High speed cipher cracking: the case of Keeloq on CUDA

Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi
Dipartimento di Elettronica e Informazione (DEI)
Politecnico di Milano
Via G. Ponzio 34/5, 20133 Milan, Italy
{agosta,barenghi,pelosi}@elet.polimi.it

Abstract: Graphic Processing Units (GPU) are increasingly popular in the field of high-performance computing for their ability to provide computational power for massively parallel problems at a reduced cost. However, the programming model exposed by the GPGPU software development tools is often insufficient to achieve full performance, and a major rethinking of algorithmic choices is needed. In this paper, we showcase such an effect on a case study drawn from the cryptography application domain. The pervasive use of cryptographic primitives in modern embedded systems is a growing trend. Small, efficient cryptosystems have been effectively employed to design and implement keyless password-based access control systems in various wireless authentication applications. The security margin provided by these lightweight ciphers should be accurately examined in light of the speed and area constraints imposed by the target environment. Research on this subject has lead to careful cryptanalyses of ciphers such as CRYPTO-1 (used for micropayment purposes) and KEELOQ (employed in vehicle keys). We present a re-design of the ASIC-oriented KEELOQ implementation to perform efficient exhaustive key search attacks while fitting tightly the parallel programming model exposed by modern GPUs. Indeed, the *bitslicing* technique allows the intrinsic parallelism offered by word-oriented SIMD computations to be effectively exploited. Through proper adaptation of the algorithm implementation to a platform radically different from the one it was designed for, we achieved a $\times 40$ speedup in the computation time w.r.t. a single-core CPU bruteforce attack, employing only consumer grade hardware. The outstanding speedup obtainable points to a significant weakening of the security margin of the cipher, since it proves that anyone with off-the-shelf hardware is able to circumvent the security measures in place.

1 Introduction

In the last years, Graphics Processing Units (GPUs) have raised wide interest as sources of computational power for non-graphical applications, due to the availability of programming models such as CUDA and OpenCL that are vastly more accessible to experts of other domains than graphics rendering APIs (OpenGL and DirectX) [OHL+08]. A major strength of GPGPU-based platform are their appealing cost-performance figures of merit. In recent times even in the field of High Performance Computing there have been major investments to build GPGPU-based supercomputers.

However, there are also factors that hinder the expansion of GPGPU computing, especially the difficulty of programming efficient applications using the available program-

ming models. Special attention must be placed to tailor the application and its algorithmic components to the specific needs of the parallel hardware, e.g. by minimizing control flow divergence and exposing as much parallelism as possible while minimizing synchronization overheads [OHL+08]. In this paper, we show how the use of specialized techniques can lead to large speedups, thus allowing the GPU to contend on an equal or favorable base (in terms of computation throughput per euro) with solutions based on CPUs or reconfigurable hardware.

The field of cryptography has been explored since the first GPGPU attempts using graphics rendering APIs [HW07]. Especially, code breaking is attractive, because it requires vast amounts of computational power. We use as a case study the KEELOQ algorithm [Mic11], which is used in remote keyless entry systems (e.g., vehicle doors or building entrances) or as authentication mechanism in wireless protocols.

Remote keyless entry systems are based on a password based access control mechanism realized through the unidirectional transmission between a secure token (*encoder*) and a receiver (*decoder*). Unauthorized accesses are possible when the encoded password (*access code*) is fixed or it is derived from a relatively low number of possible combinations. In order to prevent this kind of threat, KEELOQ is employed in the so-called *rolling code* (also known as *hopping code*) mode of operation. The basic idea is to have the access code change each time it is used through picking it from a sequence of codewords that cannot be predicted even knowing a very large number of previously used ones. The generation of such a sequence is based on the definition of both a uni-directional command transfer protocol and an encryption engine to provide the codewords to be transmitted. From an operational point of view, the information transmitted by the encoder is composed by two parts: the code-hopping part (which changes each time the encoder is activated) and a second un-encrypted part, principally containing the encoder serial number, used for identifying the transmitter at a receiving decoder. To this end, the receiver decrypts the codeword, and compares the recovered counter value with its internal one, and the recovered serial number with the one received along with the codeword. If both values match, the token is granted access.

Algorithms such as KEELOQ are designed for dedicated hardware implementation, since the target devices (remote controllers) are manufactured as very low cost ASICs. So, their direct implementation in software has much lower performances – which, in principle, makes it easier to carry out an attack using configurable hardware such as FPGAs. However, we show how the introduction of a level of parallelism not commonly seen in GPGPU algorithm design, *bit-level* parallelism, can lead to a $\times 40$ speedup over a CPU core.

The rest of this paper is organized as follows. Section 2 introduces the KEELOQ cipher, while Section 3 reviews the characteristics of the NVIDIA GPU families target in this study, as well as the SIMT programming model as implemented by the CUDA development tools. Section 4 describes the design of our solution and Section 5 provides the experimental evaluation on the case study. Finally, Section 6 outlines the most closely related works, while Section 7 draws some conclusions.

2 The KEELOQ Cipher

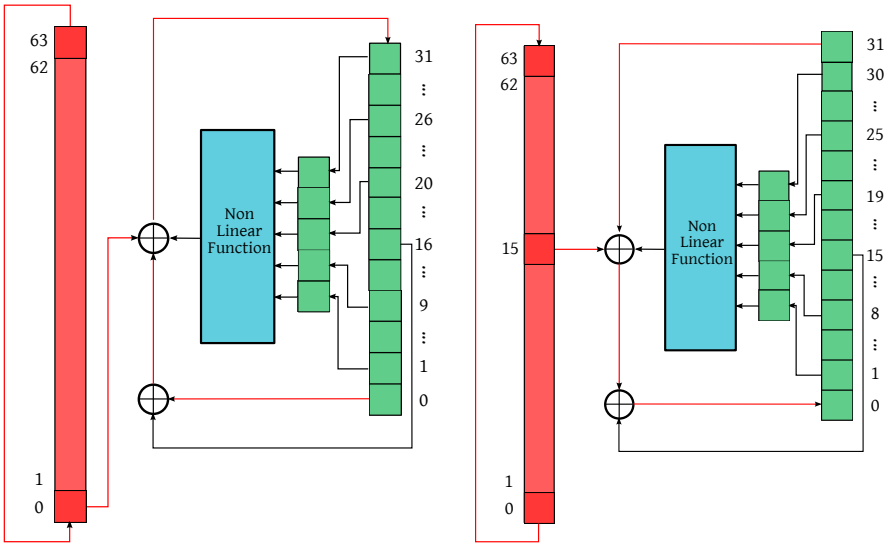
Remote keyless entry systems such as the ones employed in building doors and vehicles require a small and power efficient cipher in order to build a challenge-response authentication mechanism. The class of cryptographic primitives commonly employed in this scenario is the one of stream ciphers.

A stream cipher outputs a pseudorandom sequence of bits, known as keystream, depending on the values of the inner state of the algorithm, which is never disclosed. The inner state is initialized employing a secret value, the key, during a bootstrap phase, while the keystream is not emitted. The encryption is performed via a bitwise `xor` of the keystream with the plaintext: since this operation acts bitwise, there is no need to pad the plaintext message, regardless of its length. One of the most common way to generate a pseudorandom keystream is to employ a Feedback Shift Register (FSR): this component is basically a shift register where the contents are shifted of a single bit per clock cycle, and the new bit entering the register is generated as a function of the previous content of it. Subsequently, one bit of the keystream is derived as a function of the new contents of the register for every cycle. Even if linear feedback functions can be used in order to compute the feedback bit (such FSRs are thus denominated Linear Feedback Shift Registers), it is not cryptographically safe to do so, as it is possible to completely rebuild the structure and content of the LFSR only from the output keystream, provided also the output function is linear. Following the previous consideration, the FSR employed to build stream ciphers are endowed with a Non-Linear Feedback function and/or a non-linear output function. Non-Linear Feedback Shift Register (NLFSR) are nowadays largely employed in modern ciphers [1], as they provide a very favorable tradeoff between area and performances, while retaining a good security margin. NLFSRs are known to be more resistant to cryptanalytic attacks than linear FSRs, although building NLFSRs with guaranteed pseudo-randomness properties (such as providing an exact bound on the length of the period in the output bits) is still an open problem.

KEELOQ is the most scrutinized encryption engine used in remote keyless entry systems. It is a proprietary hardware-dedicated NLFSR-based block cipher, designed by G. Kuhn taking strong inspiration from stream cipher design techniques. Figure 1 shows the internal structure of the KEELOQ cipher: the secret key is stored in the red register on the left and is at most 64-bit wide. The key register is a FSR, and the key is mixed with the output of the state one bit per clock cycle. The 32-bit long NLFSR on the right hand side constitutes the nonlinear component of the cipher providing its effective security margin. Five bits of the NLFSR are combined together by means of a non linear function described by an equation over $\mathbb{Z}_2$ among five bits of the status register. The non linear function outputs a single bit per clock cycle, which is added to the aforementioned key bit and to b_{16} and b_0, and employed as the feedback bit of the NLFSR. To encrypt a 32-bit plaintext block, the NLFSR is initialized with the value of the plaintext, and subsequently the entire system is clocked 528 times. After the 528 updates of both registers, the content of the NLFSR is the final ciphertext.

The most common mode of operation for KEELOQ is the so-called *hopping code*, in a sce-

[1] The eSTREAM Project, `http://www.ecrypt.eu.org/stream/`



(a) Encryption (b) Decryption

Figure 1: Keeloq Cipher

nario where a remote *encoder* transmits a codeword to the authorizing *decoder* (receiver). This mode of operation involves encrypting a plaintext built out of a counter and a unique identifier (ID) of the encoding device. Every time a new 32-bit codeword (i.e. a ciphertext block) must be generated, the counter is incremented and the new plaintext is encrypted. Then, the codeword is transmitted along with the encoding device ID. The secret 64-bit key of any encoder is generated through the decoder engine as a pair of 32-bit codewords. Such a procedure implies that the decoder is able to generate the secret keys for a number of encoders starting from: *(i)* an embedded 64-bit master key (which is fixed by the manufacturer of the keyless entry system), *(ii)* the ID of the encoding device, *(iii)* and a random seed composed by 32, 48 or 60 bits.

A potential attacker may retrieve the master key from the decoding device (receiver) and eavesdrop the ID of an encoder when it is transmitted along with a codeword. Therefore, the use of a secret random seed in the secret key generation phase avoid the leakage of the secret key of the targeted encoder.

A brute-forcing attack aimed at recovering the secret key of the transmitting encoder (EK) employs two consecutively transmitted codewords, each of which is bound to the encoder ID. The attacker computes a candidate 64-bit value for EK through guessing on the bits of the random seed, while the value of the remaining part of the secret key is easily derived from the specification of the key generation protocol. Subsequently, she checks the ID value resulting from the decryption of the first codeword, and whether a match is found, the output derived from the decryption of the second codeword (employing the same EK) is used as a confirmatory step.

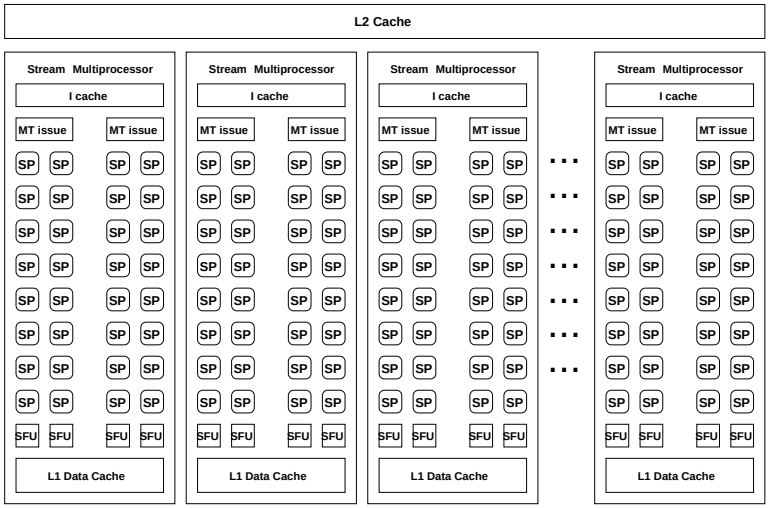


Figure 2: Overview of the NVIDIA GTX470 (Fermi) streaming processors architecture: each stream multiprocessor (SM) contains 32 streaming processors (SP), plus four special function units (SFU). A configurable L1 cache/shared memory is local to each stream multiprocessor, while L2 cache is shared among the entire set of SM. Up to 16 SM can be present in a single unit.

3 General Purpose Computing with GPUs

The GPGPU devices targeted in this work are based on the NVIDIA GT200 and Fermi architectures. Figure 2 shows a sketch of the NVIDIA GTX470 (Fermi) streaming processor array. A streaming multiprocessor (SM) contains 32 streaming processors, four special functional units and a multithreaded instruction issue unit (respectively indicated as SP, SFU and MT-Issue in Figure 2. This is a fourfold increase over the GT200 SMs. A streaming multiprocessor concurrently executes two groups of 32 threads called *warps*, for a total of 64 concurrent threads. Since each thread in a warp has its own control flow, their execution paths may diverge due to the independent evaluation of conditional statements; when this happens, the warp serially executes each path. Each multiprocessor executes warps much like the *Single Instruction Multiple Data* (SIMD) paradigm, as every thread is assigned to a different SP and every active thread executes the same instruction on different data. Finally, the Fermi architecture includes both L1 and L2 cache memories, with the L1 configurable between cache and shared memory behaviours and shared by the SPs in a single SM, and the L2 shared among all SMs in the device. The earlier GT200 only has a fast shared memory shared within each SM.

GPGPU computing requires the programmer to manage a heterogeneous system (CPU host plus GPU device) as well as to handle the massive parallelism exposed by the GPU hardware. The Compute Unified Device Architecture (CUDA) [NBGS08, NVI08], proposed by NVIDIA for its graphics processors starting with the G80 series [LNOM08], exposes a programming model that integrates host and GPU code in the same C++ source files. On the GPU device side, a *Single Instruction, Multiple Threads* programming model

is exposed, where a single *kernel* is executed by a user-specified number of threads. Every CUDA kernel is explicitly invoked by host code and executed by the device, while the host-side code continues the execution asynchronously after instantiating the kernel. On the host side, a specific synchronizing function call is provided to wait for the completion of the active asynchronous kernel computation.

The CUDA programming model abstracts the actual parallelism implemented by the hardware architecture, providing the concepts of *block* and *thread* to express concurrency in algorithms. A block captures the notion of a group of concurrent threads. Blocks are required to execute independently, so that it has to be possible to execute them in any order (in parallel or in sequence). Therefore, the synchronization primitives semantically act only among threads belonging to the same block. Intra-block communications among threads use the *logical shared memory* associated with that block.

Since the architecture does not provide support for message-passing, threads belonging to different blocks must communicate through *global memory*. The global memory is entirely mapped to the off-chip memory. The concurrent accesses to logical shared memory by threads executing within the same block are supported through an explicit barrier synchronization primitive.

A kernel call-site must specify the number of blocks as well as the number of threads within each block when executing the kernel code. The current CUDA programming model imposes a capping of 512 threads per block.

The mapping of threads to processors and of blocks to multiprocessors is mainly handled by hardware controller components. Two or more blocks may share the same multiprocessor through mechanisms that allow fast context switching depending on the computational resources used by threads and on the constraints of the hardware architecture. The number of concurrent blocks managed by a single multiprocessor is currently limited to 8.

In addition to the logical shared memory and the global memory, in the CUDA programming model each thread may access a *constant* memory. An access to this read-only memory space is faster than one to global memory, provided that there is sufficient access locality since constant memory is implemented as a region of global memory fit with an on-chip cache. Finally, another portion of the off-chip memory may be allocated as a *local memory* that is used as thread private resource. Since the local memory access is slow, the shared memory also serves as an explicitly managed cache – though it is up to the programmer to warrant that the local data being saved in shared memory are not accessed by other threads. Shared memory comes in limited amounts (threads within each block typically share 16 KB of memory) hence, it is crucial for performance for each thread to handle only small chunks of data.

Note that while the OpenCL language and API [Khr11] are gaining momentum as the industry standard in programming heterogeneous platforms composed of host CPUs and programmable accelerators, including GPGPUs, the implementations provided are still not mature enough to compete, on NVIDIA devices, with the vendor-specific software development tools. However, the programming model provided in OpenCL is, as far as GPGPU programming goes, essentially based on the same principles as the SIMT model exposed in CUDA, so the techniques and results shown in this work can be easily extended to OpenCL-driven devices.

4 Adaptation to Parallel Architectures

Many-core architectures offer large amount of parallel computing power by supplying the developer with hundreds of processing cores, each endowed with limited resources. In GPGPU, key resource limitations include:

Control flow divergence as multiple divergent control flows can be handled safely from the point of view of functionality, but with major performance losses as parallelism is inhibited along the different control flows – essentially, divergent flows of control are serialized, regardless of the data dependences among the divergent threads (which may well be non-existent). This limitation is due to the hardware design of GPGPU, where the processors in a multiprocessor unit are bound to the same program counter.

Local memory availability as a limited amount of very fast local memory must be shared among numerous processing elements. While the sharing allows fast communication among the processing elements, the local memory is much more useful when used in a read-only way, or partitioned for local use by each processing element, since true shared accesses still require costly synchronization operations, and are often difficult to code.

To exploit such parallel computing power, the critical issue is to be able to express a given application or algorithm in a form amenable to parallel execution on the target device. The literature reports three main sources of parallelism, which can be exploited with different degrees of success on various types of parallel architectures:

Thread-level parallelism is obtained when two or more tasks (regions of code with independent control flow) can be executed in parallel with few or no data dependencies (in the former case, synchronizations will be needed within each task, in the latter the synchronization point will be the end of the tasks). Thread-level parallelism is exposed by complex applications, where multiple independent tasks are performed, and is best exploited on symmetric multiprocessors, where each processor is endowed with sufficient resources to executed its assigned task. It is not suited for GPGPUs, since control flow divergence is a major factor for performance reduction in these architectures.

Loop-level parallelism is found in parallel loop constructs, where each iteration of the loop is data-independent from the others (or has limited synchronization requirements). Loop level parallelism is an excellent fit for vector processors, SIMD processors and GPGPUs, since control is fixed and identical for all iterations (barring nested conditionals, which can often be transformed to predicated code).

Instruction-level parallelism is achieved at the finest of the three common granularities, where independent instructions can be parallelized. It is commonly exploited by superscalar and Very Long Instruction Word architectures, but, like Thread-level parallelism, it is unsuitable for GPGPU due to the need to executed different instructions in parallel, rather than the same instruction of different data.

It would therefore seem that Loop-level parallelism is the only viable choice for GPGPUs, but this model is not exposed by many types of codes. A typical example are encryption primitives designed for hardware implementation. In this case, parallelism is rarely available, but this is not an issue, since the implementation is performed through dedicated ASIC, and may be even considered a benefit, since software implementations are often

aimed at *breaking* the encryption through brute force attacks. The usage of GPGPUs to perform brute force attacks is well-documented, but is often limited to mere juxtaposition of several encryption operations with different keys.

However, it is possible to push the parallelization further, by introducing an entirely different level of parallelism, *Bit-level parallelism*. Here, the goal is to parallelize operations at the single bit level, thereby obtaining remarkably uniform parallel operations. This technique is know as *bitslicing* [Bih97].

Bitslicing refers to a software technique of using a general purpose CPU to implement Single Instruction Multiple Data (SIMD) operations. The enforced strategy consists of packing the bit values belonging to different operands within a single register and of using general-purpose arithmetic/logic instructions as specialized virtual processing engines designed for SIMD operations at bit level.

Most of the symmetric cryptographic primitives are designed to process input data at bit level. Therefore, the software implementations of such algorithms on not-specialized architectures may greatly benefit from the application of the bit-slicing strategy as long as the underlying hardware resources in terms of number of registers are easily available. Biham [Bih97] achieved significant gains in performance of DES by using this method.

In the case of KEELOQ breaking, the bitslicing technique is employed in the following way. To break the cipher by brute force, we need to try the encryption of the same plaintext using all possible keys. The plaintext, in the original version, is a 32-bit word. We expand it to a 32-words array, where the i-th word in the array is $0xFFFFFFFF$ if the i-th bit of the original plaintext was 1, or $0x00000000$ otherwise. We then generate the keys (each of 64 bits) starting with the $0x0000000000000000$ key and progressively increasing its value. Each 64-words array generated in this way has the five last words (corresponding to the lower bits of the original keys) always correspond to the encoding of the same 32 values which are added to a "base" key value, which increases in steps of 32.

Thus, the number of parallel encryption runs is 32 per thread, with configurable number $n_{threads}$ of threads per each CUDA block. Overall, a grand total of $32 \times n_{threads} \times n_{blocks}$ encryption runs are performed at every time by the GPU.

5 Experimental Results

We implemented a fully bitsliced version of the Keeloq cipher both employing the CUDA programming model and pure C. The pure C version has been run on the host CPU to provide a reference implementation as far as throughput goes.

5.1 Experimental settings

The running environment where the bruteforcing speed tests were performed is an Intel Core i7 920 based system with 12Gb DDR3 DRAM, running Gentoo Linux AMD64. All the GPU binaries were compiled employing `nvcc` 4.0 from nVidia CUDA toolkit 4.0,

while the CPU baseline versions were compiled with `gcc` 4.4.6. The bitsliced implementation of the cipher has been tested on two different GPUs, which have been mounted as the only device on the 16 lane PCI-Express 2.0 port available on the motherboard in order to test the difference in performances. The first GPU card is a GeForce GTX 260 equipped with 894 Mb of GDDR5 video RAM and 192 CUDA cores, while the second card employed for testing is a GeForce GTX 470 with 448 CUDA cores and 1280 MB of GDDR5 video RAM.

5.2 Performance Evaluation

An important step in the evaluation of the performances of our bitsliced implementation of Keeloq on CUDA is the exploration of two parameters: the number of threads composing a CUDA block and the number of blocks constituting a CUDA kernel call. The first parameter regulates the level of register pressure on the shared register file of the streaming multiprocessor and the number of warps into which a CUDA block is split. Since the basic execution unit of a streaming multiprocessor is a single warp, the choice of the number of threads should consider only multiples of 32 to achieve the best fit. The level of register pressure on the Fermi architecture is dictated by the fact that the 32768 registers are shared among the contexts of up to 3 different blocks which can be scheduled on the same streaming multiprocessor. In addition to this, the SMP issue unit of the Fermi architecture is able to dual issue warps, thus it is necessary to keep twice the contexts in the registers. Combining these data with the fact that a single bitsliced keeloq breaking thread employs at most 45 registers, we obtain a SMP register pressure which can be computed as $270 \times n_{threads}$. The second parameter to be chosen regulates the level of global computational load imposed on the GPU. The main point in choosing this parameter is provide at least enough computations to the GPU so that no SMPs remain idle. Moreover, since the SMP issue unit is able to interleave different blocks in order to hide global memory access latencies, it is wise to provide extra workload to the GPU to exploit this feature. These two considerations pointed to the creation of a CUDA kernel as large as possible with architectures up to the GT200, since the static scheduling of the blocks on the SMPs did not account for extra time overhead. With the introduction of a new scheduler for multiple kernels on the Fermi architecture, this consideration may not be still valid.

Figure 3 reports the results of the exploration of the implementation parameter space: coherently with the previous considerations, the best solution is reached with 128 threads per block (34560 employed registers), when the number of blocks per SMP is enough to fill all the issue queues completely. Raising further the number of blocks per kernel leads to a decrease in performances which can be ascribed to the extra context switching effort imposed on the new scheduler. As expected also raising further the number of threads per block leads to a significant decrease in throughput due to the hindering of context switches caused by the frequent register spills and fills.

An analogous exploration campaign has been lead also on the GTX260 card, yielding 64 threads per block as the best performing choice of the parameter. This choice is coherent with the fact that the shared register file of the GTX260 is 16384 since 64 threads per block allow the issue unit of the streaming multiprocessor to perform the context switching

!th

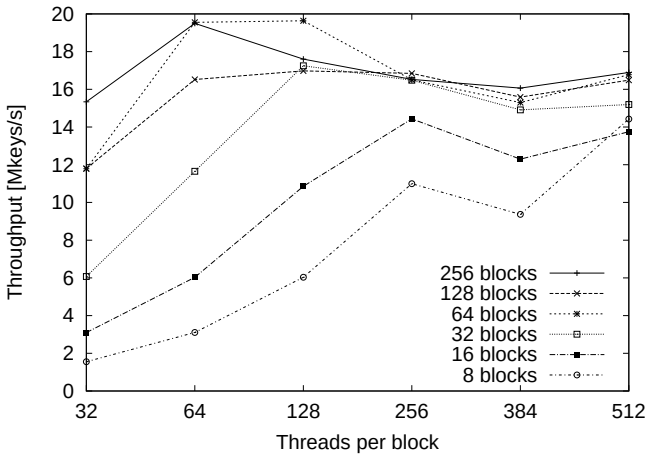


Figure 3: Throughput of the bitsliced implementation of the Keeloq breaker on the Geforce GTX470 card, related to the number of threads per block and the number of blocks per CUDA kernel invocation

Seed Length	Single Core Core i7 920 [h]	Four Cores Core i7 920 [h]	Single GPU GTX260 [h]	Single GPU GTX470 [h]
32	2.6	0.73	0.14	0.04
48	$1.73\,10^5$	$4.84\,10^4$	$9.45\,10^3$	$3.98\,10^3$
60	$7.08\,10^8$	$1.98\,10^8$	$3.81\,10^7$	$1.63\,10^7$
Throughput	$4.51\,10^5$	$1.61\,10^6$	$8.27\,10^6$	$1.96\,10^7$

Table 1: Expected timings to run an exhaustive search of the correct seed for key generation for the different platforms. The

between the three blocks in queue without the need to spill part of the register file to the global memory. In this case, however, increasing arbitrarily the number of blocks per kernel did not induce any performance penalty as expected from the GT200 architecture.

After choosing the optimal number of threads per block and blocks per kernel invocation, we evaluate the effective time needed in order to break the Keeloq key generation mechanism, with respect to the length of the employed seed. Table 1 reports the expected running times of an attack, depending on the chosen platform to perform the exhaustive search. Taking as a reference value the throughput obtained by the bitsliced implementation of Keeloq running on the host CPU (419430 keys/s), we notice that employing a 32 bit seed for the key generation does not yield a sufficient security margin, as the remote key can be recovered in 3 hours of computation. The bitsliced implementations running on the GTX260 and GTX470 GPUs achieve a $\times 20.5$ and a $\times 43.5$ speedup respectively, allowing a possible attacker to breach even the security of the 48 bit seed key generation

mechanism in a few months. Since the exhaustive search can be split over multiple GPUs, it is possible to lower the attack time to a single week, while keeping the cost envelope of the equipment below $10000, as this budget allows an attacker to build a 20 GTX 470 cluster with the current market prices.

6 Related Work

The first cryptanalysis of KEELOQ is presented in [Bog07]. The attack is based on the *slide technique* and a linear approximation of the non-linear Boolean function used in the cryptographic engine. The attack requires 2^{52} encryptions, 16GB of storage and the entire codebook, i.e., 2^{32} known plaintexts. In [IKD$^+$08] the authors introduce a specific key recovery attack against KEELOQ which combines the technique of slide attacks with a novel meet-in-the-middle approach. Their method requires 2^{16} chosen plaintexts and has a time complexity of $2^{44.5}$ encryptions which results in about two days of computation employing 50 dual core CPUs at the cost of approximately 10Keuro. The widely adoption of KEELOQ in practice, paved the way to side-channel analysis as a further viable option for attacking chips that implement it. In [EKM$^+$08] the first successful DPA and DEMA attacks on KEELOQ implementations applied to both Identify Friend or Foe (IFF) and code hopping devices, are presented. The attack is prevented if a 60-bit seed value, with good random properties, is employed for the key derivation. Nevertheless, considering the other commonly implemented options of the cipher, the authors reported how to reveal a manufacturer key from a receiver using a few 1000 power traces, and how to recover the device key of a remote control with as few as 10 traces. In [CBW08] the authors apply algebraic techniques to cryptanalyze the cipher. This attack employs the entire codebook, 2^{27} encryptions and has an estimated success probability of 44%. The results of a brute-force attack, implemented on the FPGA-based code-breaker COPACOBANA, are reported in [NK09]. The authors claim the secret key recovery of a remote control in less than 0.5 seconds if a 32-bit seed is used and in less than 6 hours in case of a 48-bit seed. The case of a 60-bit seed needs in the worst case about 1011 days at the cost of approximately 10Kdollars.

7 Conclusions

In this paper, we showed how to carry a brute force attack on the Keeloq cipher, popular in remote keyless entry systems, by exploiting the vast amount of parallelism exposed by modern graphic processing units. We proposed a full redesign of the computation strategy from the original hardware implementation-oriented algorithm to reach high performance in parallel software, by exploiting SIMD techniques down to the bit level. We report a speedup of $\times 40$ speedup in the computation time with respect to a CPU brute force attack, even though only consumer-grade hardware is used.

References

[Bih97] Eli Biham. A Fast New DES Implementation in Software. In Eli Biham, editor, *FSE*, volume 1267 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 1997.

[Bog07] Andrey Bogdanov. Linear Slide Attacks on the KeeLoq Block Cipher. In Dingyi Pei, Moti Yung, Dongdai Lin, and Chuankun Wu, editors, *Inscrypt*, volume 4990 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2007.

[CBW08] Nicolas Courtois, Gregory V. Bard, and David Wagner. Algebraic and Slide Attacks on KeeLoq. In Kaisa Nyberg, editor, *FSE*, volume 5086 of *Lecture Notes in Computer Science*, pages 97–115. Springer, 2008.

[EKM+08] Thomas Eisenbarth, Timo Kasper, Amir Moradi, Christof Paar, Mahmoud Salmasizadeh, and Mohammad T. Manzuri Shalmani. On the Power of Power Analysis in the Real World: A Complete Break of the KeeLoqCode Hopping Scheme. 5157:203–220, 2008.

[HW07] Owen Harrison and John Waldron. AES Encryption Implementation and Analysis on Commodity Graphics Processing Units. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 209–226. Springer, 2007.

[IKD+08] Sebastiaan Indesteege, Nathan Keller, Orr Dunkelman, Eli Biham, and Bart Preneel. A Practical Attack on KeeLoq. In Nigel P. Smart, editor, *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2008.

[Khr11] Khronos OpenCL Working Group. OpenCL - The open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`, Jan 2011.

[LNOM08] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *Micro, IEEE*, 28(2):39 –55, 2008.

[Mic11] Microchip Technology Inc. Security and Authentication Design Center – KEELOQ©3 Development Kit. `http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2074`, Dec 2011.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, March 2008.

[NK09] Martin Novotny and Timo Kasper. Cryptanalysis of KeeLoq with COPACOBANA. In *Workshop on Special Purpose Hardware for Attacking Cryptographic Systems (SHARCS'09)*, 2009.

[NVI08] NVIDIA Corporation. CUDA Technology. `http://www.nvidia.com/CUDA`, September 2008.

[OHL+08] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879 –899, May 2008.