
                                       ht2bs_4b(lfsr[35],lfsr[39],lfsr[41],lfsr[44]),
                                       ht2bs_4b(lfsr[46],lfsr[47],lfsr[ 1],lfsr[ 3]),
                                       ht2bs_4a(lfsr[ 4],lfsr[13],lfsr[14],lfsr[16]));

read_mem_fence(CLK_LOCAL_MEM_FENCE);

/*
 We now compute 32 bits of output (the authenticator) and immediately
 check the output bits against the expected value. We make special use
 of the NXOR to track the valid key candidates (result in bitMask).

 The following code is basically the unrolled version of:
 for(i=0; i<8; i++) { bitMask &= (~(cmpSlicedArray[i+ 0] ^ hitag2bs_round(lfsr))); }
 if(bitMask!=0)
    for(i=0; i<8; i++) { bitMask &= (~(cmpSlicedArray[i+ 8] ^ hitag2bs_round(lfsr))); }
 if(bitMask!=0)
    for(i=0; i<8; i++) { bitMask &= (~(cmpSlicedArray[i+ 16] ^ hitag2bs_round(lfsr))); }
 if(bitMask!=0)
    for(i=0; i<8; i++) { bitMask &= (~(cmpSlicedArray[i+ 24] ^ hitag2bs_round(lfsr))); }
 */

lfsr[19] ^= lfsr[21] ^ lfsr[22] ^ lfsr[25] ^ lfsr[26] ^ lfsr[27] ^ lfsr[35] ^ lfsr[41] ^
      lfsr[42] ^ lfsr[45] ^ lfsr[1] ^ lfsr[12] ^ lfsr[13] ^ lfsr[14] ^ lfsr[17] ^ lfsr[18];
bitMask &= (~(cmpSlicedArray[0] ^ ht2bs_5c(ht2bs_4a(lfsr[21],lfsr[22],lfsr[24],lfsr[25]),
                                           ht2bs_4b(lfsr[27],lfsr[31],lfsr[33],lfsr[34]),
                                           ht2bs_4b(lfsr[36],lfsr[40],lfsr[42],lfsr[45]),
                                           ht2bs_4b(lfsr[47],lfsr[ 0],lfsr[ 2],lfsr[ 4]),
                                           ht2bs_4a(lfsr[5],lfsr[14],lfsr[15],lfsr[17]))));
// [...] unrolled code [...]
lfsr[26] ^= lfsr[28] ^ lfsr[29] ^ lfsr[32] ^ lfsr[33] ^ lfsr[34] ^ lfsr[42] ^ lfsr[0] ^
      lfsr[1] ^ lfsr[4] ^ lfsr[8] ^ lfsr[19] ^ lfsr[20] ^ lfsr[21] ^ lfsr[24] ^ lfsr[25];
bitMask &= (~(cmpSlicedArray[7] ^ ht2bs_5c(ht2bs_4a(lfsr[28],lfsr[29],lfsr[31],lfsr[32]),
                                           ht2bs_4b(lfsr[34],lfsr[38],lfsr[40],lfsr[41]),
                                           ht2bs_4b(lfsr[43],lfsr[47],lfsr[ 1],lfsr[ 4]),
                                           ht2bs_4b(lfsr[ 6],lfsr[ 7],lfsr[ 9],lfsr[11]),
                                           ht2bs_4a(lfsr[12],lfsr[21],lfsr[22],lfsr[24]))));
/*
 After computing the first 8 output bits, we check if there are any
 key candidates left. If so, continue, else, terminate.
 */
if(bitMask!=0) {
    lfsr[27] ^= lfsr[29] ^ lfsr[30] ^ lfsr[33] ^ lfsr[34] ^ lfsr[35] ^ lfsr[43] ^ lfsr[1] ^
         lfsr[2] ^ lfsr[5] ^ lfsr[9] ^ lfsr[20] ^ lfsr[21] ^ lfsr[22] ^ lfsr[25] ^ lfsr[26];
    bitMask &= (~(cmpSlicedArray[ 8] ^ ht2bs_5c(ht2bs_4a(lfsr[29],lfsr[30],lfsr[32],lfsr[33]),
                                                ht2bs_4b(lfsr[35],lfsr[39],lfsr[41],lfsr[42]),
                                                    ht2bs_4b(lfsr[44],lfsr[ 0],lfsr[ 2],lfsr[ 5]),
                                                    ht2bs_4b(lfsr[ 7],lfsr[ 8],lfsr[10],lfsr[12]),
                                                    ht2bs_4a(lfsr[13],lfsr[22],lfsr[23],lfsr[25]))));
        // [...] unrolled code [...]
    }
    read_mem_fence(CLK_LOCAL_MEM_FENCE);

    if(bitMask!=0){ // first check, then continue computation
       // [...] unrolled code [...]
    }

    read_mem_fence(CLK_LOCAL_MEM_FENCE);

    if(bitMask != 0) { // first check, then continue computation
        lfsr[43] ^= lfsr[45] ^ lfsr[46] ^ lfsr[1] ^ lfsr[2] ^ lfsr[3] ^ lfsr[11] ^ lfsr[17] ^
             lfsr[18] ^ lfsr[21] ^ lfsr[25] ^ lfsr[36] ^ lfsr[37] ^ lfsr[38] ^ lfsr[41] ^ lfsr[42];
        bitMask &= (~(cmpSlicedArray[24] ^ ht2bs_5c(ht2bs_4a(lfsr[45],lfsr[46],lfsr[ 0],lfsr[ 1]),
                                                    ht2bs_4b(lfsr[ 3],lfsr[ 7],lfsr[ 9],lfsr[10]),
                                                    ht2bs_4b(lfsr[12],lfsr[16],lfsr[18],lfsr[21]),
                                                    ht2bs_4b(lfsr[23],lfsr[24],lfsr[26],lfsr[28]),
                                                    ht2bs_4a(lfsr[29],lfsr[38],lfsr[39],lfsr[41]))));
        // [...] unrolled code [...]
        lfsr[2] ^= lfsr[4] ^ lfsr[5] ^ lfsr[8] ^ lfsr[9] ^ lfsr[10] ^ lfsr[18] ^ lfsr[24] ^
             lfsr[25] ^ lfsr[28] ^ lfsr[32] ^ lfsr[43] ^ lfsr[44] ^ lfsr[45] ^ lfsr[0] ^ lfsr[1];
        bitMask &= (~(cmpSlicedArray[31] ^ ht2bs_5c(ht2bs_4a(lfsr[ 4],lfsr[ 5],lfsr[ 7],lfsr[ 8]),
                                                    ht2bs_4b(lfsr[10],lfsr[14],lfsr[16],lfsr[17]),
                                                    ht2bs_4b(lfsr[19],lfsr[23],lfsr[25],lfsr[28]),
                                                    ht2bs_4b(lfsr[30],lfsr[31],lfsr[33],lfsr[35]),
                                                    ht2bs_4a(lfsr[36],lfsr[45],lfsr[46],lfsr[ 0]))));
    }
    read_mem_fence(CLK_LOCAL_MEM_FENCE);

    /* If the bitMask is still non-zero, we have found valid key candidate(s) */
    if(bitMask !=0) {
        numCollisions[0]++; //atomic_inc(&numCollisions[0]); // not necessary, see below
        numCollisions[1] = bitMask;
        numCollisions[2] = get_global_id(0);
        // Eventually, we would have to dynamically adjust the indices. However, in
        // our tests, not more than one possible key candidate per kernel run showed up.
        // As a safety measure, one can use the atomic_inc and check for a counter > 1.
        // Depending on the platform, this causes no overhead and results in same speed.
    }
}
```

For the sake of completeness, we provide the two functions referenced within comments in the above source code:

```
// This function computes a complete round (output and feedback) function of the cipher
 unsigned int hitag2bs_round (unsigned int * lfsr) {
    unsigned int i, y;

    y = lfsr[ 0] ^ lfsr[ 2] ^ lfsr[ 3] ^ lfsr[ 6] ^ lfsr[ 7] ^ lfsr[ 8] ^ lfsr[16] ^ lfsr[22] ^
    lfsr[23] ^ lfsr[26] ^ lfsr[30] ^ lfsr[41] ^ lfsr[42] ^ lfsr[43] ^ lfsr[46] ^ lfsr[47];

    for (i = 0; i < 47; i++) { lfsr[i] = lfsr[i+1]; } // omit if unrolled by adjusting indices

    lfsr[47] = y;
    return f20bs (lfsr);
 }

// This function computes the (bit-sliced) non-linear output function of the cipher
static unsigned int f20bs (const unsigned int *x) {
return ht2bs_5c (ht2bs_4a(x[ 1],x[ 2],x[ 4],x[ 5]),
                 ht2bs_4b(x[ 7],x[11],x[13],x[14]),
                 ht2bs_4b(x[16],x[20],x[22],x[25]),
                 ht2bs_4b(x[27],x[28],x[30],x[32]),
                 ht2bs_4a(x[33],x[42],x[43],x[45]));
}
```