ECRYPT II

Information Society Technologies

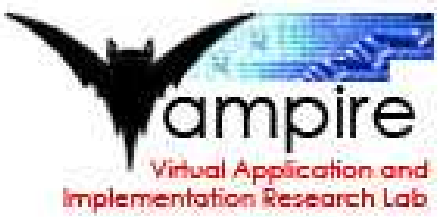
SHARCS '09

Special-purpose Hardware for Attacking Cryptographic Systems

9–10 September 2009

Lausanne, Switzerland

Organized by



within

ECRYPT II European Network of Excellence in Cryptography

Program committee:

Daniel J. Bernstein, University of Illinois at Chicago, USA
Roger Golliver, Intel Corporation, USA
Tim Güneysu, Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, Germany
Marcelo E. Kaihara, École Polytechnique Fédérale de Lausanne, Switzerland
Tanja Lange, Technische Universiteit Eindhoven, The Netherlands
Arjen Lenstra, École Polytechnique Fédérale de Lausanne, Switzerland
Christof Paar, Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, Germany
Jean-Jacques Quisquater, Université Catholique de Louvain, Belgium
Eran Tromer, Massachusetts Institute of Technology, USA
Michael J. Wiener, Cryptographic Clarity, Canada

Subreviewers:

Frederik Armknecht
Maxime Augier
Joppe Bos
Behnaz Bostanipour
Jorge Guajardo
Timo Kasper
Thorsten Kleinjung
Dag Arne Osvik
Onur Özen
Christine Priplata
Juraj Šarinay
Colin Stahlke
Robert Szerwinski

Local organization:

Martijn Stam, École Polytechnique Fédérale de Lausanne, Switzerland

Invited speakers:

Peter Alfke, Xilinx, USA
Shay Gueron, University of Haifa and Intel Corporation, Israel

Contributors:

Jean-Philippe Aumasson, FHNW, Switzerland
Daniel V. Bailey, RSA, USA
Brian Baldwin, University College Cork, Ireland
Lejla Batina, Katholieke Universiteit Leuven, Belgium
Daniel J. Bernstein, University of Illinois at Chicago, USA
Peter Birkner, Technische Universiteit Eindhoven, Netherlands
Joppe W. Bos, École Polytechnique Fédérale de Lausanne, Switzerland
Johannes Buchmann, Technische Universität Darmstadt, Germany
Hsueh-Chung Chen, National Taiwan University, Taiwan
Ming-Shing Chen, Academia Sinica, Taiwan
Chen-Mou Cheng, National Taiwan University, Taiwan
Giacomo de Meulenaer, Université Catholique de Louvain, Belgium
Itai Dinur, Weizmann Institute, Israel
Junfeng Fan, Katholieke Universiteit Leuven, Belgium
Tim Güneysu, Ruhr-Universität Bochum, Germany
Frank Gurkaynak, ETH Zürich, Switzerland
Luca Henzen, ETH Zürich, Switzerland
Jens Hermans, Katholieke Universiteit Leuven, Belgium
Chun-Hung Hsiao, Academia Sinica, Taiwan
Marcelo E. Kaihara, École Polytechnique Fédérale de Lausanne, Switzerland
Timo Kasper, Ruhr-Universität Bochum, Germany
Thorsten Kleinjung, École Polytechnique Fédérale de Lausanne, Switzerland
Tanja Lange, Technische Universiteit Eindhoven, Netherlands
Zong-Cing Lin, National Taiwan University, Taiwan
Willi Meier, FHNW, Switzerland
Nele Mentens, Katholieke Universiteit Leuven, Belgium
Peter L. Montgomery, Microsoft Research, USA
Ruben Niederhagen, RWTH Aachen, Germany
Martin Novotný, Czech Technical University in Prague, Czech Republic
Christof Paar, Ruhr-Universität Bochum, Germany
Christiane Peters, Technische Universiteit Eindhoven, Netherlands
Gerd Pfeiffer, Christian-Albrechts-University of Kiel, Germany
Bart Preneel, Katholieke Universiteit Leuven, Belgium
Francesco Regazzoni, Université Catholique de Louvain, Belgium
Manfred Schimmler, Christian-Albrechts-University of Kiel, Germany
Michael Schneider, Technische Universität Darmstadt, Germany
Peter Schwabe, Technische Universiteit Eindhoven, Netherlands
Igor Semaev, University of Bergen, Norway
Adi Shamir, Weizmann Institute, Israel
Leif Uhsadel, Katholieke Universiteit Leuven, Belgium
Gauthier van Damme, Katholieke Universiteit Leuven, Belgium
Frederik Vercauteren, Katholieke Universiteit Leuven, Belgium
Bo-Yin Yang, Academia Sinica, Taiwan

Program and table of contents:

Wednesday September 9

Thursday September 10

Three Years of Evolution: Cryptanalysis with COPACOBANA

Tim Güneysu[1], Gerd Pfeiffer[2], Christof Paar[1], Manfred Schimmler[2]
[1] Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
{gueneysu,cpaar}@crypto.rub.de
[2] Institute of Computer Science and Applied Mathematics, Faculty of Engineering,
Christian-Albrechts-University of Kiel, Germany
{gp,masch}@informatik.uni-kiel.de

Abstract

In this paper, we review three years of development and improvements on COPACOBANA, the probably most popular, reconfigurable cluster system dedicated to the task of cryptanalysis. Latest changes on the architecture involve modifications for larger and more powerful FPGA devices with dedicated 32 MB of external RAM and point-to-point communication links for improved data throughput. We outline how advanced cryptanalytic applications, such as Time-Memory Tradeoff (TMTO) attacks or attacks on asymmetric cryptosystems, can benefit from these new architectural improvements.

1 Introduction

The security of symmetric and asymmetric ciphers is usually determined by the size of their security parameters, in particular the key-length. Hence, when designing a cryptosystem, these parameters need to be chosen according to the assumed computational capabilities of an attacker. Depending on the chosen security margin, many cryptosystems are potentially vulnerable to attacks when the attacker's computational power increases unexpectedly. In real life, the limiting factor of an attacker is often the financial resources. Thus, it is quite crucial from a cryptographic point of view to not only investigate the complexity of an attack, but also to study possibilities to lower the cost-performance ratio of attack hardware. For instance, a cost-performance improvement of an attack machine by a factor of 1000 effectively reduces the key lengths of a symmetric cipher by roughly 10 bit (since $1000 \approx 2^{10}$). Many cryptanalytical schemes spend their computations in independent operations, which allows for a high degree of parallelism. Such parallel functionality can be realized by individual hardware blocks that operate simultaneously, improving the running time of the overall computation by a perfect linear factor. At this point, it should be remarked that the high non-recurring engineering costs for ASICs have put most projects for building special-purpose hardware for cryptanalysis out of reach for commercial or research institutions. However, with the recent advent of low-cost programmable ICs which host vast amounts of logic resources, special-purpose cryptanalytical machines have now become a possibility outside government agencies.

In this work we review the evolution of a special-purpose hardware system which provides a cost-performance that can be significantly better than that of recent PCs (e.g., for the exhaustive key search on DES). The hardware architecture of this Cost-Optimized Parallel Code Breaker (COPACOBANA) was initially introduced on the SHARCS and CHES workshops in 2006 [24, 25]. In this contribution we will describe further research on cryptanalytical applications over

the last three years and ongoing improvements on the hardware to cope with new requirements of these advanced applications.

The original prototype of the COPACOBANA cluster consists of up to 120 FPGA nodes which are connected by a shared bus providing an aggregate bandwidth of 1.6 Gbps on the backplane of the machine. COPACOBANA is not equipped with dedicated memory modules, but offers a limited number of RAM blocks inside each FPGA. Furthermore, COPACOBANA is connected to a host PC with a single interface to control all operations and provide a little amount of I/O data.

In the following sections, we present cryptanalytic case studies for a large variety of attacks which all make use of the COPACOBANA cluster system. Examples for these case studies include exhaustive key search attacks on the Data Encryption Standard (DES) blockcipher and related systems (e.g., Norton Diskreet), the electronic passport as well as the GSM mobile phone encryption based on the A5/1 streamcipher. More advanced attacks with COPACOBANA comprise implementations for integer factorization (with the Elliptic Curve Method), computations on elliptic curve discrete logarithms and Time-Memory Tradeoffs (TMTO). We briefly review attack implementations and compile a list of improvements on hardware level that can lead to improved performance. Finally, we present a modified cluster architecture which addresses most of the determined issues and promises excellent performance results for the next generation of cryptanalytic applications.

The paper is structured as follows: we begin with a brief review of the original COPACOBANA architecture and a list of case studies on cryptanalytic attacks in Section 3. Next, we present the modified cluster architecture with improvements based on our findings in the previous section. We conclude with an outlook on future cryptanalysis based on COPACOBANA.

2 Architecture of COPACOBANA

Our first prototype of an Cost-Optimized Parallel Code Breaker (COPACOBANA) was produced for less than € 10,000 (material and manufacturing costs only). It was primarily designed for applications and simple cryptanalytic attacks with high compuational complexity but minimal requirements on communications and local memory. In addition to that, it assumes that the computationally expensive operations are inherently parallelizable, i.e., single parallel instances do not need to communicate with each other. The design for limited communication bandwidth was driven by the fact that the computation phase heavily outweighs the data input and output phases. In fact, COPACOBANA was designed for applications in which processes are computing most of the time, without any input or output. Communication was assumed to be almost exclusively used for initialization and reporting of results. A central control instance for the communication can easily be accomplished by a conventional (low-cost) PC, connected to the FPGAs on the cluster by a simple interface. Furthermore, simple brute-force attacks typically demand for very little memory so that we considered the available memory on low-cost FPGAs (such as the Xilinx Spartan-3 devices) to be sufficient.

Recapitulating, COPACOBANA consists of many independent low-cost FPGAs, connected to a host-PC via a standard interface, e.g., USB or Ethernet. The benefit of such a standard interface is the easy scalability and to attach more than one COPACOBANA device to a single host-PC. Note that the initialization, control of FPGAs, and the accumulation of results is done by the host. All time-critical computations such as the cryptanalytical core tasks are performed

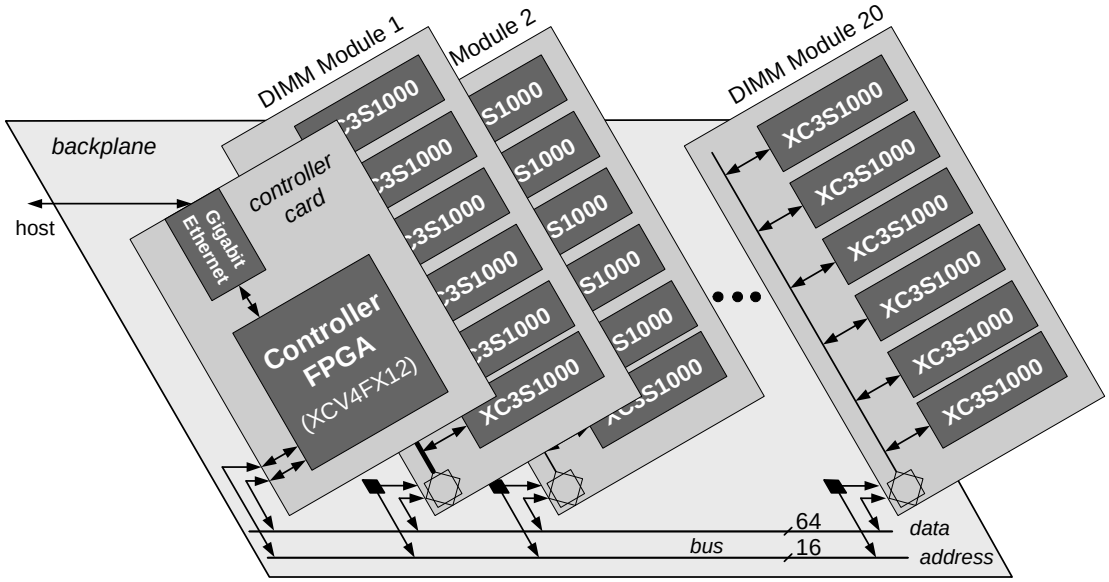


Figure 1: Architecture of COPACOBANA

on the FPGAs. The first prototype of COPACOBANA was equipped with up to 120 FPGAs, distributed along 20 slots - in FPGA modules which can be plugged into a single backplane. Note that the choice for 120 FPGAs was driven by the form factor of the FPGA module which was designed according to cheap and standardized DIMM interface specifications. One (single-sided) DIMM-sized FPGA module can host 6 FPGA devices in a 17×17 mm package, such as the Xilinx Spartan3-1000 FPGA (XC3S1000, speed grade -4, FT256 packaging). This device comes with 1 million system gates, 17280 equivalent logic cells, 1920 Configurable Logic Blocks (CLBs) equivalent to 7680 slices, 120 Kbit Distributed RAM (DRAM), 432 Kbit Block RAM (BRAM), and 4 digital clock managers (DCMs) [40]. The backplane of COPACOBANA connects all FPGA devices with a shared 64-bit data and 16-bit address bus. The entire cluster system is depicted in Figure 1.

COPACOBANA was designed for single master bus arbitration for simplified control. However, in case the communication scheduling of an application is not predictable, the bus master is required to poll all FPGAs for new events and returned data. This significantly slows down the communication performance and increases latencies when reading data back from the FPGAs. Data transfer from and to the FPGAs is accomplished by a dedicated control unit. Originally, we decided to pick a small development board with an FPGA (CESYS USB2FPGA [6]) in favor of a flexible design. The board provides an easy-pluggable 96-pin connector which we use for the connection to the backplane. In later versions of COPACOBANA, we replaced the USB controller using an TCP/IP-based unit so that COPACOBANA can be controlled remotely and can be placed externally, for example in a server room.

3 Cryptanalytic Applications for COPACOBANA

In this section, we briefly describe cryptanalytic applications which we have already implemented on our initial release of the COPACOBANA cluster system. We compiled the most important

facts and key points of each application into individual case studies which eventually should help to identify shortcomings and potential enhancements of our cluster architecture.

3.1 Exhaustive Key Search Scenarios

In the following sections, we present a short survey about our work on exhaustive key search and guessing attacks on a variety of real-world systems. Since all these applications consist mostly out of very basic tasks that can be efficiently parallelized on completely independent computational cores, they can be perfectly mapped onto a highly parallel cluster system such as COPACOBANA.

3.1.1 Case-Study I: Breaking DES with Exhaustive Search

Our first cryptanalytic target application was the exhaustive key search on the DES block cipher. We implemented a known-plaintext attack and used an improved version of the DES engine of the Université Catholique de Louvain's Crypto Group [36] as a core component. Inside a single FPGA, we could place four of such DES engines which allows for sharing plaintext-ciphertext input pairs and the key space. Our first implementation and successful attack was presented in [25]. Since this original publication, we were able to improve the system performance by use of additional pipelined comparators and simplified control logic. Now, we are able to operate each of the FPGAs at an increased clock rate of 136 MHz with a overall gain in performance by 36%, compared to [25]. Consequently, 2^{42} keys can be checked in $2^{40} \times 7.35\,\text{ns}$ by a single FPGA, which is approximately 135 minutes. Since COPACOBANA hosts 120 of these low-cost FPGAs, the entire system can check $4 \times 120 = 480$ keys every 7.35 ns, i.e., 65.28 billion keys per second. To find the correct key, COPACOBANA has to search through an average of 2^{55} different keys. Thus, it can find the right key in approximately $T = 6.4$ days on average. By increasing the number n of COPACOBANAs used for this task, we can further decrease this average runtime of the attack by a linear factor $1/n$.

3.1.2 Case-Study II: Cracking Norton Diskreet

In the 1990s, Norton Diskreet, a part of the well-known Norton Utilities package, was a very popular encryption tool. Diskreet can be used to encrypt single files as well as to create and manage encrypted virtual disks. The tool provides two encryption algorithms one can choose from, a (cryptographically weak) proprietary algorithm and the DES in cipher block chaining (CBC) mode. To encrypt a file or virtual disk, Diskreet asks for a password with a minimal length of 6 and a maximal length of 40 bytes. From this password the 56-bit DES-key is generated. The password-to-key mapping works as follows: First, leading whitespace characters are removed before the password is converted to uppercase characters which are divided into chunks of 8 bytes. Then, all 8-byte blocks are subsequently XORed with each other and the resulting sum is used as DES-key. Obviously, this method of key generation is unfavorable since the password-to-key mapping is not chaotic at all. As explained more thoroughly in [13], we can modify the key generator to our implementation (cf. Section 3.1.1) so that it generates only a limited set of keys according the password distribution based on different assumptions. The performance of the attack with a single COPACOBANA is shown in Table 1.

Key space	**Remark**	**DES decryptions (on average)**	**Runtime**
$\{A, \ldots, Z, @, [, \backslash,], ^{\prime}, \_\}$	Known pwd length	2^{31}	32.8 μs
$\{A, \ldots, Z, @, [, \backslash,], ^{\prime}, \_\}$	Unknown pwd length	2^{35}	0.53 s
7-bit ASCII	8 Characters	2^{47}	35.9 m
8-bit ASCII	8 Characters	2^{55}	6.39 d

Table 1: Breaking Norton Diskreet with COPACOBANA

3.1.3 Case-Study III: Hacking the ePassport

The electronic passport (ePass), as specified by the international civil aviation organization (ICAO), is deployed in many countries all over the world. The security and privacy threats have been widely discussed (e.g., [20], [22], [17], [19]). A chip embedded in the machine readable travel document (MRTD) contains private data as text, such as name, date of birth and sex, as well as biometrics [29]. A digital facial photograph and in some countries additionally fingerprints or an iris scan of the passport holder can be accessed via a contactless interface based on the ISO 14443 [18] standard. The wireless communication constitutes new opportunities for attackers, such as relay attacks [21] or eavesdropping from a range of several meters, as investigated in [14], [35] and [9]. To prevent unauthorized access to the information transferred via the radio frequency (RF) interface some countries, among them Germany and the Netherlands, employ the so-called basic access control (BAC). The BAC is meant to secure the interchanged data, i.e., establish a confidential channel, by employing symmetric cryptography. The secret keys needed for carrying out the BAC are stored in the embedded integrated circuit (IC) and can also be derived from a machine readable zone (MRZ) that is printed on the paper document.

We here shortly sketch a possible attack on the BAC using COPACOBANA which is adapted from [4] and more thoroughly described in [27, 13]. We assume that a device for eavesdropping of the RF field can be mounted nearby an e-passport inspection system, such that all bits transmitted via the air channel can be captured and stored in a database. The fundamental secret required to access the private data rely on an authentication key k_{MAC} and an encryption key k_{ENC} that are derived from the MRZ information on the paper document according to

$$k_i = msb_{16}\left(\text{SHA-1}\left(msb_{16}\left(\text{SHA-1}\left(\text{MRZ}\right)\right) \| C\right)\right),$$

where the $msb_{16}(x)$ function selects the most significant 16 byte of x. After the first execution of SHA-1[32], the result is concatenated with a constant C which is either $C = \texttt{0x00000001}$ for k_{ENC} or $C = \texttt{0x00000002}$ for k_{MAC}. The keys are then used with a Triple DES (TDES) block-cipher [30]. Having access to system messages by eavesdropping the near-field communications, we can attack the combination of SHA-1 and TDES in a brute-force attack scenario (although the theoretical keyspace available to TDES is out of reach for conventional exhaustive key search attacks). This is possible since the entropy of the MRZ can be found to be as low as $\approx 2^{33}$ for realistic scenarios based on the BAC realizations of the Netherlands and Germany [35, 27]. Hence, instead of performing an exhaustive search on every possible TDES key, we implemented again a smart generator which only produces a limited set of outputs that are reasonable MRZ values.

The time critical component in our attack implementation is the SHA-1 hash function, determining the maximum clock frequency of $f_{clk} = 40\,\text{MHz}$ and requiring 80 clock cycles for

one key candidate. Processing of one key thus requires $80 \times 25\,\text{ns} = 2\,\mu\text{s}$. As there are 120 FPGAs running in parallel, each consisting of four encryption engines, $4 \times 120 = 480$ keys are tested every $2\,\mu$s, resulting in a throughput of $2^{27.84} \approx 240$ million keys per second. On average, testing of 2^{33} keys reveals the correct candidate in $\frac{2^{32}}{2^{27.84}} \approx 18$ seconds which can be regarded as real-time, compared to the duration of one inspection at the border control.

3.1.4 Case-Study IV: Breaking the A5/1 Streamcipher

A5/1 is a synchronous stream cipher that is used for protecting GSM communication. In the GSM protocol, the communication channel is organized in 114-bit frames that are encrypted by XORing them with 114-bit blocks of the keystream produced by the cipher as follows: A5/1 is based on three LFSRs, that are irregularly clocked. The three registers are 23, 22 and 19 bits long, representing the internal 64-bit state of the cipher. During initialization, a 64-bit key k is clocked in, followed by a 22-bit initialization vector that is derived from the publicly known frame number. After that a warm-up phase is performed where the cipher is clocked 100 times and the output is discarded. For a detailed description of A5/1 please refer to [3].

Most of previously proposed attacks against A5/1 lack from practicability and/or have never been fully implemented. In contrast to these attacks, we present in [11] a real-world attack revealing the internal state of A5/1 in about 6 hours on average (and about 12 hours in the worst-case) using COPACOBANA. The implementation is an optimization of a *guess-and-determine* attack as proposed in [23], including an improvement in runtime of about 13% compared to their original approach. Each FPGA contains 23 guessing engines running in parallel at a clock frequencey of 104 MHz each. To mount the attack, only 64 consecutive bits of a known keystream are required and we do not need any precomputed data. Note, however, that an average of 6 hours runtime still cannot be considered a real-time attack when using a single COPACOBANA. In this case, we need to record the full communication first and attack its encryption offline afterwards. Alternatively, by adding further machines the attack time will be linearly reduced, e.g., 100 machines only require 3.6 minutes for a sucessfull attack on average.

3.2 Advanced Cryptanalytic Applications

In the last section, we briefly surveyed simple attacks based on exhaustive key searches or guessing. All these attacks have in common that their performance is basically limited by the number of computations. In other words, the available logic of the FPGA devices on COPACOBANA directly determines the performance of the attack. By incrementing the number of COPACOBANA units we yield a speed-up in performance by a perfect linear factor. This, however, does not hold for the following, more advanced attacks.

3.2.1 Case-Study V: Time-Memory (Data) Tradeoffs

Time-Memory Tradeoff (TMTO) and Time-Memory-Data Tradeoff (TMDTO) methods were designed as a compromise between the two well-known extreme approaches: either to perform an exhaustive search on the entire key space of the cipher or precomputing exhaustive tables representing all possible combinations of keys and ciphertexts for a given plaintext. The TMTO and TMDTO strategies offer a way to reasonably reduce the actual search complexity (by doing some kind of precomputation) while keeping the amount of precomputed data reasonably low, whereas "reasonably" has to be defined more precisely. Roughly speaking, it depends on the concrete attack scenario (e.g., real-time attack), the internal step function and the available resources for the precomputation and online (search) phase.

Method	DU [GB]	PT (COPA) [days]	OT [ops]	TA	SR
Hellman	1897	24	$2^{40.2}$	$2^{40.2}$	0.80
Rivest	1690	95	2^{21}	$2^{39.7}$	0.80
Oechslin	1820	23	$2^{21.8}$	$2^{40.3}$	0.80

Table 2: TMTO methods according to: expected runtimes and memory requirements using COPACOBANA for precomputations.

Existing TMTO methods by Hellman, Rivest and Oechslin [16, 7, 31] share the natural property that in order to achieve a significant success rate much precomputation effort is required on chained computations. A representation of start point and end point of each chain is stored in (a set of) large tables, e.g., on hard disk drives. The actual attack takes place in a second search phase (online phase) in which another chain computation is performed on the actual data and compared to the stored endpoints in the tables. In case a matching endpoint is found in the table, the sequence of keys can be reconstructed using the corresponding start point. There are few contributions attacking DES with the TMTO approach. In [38] an FPGA design for an attack on a 40-bit DES variant using Rivest's TMTO method [7] was proposed. In [28] a hardware architecture for UNIX password cracking based on Oechslin's method [31] was presented. However, to the best of our knowledge, a set of complete TMTO precomputation tables for *full* 56-bit DES was never created up to now.

The idea of cryptanalytic TMDTO is due to Babbage, Biryukov and Shamir [1, 2]. TMDTOs are variants of TMTOs exploiting a scenario where multiple data points $y_1, \ldots, y_D$ of the function g are given and one has just to be successful in finding a preimage of one of them. Such a scenario typically arises in the cryptanalysis of stream ciphers where we like to invert the function mapping the internal state (consisting of $\log_2(N)$ bits) to the first $\log_2(N)$ output bits of the cipher produced from this state. We employed this method to mount an advanced attack on the A5/1 streamcipher which provides a runtime of far less than 6 hours (cf. Section 3.1.4), at the cost of a reduced success probability.

In [13] we present possible configurations and parameters to use COPACOBANA for TMTO/TMDTO precomputations both to attack the DES blockcipher and the A5/1 streamcipher. Our estimates took the assumed communication bandwidth between host-PC and backplane of 24 MBit/s into account. To break DES with TMTOs on COPACOBANA, Table 2 presents our worst case expectations concerning success rate (SR), disk usage (DU), the duration of the precomputation phase (PT) as well as the number of table accesses (TA) and calculations (C) during the online phase (OT). Note that for this extrapolation, we have used again the implementation of our exhaustive key search on DES (cf. Section 3.1.1).

For the implementation of the TMDTO attack on A5/1, we selected the set of parameters presented in the second row of Table 3, since it produces a reasonable precomputation time and a reasonable size of the tables as well as a relatively small number of table accesses. The success rate of 63% may seem to be small, but it increases significantly if more data samples are available: For instance, if 4 frames of known keystream are available, then $D = 4 \cdot 51 = 204$ and thus the success rate is increased to 96%.

Although both attacks are realistic (precomputations of less than one and about 3 months for DES and A5/1 respectively), we encountered several issues while running the attacks. One

m	S	d	I_ℓ	PT [days]	DU [TB]	OT [secs]	TA	SR [%]
2^{41}	2^{15}	5	$[2^3,2^6]$	337.5	7.49	27.8	2^{21}	0.86
2^{40}	2^{14}	5	$[2^4,2^7]$	95.4	4.85	10.9	2^{20}	0.63
2^{39}	2^{15}	5	$[2^3,2^6]$	84.4	3.48	27.8	2^{21}	0.60
2^{37}	2^{15}	6	$[2^4,2^8]$	47.7	0.79	73.5	2^{21}	0.42

Table 3: A5/1 TMDTO: Expected runtimes and memory requirements. The choice and explanation for parameters are described thoroughly in [13]

problem is the access time to hard disk storage in the online phase which is not reflected in the tables above and takes a considerable amount of time for itself (about 4-10 ms per access). Moreover, the generation of precomputation tables is slower than expected with respect to the communication link between host-PC and COPACOBANA backplane. It turned out that the communication speed of 24 MBit/s must be considered as theoretical throughput limit since additional data overhead and latencies need to be taken into account as well. In other words, to support TMTO/TMDTO for required parameters, the communication facilities of COPACOBANA need to be improved by at least one order of magnitude.

3.2.2 Case-Study VI: Integer Factorization

The factorization of a large composite integer n is a well-known mathematical problem which has attracted special attention since the invention of public key cryptography. RSA is known as the most popular asymmetric cryptosystem and was originally developed by Ronald Rivest, Adi Shamir and Leonard Adleman in 1977 [34]. Since the security of RSA relies on the attacker's inability to factor large numbers, the development of a fast factorization method could allow for cryptanalysis of RSA messages and signatures. Recently, the best known method for factoring large RSA integers is the General Number-Field Sieve (GNFS). An important step in the GNFS algorithm is the factorization of mid-sized numbers for smoothness testing. For this purpose, the Elliptic Curve Method (ECM) has been proposed by Lenstra [26] which has been proved to be suitable for parallel hardware architectures in [37, 10, 8], particularly on FPGAs.

The ECM algorithm performs a very high number of operations on a very small set of input data, and is not demanding in terms of high communication bandwidth. Furthermore, the implementation of the first stage of ECM requires only little memory since it is based on point multiplication on an elliptic curve. The operands required for supporting GNFS are well beyond the width of current computer buses, arithmetic units, and registers, so that special purpose hardware can provide a much better solution.

In [8] it has been shown that the utilization of DSP slices in Virtex-4 FPGAs for implementing a Montgomery multiplication can significantly improve the ECM performance. In that work, the authors used a fully parallel multiplier implementation which provides the best known performance figures for ECM phase 1 so far, however they did provide details how to realize ECM phase 2.

To accelerate integer arithmetic using a similar strategy, we designed a new slot-in module for use with a second release of COPACOBANA hosting 8 Xilinx Virtex-4 XC4VSX35 FPGAs, each providing 192 DSP slices. Due to the larger physical package of the FPGAs (FF668 package with dimension of 27x27 mm) we enlarged the modules. This included also modifications of the

Aspect	**Our work**	**Results** [10]
Modular Adder/Subtracter	14 clk	31 clk[1]
Montgomery Multiplication	118 clk	167 clk[1]
Point Doubling	434 clk	n/a
Point Addition	500 clk	n/a
Combined Point Doubling and Addition	689 clk	947 clk[1]
Clock Frequency of an ECM Core	200 MHz	135 MHz

Table 4: Clock cycles and frequency for point multiplication of 151-bit numbers required in phase 1 of ECM on Virtex-4 devices (single core)

corresponding connectors on the backplane. For more efficient heat dissipation at high clock frequencies up to 400 MHz, an actively ventilated heat sink is attached to each FPGA. With a more powerful power supply providing 1.5 kW at 12 V, we could run a total of 128 Virtex 4 SX35 FPGAs distributed over 16 plug-in modules.

In contrast to [8], we used a multi-core ECM design per FPGA. A single ECM engine comprises of an arithmetic unit computing modular multiplication and additions, a point multiplication unit for phase 1 and ROM tables for phase 2. Only point operations on the elliptic curve are performed on FPGAs, this means that the setup of the Montgomery curve needs to be done on the host-PC and then transferred to the FPGAs. We provide first estimates for ECM phase 1 shown in Table 4 and compare our results to the implementation presented in [10].

Although the implementation based on DSP slices promise better results on Virtex-4 FPGAs compared to [10], we like to point out that the switch to Virtex-4 SX35 devices has a strong negative effect on our cost-performance ratio. With respect to a Spartan-3 device, a single Virtex-4 SX35 is much more expensive (roughly a factor of 10-20) and does not outperform the cheaper Spartan-3 devices by an corresponding factor. Hence, we still consider Spartan-3 devices more appropriate for cryptanalytical tasks due to their better cost-performance ratio. In particular, latest Spartan-3A DSP and Spartan-6 devices also come with a number of DSP slices, so that expensive Virtex devices (which formerly were the only devices with DSP slices) are not a necessity anymore.

Another issue of ECM is memory. Although ECM stage 1 has only very moderate memory constraints it can be considerately improved by additional computations in a second stage. However, stage 2 involves a significant amount of precomputations as well as storage for prime numbers. Since memory on FPGA devices is rather limited (192 × 18 KBit BRAM elements per device), a fast accessible, external memory could help to improve the beneficial effect of stage 2 by storing even larger tables.

3.2.3 Case-Study VII: Solving Elliptic Curve Discrete Logarithms

Another popular problem used for building public-key cryptosystems is known as the Discrete Logarithm Problem (DLP) where the exponent ℓ should be determined for a given $a^\ell \mod n$. A popular derivative is the Elliptic Curve Discrete Logarithm Problem (ECDLP) for Elliptic Curve Cryptosystems (ECC) [15].

[1]The presented cycle count for 151-bit integers was scaled down accordingly to the results given in [10] for 198-bit parameters. Here, Gaj *et al.* reported their implementation to take 1212 cycles for one combined 198 bit point doubling and addition (ECM stage 1).

An attack on ECC relies on the same algorithmic primitives as the crypto system itself, namely point addition and point doubling. Up to now, the best known algorithm for this purpose is the Pollard's Rho (PR) algorithm for parallel implementation described in [39]. This variant of the original PR method [33] allows for a linear gain in performance with the number of available processors. This can be efficiently implemented in hardware as presented in [12].

The PR algorithm essentially determines distinguished points on the elliptic curve. These points are reported to a central host computer which awaits a collision of two points. A distinguished point is defined to be a point with a specific characteristic, e.g., its x-coordinate has a fixed number of leading zero bits. To reach such a distinguished point, PR follows a so called pseudo-random walk on the elliptic curve by subsequently adding points from a fixed, finite set of random points. Hence, with careful parametrization of the distinguished point criterion, the duration of a computation until a distinguished point is found can be adapted to the bandwidth constraints of the system. Furthermore, the PR does not need a large memory for computation so that the COPACOBANA system seems to be a suitable platform for running the algorithm. As with the ECM unit, a single PR unit is comprised of an arithmetic unit, a few kilobytes of RAM and control logic. The arithmetic unit supports modular inversion as an additional function required for uniquely determining distinguished points.

For a parallelized PR on COPACOBANA according to the method presented in [39], all instances of the algorithm can run completely independent from each other. For solving the discrete logarithm problem over curves defined over prime fields $\mathbb{F}_p$, we have to compute approximately $\sqrt{q}$ points, where q is the largest prime power of the order of the curve. Note that the transfer of data between host computer and point processing units on the FPGA can be performed independently from the computations.

Implementing the PR on Spartan-3 FPGAs for solving the ECDLP over curves with a length of 160 bits and using an affine point representation, we achieve a maximum clock frequency of approximately 40 MHz and an area usage of 6067 slices (79%) for two parallel instances. The corresponding point addition requires 846 cycles so that slightly less than 50,000 point operations can be performed per second by one unit. Consequently, a single COPACOBANA can compute about 11.3 million point operations per second. Table 5 compares our results for COPACOBANA with challenges and corresponding estimates from Certicom based on the computing time of an (outdated) Intel Pentium 100. To compare our results with COPACOBANA against more recent systems, we refer to the solved ECC P-109 challenge that took 10,000 computers (mostly PCs) running 24 hours a day for a total time of 549 days. To solve this challenge in the same time, according to Table 5 about 17 COPACOBANAs (and subsequently an investment of € 170,000) would be required. In return, assuming a single PC of the original cluster to cost only about € 200, this already sums up to the amount of € 2 million, excluding any additional operational costs for power and cooling.
Note that our Pollard-Rho implementation also can benefit from advanced FPGAs with DSP slices (cf. Section 3.2.2. The large n-bit adders can be more efficiently implemented using cascades of the 48-bit adder contained in each DSP slice.

4 Enhancing the COPACOBANA Architecture

Recapitulating the issues of our COPACOBANA architecture according to the application shown in Section 3, we should consider a redesign to meet the following requirements:

- *Larger FPGA Devices:* more logical elements on an FPGA enable more complex applications or more computational cores per device.

k	**Certicom Est.** [5]	**Single XC3S1000**	**Single COPACOBANA**
79	146 d	15.3 d	3.06 h
89	12.0 y	1.62 y	4.93 d
97	197 y	30.7 y	93.4 d
109	$2.47 \cdot 10^4$ y	$2.91 \cdot 10^3$ y	24.3 y
131	$6.30 \cdot 10^7$ y	$7.40 \cdot 10^6$ y	$6.17 \cdot 10^4$ y
163	$6.30 \cdot 10^{12}$ y	$9.15 \cdot 10^{11}$ y	$7.62 \cdot 10^9$ y
191	$1.32 \cdot 10^{17}$ y	$1.89 \cdot 10^{16}$ y	$1.57 \cdot 10^{14}$ y
239	$3.84 \cdot 10^{24}$ y	$8.62 \cdot 10^{23}$ y	$7.18 \cdot 10^{21}$ y

Table 5: Expected runtime on different platforms and for different Certicom ECC challenges

- *Local Memory per FPGA:* a few megabytes of fast SRAM should be placed adjacent to each FPGA device to provide additional storage while solving more complex problems.
- *Dedicated Communication Links*: Only a single access to an FPGA at a time was possible in the recent communication models. Individual links for each FPGA simplify data communication and also enable high performance from simultaneous data exchange.
- *Improved Controller between Host-PC and COPACOBANA*: the main bottleneck with respect to data communication is the controller interface between backplane and host-PC. The embedded controllers for USB and Gigabit Ethernet, which we used for previous controller interfaces, did not provide sufficient performance due to high overhead (e.g., TCP/IP and USB frame packaging). A solution could be to integrate the host-PC inside the COPACOBANA case and directly link the PC's mainboard with the backplane, e.g., using a high-speed PCIe link.
- *Improved Signaling*: the single-ended I/O lines for the data and address bus system were subject of significant noise and side-effects like signal crosstalk. Improved signal quality can be achieved by switching to differential I/O at the cost of another signal line per I/O port. This is also beneficial when attempting to increase the data transmission frequency.

To address the aspects mentioned above, we entirely redesigned the backplane and FPGA module of our cluster system. The new FPGA module consists of 8 FPGAs with a package size up to $27 \times 27mm$, a CPLD for system control (e.g., temperature and voltage monitoring) and DC/DC converters (dependant on the FPGA type used). Most suitable FPGA devices for the new systems are the low-cost Xilinx Spartan-3 5000 with up to 74,880 logic cells (104 hardware multipliers, 104 BRAMs, FG676 packaging), the Xilinx Spartan-3A DSP 3400 with 53,712 logic cells (126 DSP48A, 126 BRAMs, FG676 packaging) and the upcoming class of Spartan-6 FPGAs. For the first prototype of the enhanced COPACOBANA, we chose 128 Spartan-3 5000 devices. Moreover, we placed 32 MB of SRAM adjacent to each FPGA that can be accessed by the FGPA within a single clock cycle at 100 MHz.

The new COPACOBANA design integrates the host-PC (uATX format) in the same case so that short (and fast) interfaces can be used. Target applications like for example Time-Memory Tradeoffs (cf. Section 3.2.1) require a high data rate between COPACOBANA and an external hard disk drive. One option could be to place a hard disk drive physically inside COPACOBANA, e.g., attached via a SATA interface to the integrated host-PC. The second option would let the integrated host-PC access other IT-infrastructure like a Storage-Area Network by additional interfaces or its two Gigabit Ethernet ports.

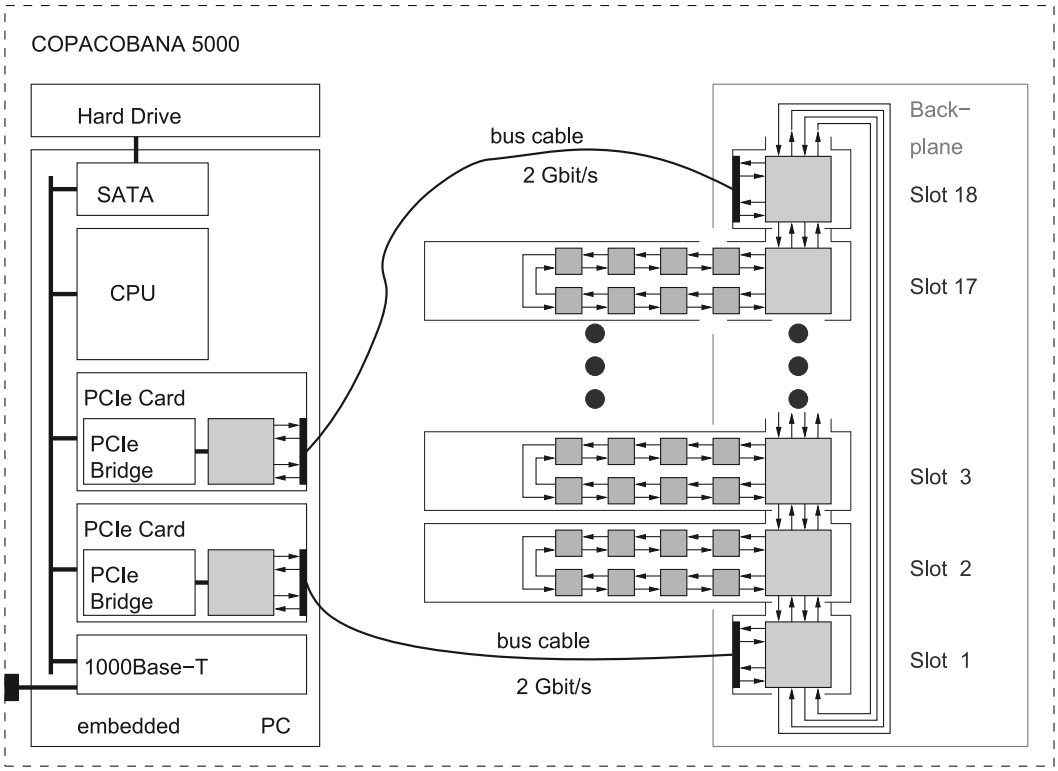


Figure 2: Enhanced COPACOBANA Architecture based on Xilinx Spartan-3 5000 FPGAs

The integrated PC connects to the FPGA-cluster by one or two PCIe modules. Therefore, the former communication bottleneck due to the single controller is completely eliminated. The new enhanced architecture COPACOBANA consists of an 18 slot backplane equipped with 16 FPGA-cards and two controller cards which connect the FPGA backplane to the integrated PC. Additional components of the new system are the $1.5kW$ main power supply unit (with 125A at 12V), six high-performance fans and a 19-inch rack of three hight units for the housing.

There are fast serial point-to-point connections between every two neighbors building a chain of FPGAs. In the configuration as described below, we employed eight Xilinx Spartan-3 5000 which are arranged as a systolic one-dimensional array, i.e., in each clock cycle data is transferred from one FPGA to the next in pipeline fashion according to a global, synchronous clock. Note that transferring data using a systolic array introduces significant latencies on the data path. However, since the target applications do not have real-time requirements, this should not be an issue for our cryptanalytic applications where operations usally can be interleaved to hide any latencies. Between the controller-cards and the integrated PC the maximum data rate is limited to 250 MByte/s due to the limitations of the PCIe connection. For high throughput, the I/O capabilities of the FPGA has to be considered carefully. The highest throughput can be achieved by connecting communicating chips by short point-to-point lines. To allow efficient broadcasts to all FPGAs simultaneously, there is a direct connection between adjacent FPGA-cards. The point-to-point interconnections consist of 8 pairs of wires in each direction. Each pair is driven by low voltage differential signaling (LVDS) with a speed of 250MHz, thus achieving a data-rate of 2Gbit/s. Figure 2 shows the overall architecture of our enhanced COPACOBANA.

As in the original system, the backplane connects to all FPGA cards on which individually the clock signals, data and power are (re-)generated and distributed. However, the bus is

managed now cooperatively by all FPGA modules instead of a single bus master, i.e., a central controller. Each FPGA module can transfer incoming data to the next slot or it can remove the data out of the stream. In this case, an empty data frame cycles through the bus pipeline from slot to slot. Note that another card is allowed to insert new data now into this empty slot. The two counter-rotating systolic datapaths allow to minimize the worst case latency to half of the total number of slots multiplied by the clock cycle time. The enhanced bus system assigns one ascending and one descending slot to each single card. This leads to a ring of point to point connections in which the bus system can be seen as a circular, parallel shift register.

Due to the modular architecture further developments can be incorporated seamlessly. For example, alternative FPGA modules equipped with a Spartan-3A DSP 3400 or Spartan-6 FPGAs can easily plugged into the backplane. In particular, it is possible to run even a heterogenous configuration, with a mixed set of FPGAs for different tasks.

5 Expected Improvements in Cryptanalytical Applications

At the time of writing, the cluster incorporating the presented enhancements as described in Section 4 is still in production and is expected to become available in October 2009. Hence, we now provide first estimates and projections concerning the expected performance of the new cluster system. We will revise our figures as soon as the system (and adapted cryptanalytical implementations) become available. With the design modifications on the COPACOBANA architecture, we can firstly make use of more logic resources due to the larger Spartan-3 5000 FPGAs. With respect to the original Spartan-3 1000 FPGAs, the amount of logic has increased by a factor of 4.5 per FPGA device. For our exhaustive key search applications as shown in Section 3.1, we thus assume a linear speedup (at least) by a factor of 4. More precisely, the original DES breaking application implemented four engines per Spartan-3 1000 so that we expect 16 engines to run at the same clock frequency per Spartan-3 5000. This will reduce the average runtime to break DES to a single day and 19 hours with only one enhanced COPACOBANA. Similar linear performance speed-ups can be gained for the ePass cracker and A5/1 breaker (about 4.5 seconds and 1.5 hours, respectively). However, note that due to the higher cost of the enhanced COPACOBANA (which grows by a factor of 4.5 as well), the cost-performance is not better than with the original machine (even worse, when taking Moore's Law into account). In general, brute-force techniques do not benefit from the new architectural improvements, i.e., instead of one new COPACOBANA also five original machines based on Spartan-3 1000 can be used. In this case, the only advantage of the new design is due to the reduced power consumption.

The real advantages of the enhanced machine manifest for advanced cryptanalytical application with higher demands on the infrastructures such as communication throughput and local memories adjacent to the FPGAs. The efficient generation of TMTO tables will now become available so that we expect to finish the tables for A5/1 in less than a month. Furthermore, we are working towards an implementation of our ECM core (cf. Section 3.2.2) for Spartan-3A DSP 3400 FPGAs which also integrate 126 DSP slices, however at much lower costs compared to Virtex-4 devices. Finally, we plan to adapt the same implementation strategy based on DSP slices also for the Pollard-Rho ALU. This can also result in a gain in performance by an order of magnitude. Last but not least, the new COPACOBANA also could provide a suitable platform for even more complex applications, e.g., additional tasks required by the number field sieve, index calculus methods or lattice basis reduction algorithms.

6 Conclusion

In this work, we presented a series of cryptanalytical applications for a cost-efficient hardware architecture (COPACOBANA). According to our findings based on a varietly of cryptanalytical implementations, we identified shortcomings in the design of our cluster. Then, we came up with an enhanced version which also provides larger and more powerful FPGAs, up to 4 GB of local memory and fast point-to-point serial communication links between all devices and the controller. These modifications bring COPACOBANA into a promising position to tackle even more complex tasks in cryptanalsis as well as from general-purpose computing. With all these enhancements, in particular for communication and local memory, the new COPACOBANA cluster becomes even useful beyond pure code-breaking, catching up with respect to other supercomputing platforms, still at low costs.

References

[1] S. Babbage. A Space/Time Tradeoff in Exhaustive Search Attacks on Stream Ciphers. In *European Convention on Security and Detection*, volume 408 of *IEE Conference Publication*, 1995.

[2] A. Biryukov and A. Shamir. Cryptanalytic Time/Memory/Data Tradeoffs for Stream Ciphers. In *Proc. of ASIACRYPT'00*, volume 1976 of *LNCS*, pages 1–13. Springer, 2000.

[3] A. Biryukov, A. Shamir, and D. Wagner. Real Time Cryptanalysis of A5/1 on a PC. In *Proc. of FSE'00*, volume 1978 of *LNCS*, pages 1–18. Springer, 2001.

[4] D. Carluccio, K. Lemke-Rust, C. Paar, and A.-R. Sadeghi. E-Passport: The Global Traceability or How to Feel Like an UPS Package. In *Proc. of WISA'06*, volume 4298 of *LNCS*, pages 391–404. Springer.

[5] Certicom Corporation. Certicom ECC Challenges, 2005. `http://www.certicom.com`.

[6] CESYS GmbH. USB2FPGA Product Overview. `http://www.cesys.com`, January 2005.

[7] D. Denning. Cryptography and Data Security. Addison-Wesley, 1982.

[8] G. de Meulenaer, F. Gosset, M. M. de Dormale, and J.-J. Quisqater. Integer Factorization Based on Elliptic Curve Method: Towards Better Exploitation of Reconfigurable Hardware. In *Proc. of FCCM'07*, pages 197–206. IEEE Computer Society, 2007.

[9] T. Finke and H. Kelter. Radio Frequency Identification – Abhörmöglichkeiten der Kommunikation zwischen Lesegerät und Transponder am Beispiel eines ISO14443-Systems. `http://www.bsi.de/fachthem/rfid/Abh_RFID.pdf`.

[10] K. Gaj, S. Kwon, P Baier, P. Kohlbrenner, H. Le, M. Khaleeluddin, and R. Bachimanchi. Implementing the Elliptic Curve Method of Factoring in Reconfigurable Hardware. In *Proc. of CHES '06*, volume 4249 of *LNCS*, pages 119–133. Springer, 2006.

[11] Timo Gendrullis, Martin Novotný, and Andy Rupp. A real-world attack breaking A5/1 within hours. In Elisabeth Oswald and Pankaj Rohatgi, editors, *Proc. of CHES '08*, volume 5154 of *Lecture Notes in Computer Science*, pages 266–282. Springer, 2008.

[12] T. Güneysu, C. Paar, and J. Pelzl. Attacking Elliptic Curve Cryptosystems with Special-Purpose Hardware. In *Proc. of FPGA'07*, pages 207–215. ACM Press, 2007.

[13] Tim Güneysu, Timo Kasper, Martin Novotný, Christof Paar, and Andy Rupp. Cryptanalysis with COPACOBANA. *IEEE Trans. Comput.*, 57(11):1498–1513, November 2008.

[14] G. P. Hancke. Practical Attacks on Proximity Identification Systems (Short Paper). In *Proc. of SP'06*, pages 328–333. IEEE Computer Society, 2006.

[15] D. R. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Verlag, 2004.

[16] M. E. Hellman. A Cryptanalytic Time-Memory Trade-Off. In *IEEE Transactions on Information Theory*, volume 26, pages 401–406, 1980.

[17] J.-H Hoepman, E. Hubbers, B. Jacobs, M. Oostdijk, and R. Wichers Schreur. Crossing Borders: Security and Privacy Issues of the European e-Passport. In *Proc. of IWSEC'06*, volume 4266 of *LNCS*, pages 152–167. Springer, 2006.

[18] ISO/IEC 14443. Identification Cards - Contactless Integrated Circuit(s) Cards - Proximity Cards - Part 1-4. www.iso.ch, 2001.

[19] S. Vaudenay J. Monnerat and M. Vuagnoux. About Machine-Readable Travel Documents. In *Proc. of RFIDSec'07*, pages 15–28, 2007.

[20] A. Juels, D. Molnar, and D. Wagner. Security and Privacy Issues in E-Passports. In *Proc. of SecureComm'05*, pages 74–88. IEEE Computer Society, 2005.

[21] T. Kasper, D. Carluccio, and C. Paar. An Embedded System for Practical Security Analysis of Contactless Smartcards. In *Proc. of WISTP'07*, volume 4462 of *LNCS*, pages 150–160. Springer, 2007.

[22] G.S. Kc and P.A. Karger. Security and Privacy Issues in Machine Readable Travel Documents (MRTDs). RC 23575, IBM T. J. Watson Research Labs, April 2005.

[23] J. Keller and B. Seitz. A Hardware-Based Attack on the A5/1 Stream Cipher. Technical report, Fernuni Hagen, Germany, 2001. `http://pv.fernuni-hagen.de/docs/apc2001-final.pdf`.

[24] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, A. Rupp, and M. Schimmler. How to Break DES for € 8,980. In *SHARCS Workshop*. Cologne, Germany, 2006.

[25] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler. Breaking Ciphers with COPACOBANA - A Cost-Optimized Parallel Code Breaker. In *Proc. of CHES '06*, volume 4249 of *LNCS*, pages 101–118. Springer, 2006.

[26] H. Lenstra. Factoring Integers with Elliptic Curves. *Annals Math.*, 126:649–673, 1987.

[27] Y. Liu, T. Kasper, K. Lemke-Rust, and C. Paar. E-Passport: Cracking Basic Access Control Keys. In *Proc. of OTM'07, Part II*, volume 4804 of *LNCS*, pages 1531–1547. Springer, 2007.

[28] N. Mentens, L. Batina, B. Prenel, and I. Verbauwhede. Time-Memory Trade-Off Attack on FPGA Platforms: UNIX Password Cracking. In *Proc. of ARC'06*, volume 3985 of *LNCS*, pages 323–334. Springer, 2006.

[29] ICAO TAG MRTD/NTWG. Biometrics Deployment of Machine Readable Travel Documents, Technical Report, 2004.

[30] NIST FIPS PUB 46-3. *Data Encryption Standard*. Federal Information Processing Standards, National Bureau of Standards, U.S. Department of Commerce, January 1977.

[31] P. Oechslin. Making a Faster Cryptanalytic Time-Memory Trade-Off. In *Proc. of CRYPTO'03*, volume 2729 of *LNCS*, pages 617–630. Springer, 2003.

[32] National Institute of Standards and Technology. FIPS 180-3 Secure Hash Standard (Draft). `http://www.csrc.nist.gov/publications/PubsFIPS.html`.

[33] J. M. Pollard. Monte Carlo Methods for Index Computation mod p. *Mathematics of Computation*, 32(143):918–924, July 1978.

[34] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[35] Harko Robroch. ePassport Privacy Attack, Presentation at Cards Asia Singapore, April 26,2006. `http://www.riscure.com`.

[36] G. Rouvroy, F.-X. Standaert, J.-J. Quisquater, and J.-D. Legat. Design Strategies and Modified Descriptions to Optimize Cipher FPGA Implementations: Fast and Compact Results for DES and Triple-DES. In *Proc. of FPGA'03*, pages 247–247. ACM, 2003.

[37] M. Šimka, J. Pelzl, T. Kleinjung, J. Franke, C. Priplata, C. Stahlke, M. Drutarovský, V. Fischer, and C. Paar. Hardware Factorization Based on Elliptic Curve Method. In *Proc. of FCCM'05*, pages 107–116. IEEE Computer Society, 2005.

[38] F. Standaert, G. Rouvroy, J. Quisquater, and J. Legat. A Time-Memory Tradeoff using Distinguished Points: New Analysis & FPGA Results. In *Proc. of CHES '02*, volume 2523 of *LNCS*, pages 596–611. Springer, 2002.

[39] P.C. van Oorschot and M.J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *Journal of Cryptology*, 12(1):1–28, 1999.

[40] Xilinx. Spartan-3 FPGA Family: Complete Data Sheet, DS099. `http://www.xilinx.com`, January 2005.

Sparse Boolean equations and circuit lattices

Igor Semaev*

Department of Informatics, University of Bergen, Norway
igor@ii.uib.no

Abstract. A system of Boolean equations is called sparse if each equation depends on a small number of variables. Finding efficiently solutions to the system is an underlying hard problem in the cryptanalysis of modern ciphers. In this paper we study new properties of the Agreeing Algorithm, which was earlier designed to solve such equations. Then we show that mathematical description of the Algorithm is translated straight into the language of electric wires and switches. Applications to the DES and the Triple DES are discussed. The new approach, at least theoretically, allows a faster key-rejecting in brute-force than with Copacobana.

Key words: Sparse Boolean equations, equations graph, electrical circuits, switches

1 Introduction

Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of Boolean variables. By X_i, $1 \le i \le m$ we denote subsets of X of size $l_i \le l$. The system of equations

$$f_1(X_1) = 0, \ldots, f_m(X_m) = 0 \tag{1}$$

is considered, where f_i are Boolean functions (polynomials in algebraic normal form) and they only depend on variables X_i. Such equations are called l-sparse. We look for the set of all $0,1$-solutions to (1). Obviously, the equation $f_i(X_i) = 0$ is determined by the pair $E_i = (X_i, V_i)$, where V_i is the set of $0,1$-vectors in variables X_i, also called X_i-vectors, where f_i is zero. In other words, V_i is the set of all solutions to $f_i = 0$. The function f_i is uniquely defined by V_i. Given f_i, the set V_i is computed with 2^{l_i} trials.

In [15] Agreeing and Gluing procedures were described. Then they were combined with variables guessing to solve (1). See also earlier work [22]. Table 1 summarizes expected complexity estimates for simple combinations of the Agreeing and Gluing in case of $m = n$ and a variety of l. Each instance of (1) may be encoded by a CNF formula with clause length l in the same variables. So l-SAT solving algorithms provide with worst case complexity estimates. The table data

* The author was partially supported by the grant NIL-I-004 from Iceland, Lichtenstein and Norway through the EEA Financial Mechanism and the Norwegian Financial Mechanism.

Table 1. Algorithms' running time.

l	3	4	5	6
the worst case,[12]	1.324^n	1.474^n	1.569^n	1.637^n
Gluing1, expectation,[18]	1.262^n	1.355^n	1.425^n	1.479^n
Gluing2, expectation,[18]	1.238^n	1.326^n	1.393^n	1.446^n
Agreeing-Gluing1, expectation,[19]	1.113^n	1.205^n	1.276^n	1.334^n

.

suggests that Agreeing-Gluing based methods should be very fast in practice. This is the reason why a hardware implementation of the Agreeing Algorithm is here proposed. In spite of relatively high worst case bound on l-SAT problem complexity, there exist a number of efficient l-SAT solvers. They became useful tool in cryptanalysis [4, 5]. However, an efficient hardware version of the approach is still unknown.

Conjectured asymptotic bounds on the complexity of the popular Gröbner Basis Algorithm and its variants as XL, see [8, 3], are found in [1, 20]. They are far worse than the estimates by the brute force approach except for quadratic and very over-defined equation system. It was found in [16] that a linear algebra variant(called MRHS) of the Agreeing-Gluing significantly overcomes(on AES type Boolean equations in around 50 variables) F4 method, a Gröbner Basis Algorithm implemented in Magma.

We first study here a new property of the Agreeing Algorithm. This algorithm implements pairwise simplification to the initial equations after some suitable guess. We will show that the result only depends on a smaller subset of equation pairs. This significantly reduces memory requirements for the Agreeing Algorithm. E.g. for the DES instead of 3545 pairs, the algorithm should only run through 1404 of them with the same output. In case of the Triple DES the figure is 3929 instead of 16831, see Table 2.

Then we suggest implementing the Agreeing Algorithm in hardware. The main features of the related device, called Circuit Lattice(CL), are:

- No memory locations are necessary as no one bit is kept by the device in common sense. Solutions to particular equations are circuits with two type of switches and the whole system is a network of connections between them represented as a circuit lattice. See Fig. 5 for instance.
- Voltage is induced by variables guess. Its expansion is then directed by switches implemented as electronic relays or transistors on a semiconductor chip. The potential difference detected in some particular circuits indicates the system is inconsistent after the guess.
- The number of input contacts is essentially $2s$, where s is the number of variables guessed during the solution of the system. That is at most $2n$ anyway. Some power contacts and one output contact that sends out a signal when the system is found inconsistent should be added.
- The speed of the device is determined by the time of switching, where lots of switches turn simultaneously. Switches are not necessarily synchronized, so that the device does not work as a conventional computer.

It is very unlikely to solve the system by Agreeing alone. So some guesses on the variable values should be made. The system is then checked for consistence with the Agreeing Algorithm. As most of the guesses should be incorrect, it is important to have an efficient way to check the system's inconsistence. The suggested Circuit Lattice is designed to achieve this goal. Implementing equations from a cipher, it may be used for a brute force attack. When trying the current key, one introduces the guess into the device, and checks whether the system is inconsistent.

Common approaches to the key search [6, 21, 2, 7, 17, 13, 14] are based on the parallelization of the job to many special purpose chips, which efficiently implement the encryption. The best reported speed for one DES encryption with Copacobana is about 0.1 GHz per chip, [13]. Approximately the same speed per each of its SPU is achieved by Cell Processor, see [14]. Therefore about 0.034 GHz for the Triple DES anyway. This is the key rejecting rate.

In contrast, our idea is to not implement any encryption. If constructed, the Circuit Lattice might achieve a higher key rejecting rate, see the discussion in Section 7. Moreover, depending on the equation system from the cipher, the number of key bits necessary to guess before solving or observing an inconsistence may vary. For instance, in [15] it was reported that 37-38 key variables out of 56 are guessed and the rest of the system from 6 rounds of the DES is solved by the Agreeing Algorithm alone. So it is sometimes not necessary to guess all key bits. There may exist a lot of equation systems describing one particular cipher produced, for instance, with the Gluing procedure. Our approach therefore has more flexibility.

It was also reported in [16] that admitting up to 2^s right hand sides(produced with Gluing during system solution) in MRHS equations for the AES-128, one should only guess $128 - s$ of the key bits before the system is solved. A fast way, based on some physical principle, for checking the system's inconsistence after the guess might result in breaking a real world cipher. Two principles may be in use here: electric potential expansion and the expansion of light. We will presently follow the first principle.

This proposal is different from an independent work by Geiselmann, Matheis and Steinwandt, which describes a hardware implementation of main MRHS routines, see [11].

The author is grateful to Håvard Raddum for useful discussions, Thorsten Schilling for indicating a flaw in the first version of Lemma 2 and one of the anonymous referees from WCC'09 for suggestions on improving the presentation.

2 Agreeing Procedure

For equations $E_1 = (X_1, V_1)$ and $E_2 = (X_2, V_2)$, let $X_{1,2} = X_1 \cap X_2$. Then let $V_{1,2}$ be the set of $X_{1,2}$-subvectors of V_1, that is the set of projections of V_1 to variables $X_{1,2}$. Similarly, the set $V_{2,1}$ of $X_{1,2}$-subvectors of V_2 is defined. We say the equations E_1 and E_2 agree if $V_{1,2} = V_{2,1}$. Otherwise, we apply the procedure called Agreeing. All vectors whose $X_{1,2}$-subvectors are not in $V_{2,1} \cap V_{1,2}$ are

deleted from V_1 and V_2. Obviously, we delete V_i-vectors which can't make part of any common solution to the equations. Then we put $E_i \leftarrow (X_i, V_i')$, where $V_i' \subseteq V_i$ consist of the survived vectors.

2.1 Agreeing Algorithm

The goal of the Agreing Algorithm is to identify wrong solutions to equations E_i and remove them from V_i by pairwise application of the Agreeing Procedure. The output doesn't depend on the order of pairwise agreeings, see [16]. Application of the procedure to E_i and E_j where $X_i \cap X_j = \emptyset$ can be avoided. We will show that some pairs E_i, E_j can be avoided too even if $X_i \cap X_j \neq \emptyset$. This significantly optimizes memory requirement of the Agreeing Algorithm and the hardware implementation described in Section 3.

The equations $E_1, \ldots, E_m$ are vertices in an equation graph G. Vertices E_i and E_j are connected by the edge (E_i, E_j) labeled with $X_{i,j} = X_i \cap X_j \neq \emptyset$. There may occur different edges with the same labels. The Agreeing Procedure, being applied to E_i and E_j, implements a kind of information exchange between them through the edge (E_i, E_j). That is for $Y \subseteq X_{i,j}$ the information $Y \neq a$ for some binary string a is transmitted from E_i to E_j or backwards. For simplicity, the same symbol Y also denotes an ordered string of variables Y. We will now show that some of the edges in the graph G are obsolescent in this respect.

A subgraph G_m of G is called minimal if it is on the same vertices and

1. For any (E_i, E_j) in G, there exists a sequence of vertices

$$E_i, E_k, E_l, \ldots, E_r, E_j, \tag{2}$$

   where $(E_i, E_k), (E_k, E_l), \ldots, (E_r, E_j)$ are in G_m and $X_{i,j}$ is a subset in each label $X_{i,k}, X_{k,l}, \ldots, X_{r,j}$.
2. G_m has minimal number of edges.

The edges of a minimal subgraph are called maximal and denoted A for some fixed G_m. Minimal subgraph is not uniquely defined.

Lemma 1. *The Agreeing Algorithm output doesn't depend on whether the Agreeing procedure runs through all edges of G or through only maximal edges.*

Proof. Let $Y \subseteq X_{i,j}$ for the equations E_i and E_j. Assume we learn, from the equation E_i, that $Y \neq a$ for some string a. The Agreeing procedure expands $Y \neq a$ from E_i to E_j. Therefore, there exists a path (2), where

$$Y \subseteq X_{i,j} \subseteq X_i, X_k, X_l, \ldots, X_r, X_j.$$

So $Y \neq a$ is expanded from E_i to E_j through the path (2) by agreeing pairwise E_i, E_k, then E_k, E_l,... and E_r, E_j. This proves the Lemma.

We now formulate the algorithm to compute a minimal subgraph of G:

1. For $Y \subseteq X$ find all edges (E_s, E_r) in G such that $Y \subseteq X_{s,r}$. We only need do that for Y which is a label in G. Denote a subgraph of G on the vertices $E_s, E_r, \ldots$ with all such edges (E_s, E_r) by G_Y. Remark that G_Y is a complete graph.
2. Find the set V_Y of edges (E_s, E_r) in G_Y, where $X_{s,r} = Y$. Find a largest subset $W_Y \subseteq V_Y$ such that G_Y is still connected after removing the edges W_Y. Remark that W_Y is not uniquely defined.
3. Remove the edges W_Y from G for all Y and get G_m.

Lemma 2. *Let G_m be the algorithm's output graph. Then G_m is minimal.*

Proof. We first prove that for any edge (E_i, E_j) in G there is a path (2) on G_m. Let $Y = X_{i,j}$. If (E_i, E_j) is not in W_Y, then it is nothing to prove as (E_i, E_j) in G_m. Assume $(E_i, E_j) \in W_Y$. Then there is a path on G_Y from E_i to E_j through the edges $(E_r, E_s) \notin W_Y$ and $Y \subseteq X_{r,s}$. This is because G_Y remains connected after removing W_Y. If all such $(E_r, E_s) \notin W_{X_{r,s}}$, then the required path is found, as all these edges are in G_m.

Otherwise, assume some $(E_r, E_s) \in W_Z$, where $Z = X_{r,s}$. Therefore $Y \subsetneq Z$ and the edge (E_r, E_s) was removed from G. Then there is a path on G_Z from E_r to E_s through edges $(E_k, E_l) \notin W_Z$. This is because G_Z is still connected after removing the edges W_Z. Moreover, $Y \subsetneq Z \subseteq X_{k,l}$ for (E_k, E_l). If all such $(E_k, E_l) \notin W_{X_{k,l}}$, then the required path is found, as all these edges are in G_m.

Otherwise, we continue so on and stop at some point as the sequence of the graphs $G_Y \supsetneq G_Z \supsetneq \ldots$ is strictly decreasing.

The resulting graph G_m is with minimal number of edges. Otherwise, let be possible to remove one more edge (E_r, E_s) from G_m and still have some path (2) for any (E_i, E_j). Then there exists $Z = X_{r,s}$ and a bigger W_Z such that removing W_Z from G_Z keeps this graph connected. That is impossible by the definition of W_Z. The Lemma is proved.

Example. Let there be five Boolean equations in four variables, where $X_1 = \{x_1, x_2\}, X_2 = \{x_2, x_3\}, X_3 = \{x_3, x_4\}, X_4 = \{x_1, x_3\}$ and $X_5 = \{x_2, x_4\}$. The graph G has 5 vertices and 7 edges: (E_1, E_2) labeled with $X_{1,2} = \{x_2\}$, (E_2, E_3) labeled with $X_{2,3} = \{x_3\}$, and so on. Two edges (E_1, E_2) and (E_2, E_4) are to be removed as they are obsolescent for the Agreeing Algorithm.

2.2 Agreeing2 Algorithm

This is an asymtotically faster variant of the Agreeing Algorithm, see [16].

(Precomputation.) For each maximal edge (E_i, E_j) find the set $X_{i,j}$ and the number $r = |X_{i,j}|$. For each r-bit address b unordered tuple of lists

$$\{V_{i,j}(b); V_{j,i}(b)\} \tag{3}$$

is precomputed. The lists $V_{i,j}(b)$ and $V_{j,i}(b)$ consist of vectors from V_i and respectively V_j whose projection to variables $X_{i,j}$ is b. The set of tuples is

sorted using some linear order. The algorithm marks vectors in tuples (3), then deletes all marked vectors from V_i. We say list $V_{i,j}(b)$ empty if it does not contain any entries or all they are marked.

(Agreeing.) The Algorithm starts with the first tuple $\{V_{i,j}(b); V_{j,i}(b)\}$, where just one list is empty and follows the rules:

1. Let the current tuple be $\{V_{i,j}(b); V_{j,i}(b)\}$, where $V_{i,j}(b)$ is empty, while $V_{j,i}(b)$ is not. Then all the vectors a in $V_{j,i}(b)$ are made marked one after the other.
2. For a in $V_{j,i}(b)$ the projection d of a to variables $X_{j,k}$ is computed, where (E_j, E_k) is a maximal edge. Then a in $V_{j,k}(d)$ is made marked. The tuple $\{V_{j,k}(d); V_{k,j}(d)\}$ is now current.
3. If just one of $V_{j,k}(d)$ or $V_{k,j}(d)$ is found empty, then apply step 1. If not, then take another maximal edge (E_j, E_k) or mark another a in $V_{j,i}(b)$. If $V_{j,i}(b)$ is already empty, then backtrack to the tuple last to $\{V_{i,j}(b); V_{j,i}(b)\}$.
4. For each starting tuple the algorithm walks through a search tree with backtracking. If new deletions do not occur in the current tree, then the next tuple, where just one list is empty, is taken.
5. The algorithm stops when in all tuples $\{V_{i,j}(b); V_{j,i}(b)\}$ the lists both are empty or both non-empty. Then all vectors that have been earlier marked in the tuples are now deleted from V_i.

We remark that each tuple $\{a_1, \ldots, a_r; b_1, \ldots, b_s\}$ implements two implications. First, marking all $\{a_1, \ldots, a_r\}$ implies marking all $\{b_1, \ldots, b_s\}$, which can be denoted $\bar{a}_1, \ldots, \bar{a}_r \Rightarrow \bar{b}_1, \ldots, \bar{b}_s$, and vise versa $\bar{b}_1, \ldots, \bar{b}_s \Rightarrow \bar{a}_1, \ldots, \bar{a}_r$. Agreeing2 Algorithm simply expands marking through these implications.

Lemma 3. *Equations* (1) *are pairwise agreed if and only if in all* $\{V_{i,j}(b); V_{j,i}(b)\}$ *defined for maximal edges* (E_i, E_j) *the lists both are empty or both non-empty.*

Lemma 4. *Let for at least one edge* (E_i, E_j) *the lists* $V_{i,j}(b)$ *be empty for all* b*. Then the system is inconsistent.*

2.3 Example

Let three Boolean equations E_1, E_2, E_3 be given in algebraic normal form:

$$\begin{aligned} 1 + x_3 + x_1x_2 + x_1x_3 + x_1x_2x_3 &= 0, \\ 1 + x_1 + x_4 &= 0, \\ 1 + x_3 + x_2x_4 + x_3x_4 + x_2x_3x_4 &= 0. \end{aligned}$$

Represent them as lists of solutions:

$$\begin{array}{c|ccc} & x_1 & x_2 & x_3 \\ \hline a_1 & 0 & 0 & 1 \\ a_2 & 0 & 1 & 1 \\ a_3 & 1 & 1 & 0 \end{array}, \quad \begin{array}{c|cc} & x_1 & x_4 \\ \hline b_1 & 0 & 1 \\ b_2 & 1 & 0 \end{array}, \quad \begin{array}{c|ccc} & x_2 & x_3 & x_4 \\ \hline c_1 & 0 & 1 & 0 \\ c_2 & 1 & 0 & 1 \\ c_3 & 1 & 1 & 0 \end{array}. \tag{4}$$

The list of tuples: $P = \{a_1, a_2; b_1\}$, $Q = \{a_3; b_2\}$, $R = \{b_1; c_2\}$, $T = \{b_2; c_1, c_3\}$, $U = \{a_1; c_1\}$, $V = \{a_2; c_3\}$, $W = \{a_3; c_2\}$. As there are no tuples with just one list empty, a guess is necessary to start marking. We mark with a bar.

Assume $x_4 = 0$. So b_1 should be marked. We now have two tuples, where just one of the lists is empty: $\{\bar{b}_1; a_1, a_2\}$ and $\{\bar{b}_1; c_2\}$. According to the algorithm, take the first of two. Then a_1 get marked in $\{\bar{b}_1; a_1, a_2\}$ and $\{a_1; c_1\}$. Therefore, c_1 get marked in $\{\bar{a}_1; c_1\}$ and then in $\{c_1, c_3; b_2\}$. Now backtrack and mark a_2 in $\{\bar{b}_1; \bar{a}_1, a_2\}$ and $\{a_2; c_3\}$, and so on. The sequence of marking is represented in Fig.1. Instances in all tuples have been marked. The guess was wrong. We

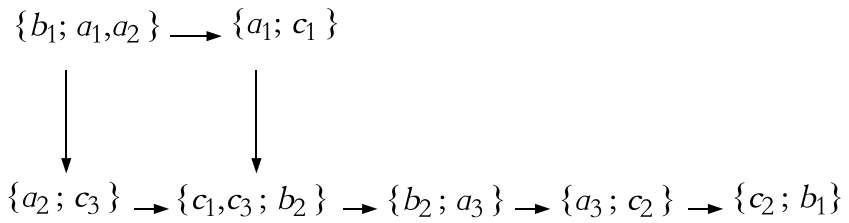


Fig. 1. The marking expansion.

alternatively could add a new tuple $\{b_1; \emptyset\}$ to the tuple list and start marking. Similarly, all tuple lists become empty in case $x_4 = 1$. The system has no solution.

3 Agreeing with a Circuit Lattice

Switches. Circuit lattice is a combination of switches and wires. There are two types of switches as in Fig. 2. Type 1 switch(1-switch) controls vertical circuits connected in parallel and powered by the same battery by any of horizontal circuits also connected in parallel and powered by another battery . So that voltage detected in at least one horizontal circuit makes the switch close. That may induce voltage in all vertical circuits simultaneously. Similarly, type 2 switch(2-switch) controls horizontal circuits connected in parallel by any of vertical circuits; voltage detected in a vertical circuit makes the switch close. That may induce voltage in all horizontal circuits. Only switches with one vertical and one horizontal input circuits are used in this Section in order to construct Circuit Lattice. Later, in Section 4 we will see that using switches with multiple horizontal and vertical input circuits enables constructing Reduced Circuit Lattices with much low number of switches.

Circuit lattice construction. Assume the list of tuples (3) is precomputed. The device is a lattice of horizontal and vertical circuits with intersections at switches of two types as in Fig. 5. The horizontal circuits are in one-to-one correspondence with solutions $a \in V_i$ to equations E_i in (1). So

Fig. 2. The type 1 and 2 switches.

1. each $a \in V_i$ defines the horizontal circuit labeled a as in Fig. 3. 1-switches on the horizontal circuit a are connected either in series or in parallel. We choose here series connection. 2-switches should be connected in parallel.
2. each tuple $\{a_1, \ldots, a_r; b_1, \ldots, b_s\}$ defines two vertical circuits, see Fig. 4. They implement two related implications. The left crosses horizontal circuits $a_1, \ldots, a_r$ at switches of type 1 and $b_1, \ldots, b_s$ at switches of type 2. Therefore it implements implication $\bar{a}_1, \ldots, \bar{a}_r \Rightarrow \bar{b}_1, \ldots, \bar{b}_s$. That means potential in all horizontal circuits $a_1, \ldots, a_r$ implies potential in all horizontal circuits $b_1, \ldots, b_s$ simultaneously. Similarly, the right circuit in Fig. 4 implements another implication $\bar{b}_1, \ldots, \bar{b}_s \Rightarrow \bar{a}_1, \ldots, \bar{a}_r$. Also see Fig. 5, which represents circuit lattice for equations (4).

The number of 1-switches equals the number of 2-switches on each horizontal circuit. This is the number of tuples (3), where a occurs. As the horizontal circuits are labeled by vectors $a \in V_i$, there are $\sum_i |V_i|$ horizontal circuits. Assume

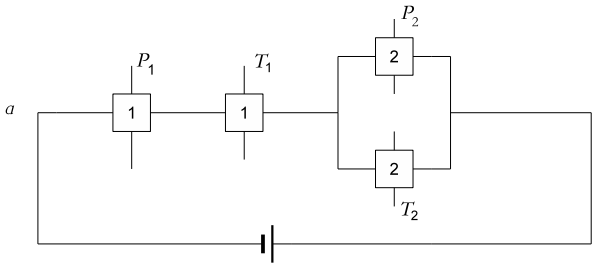


Fig. 3. The horizontal circuit for a particular solution a.

voltage(potential) is detected in a horizontal circuit. That is due to one of 2-switches on that circuit was closed. Then all 1-switches on this circuit get closed too. This may imply voltage in vertical circuits, e.g. in circuits P_1 and T_1 in Fig. 3. That happens if all other 1-switches on these vertical circuits(e.g. on P_1 and T_1) are closed. Then their 2-switches get closed. That affects new horizontal circuits and voltage expands so on. We remark that all horizontal circuits consume power

from the same battery. All vertical circuits may be powered from another battery.

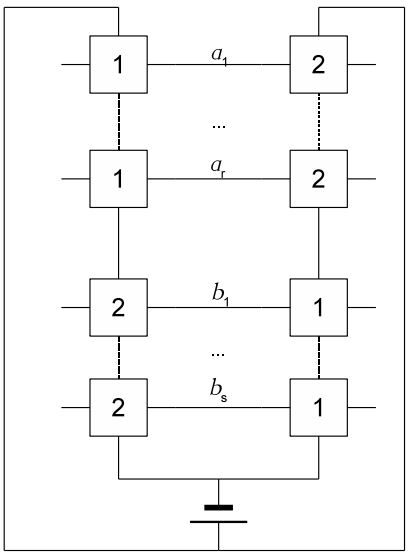


Fig. 4. The vertical circuits defined by $\{a_1, \ldots, a_r; b_1, \ldots, b_s\}$.

Solving. Solving starts with inducing potential into the circuit lattice. The potential may appear due to the tuples with just one of the lists empty. That is similar to Agreeing2 method explained before, as we start the algorithm with such tuples. So potential appears in one of two vertical circuit constructed from $\{\emptyset; b_1, \ldots, b_s\}$ as soon as the battery is switched on. This induces voltage in the horizontal circuits $b_1, \ldots, b_s$. Voltage may be then induced in some new vertical and horizontal circuits, and so on. One easily sees that potential is detected in a horizontal circuit labeled a if and only if a is marked by Agreeing2 algorithm. That is a can't be a part of any common solution to equations (1). Therefore, the following statement is obvious.

Lemma 5. *Assume that after inducing potential in the circuit lattice, it is detected in each horizontal circuit $a_j \in V_i$ for at least one V_i. Then the system is inconsistent.*

If there are no tuples with just one empty list, then the device won't start. So variable guesses are to be introduced to start voltage expansion. Assume we are to guess the value $x \in X_i$ for some equation E_i. Let $a_1, \ldots, a_t$ be all vectors in V_i, where $x = 0$, and $a_{t+1}, \ldots, a_r$ all vectors in V_i, where $x = 1$. Each horizontal circuit $a \in V_i$ is provided with one additional 2-switch. It is connected in parallel with other 2-switches. Two vertical circuits are constructed: S_1 and S_2 by connecting new 2-switches above on horizontal circuits $a_{t+1}, \ldots, a_r$ and

$a_1, \ldots, a_t$ respectively. It is not necessary to use 1-switches here as they won't play any role. To guess $x = 0$ one switches on the vertical circuit S_1, while S_2 is off. To guess $x = 1$ one switches on another vertical circuit S_2 with S_1 is off. See Fig. 5 for an example. Remark that S_1 and S_2 are there constructed for guessing the value x_4 in E_2.

Example. Circuit lattice in case of (4) is represented in Fig. 5. Two vertical

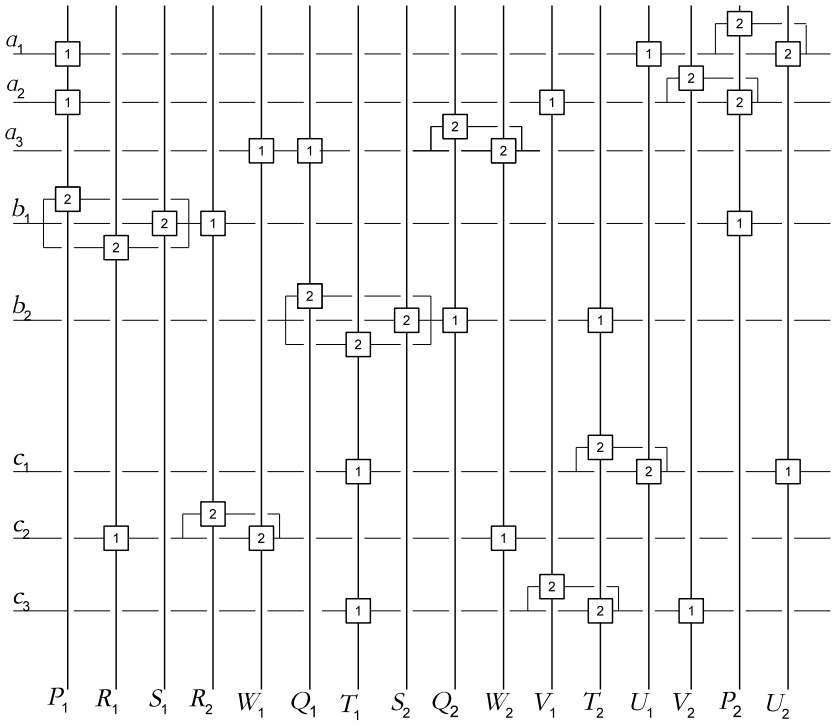


Fig. 5. The circuit lattice for equations (4).

circuits related to tuples $P, Q \ldots$ are denoted $P_1, P_2, Q_1, Q_2 \ldots$. There are two additional circuits S_1 and S_2 used for introducing guesses on x_4. Each of these two circuits incorporates one additional 2-switch. So the device composes of 34 switches on the whole. In order to check $x_4 = 0$, one turns the circuit S_1 on, while S_2 is off. This results in 2-switch on the circuit S_1 get close and voltage appears in the horizontal circuit b_1. Two 1-switches on b_1 get closed and therefore voltage appears in two vertical circuits R_2 and P_2. All 2-switches on them become closed and voltage expands to the horizontal circuits a_1, a_2, c_2 and so on. Finally, after a number of simultaneous switch turns, voltage is detected in all horizontal circuits. The guess was wrong. Similarly, the circuit S_2 is switched on, S_1 is off, in order to check $x_4 = 1$. All horizontal circuits get voltage. The guess was wrong too. The system is therefore inconsistent.

The number of switches. The main characteristic of the device is the number of switches. This is twice the number of vectors in all tuples (3) for maximal edges and computed by the formula

$$2\sum_{A}\sum_{b}(|V_{i,j}(b)|+|V_{j,i}(b)|)=2\sum_{A}(|V_i|+|V_j|). \tag{5}$$

The external sum is over all maximal edges $(E_i, E_j) \in A$ in G. For guessing s variables $x_1 \in X_{i_1}, \ldots, x_s \in X_{i_s}$ there should be also $|V_{i_1}|+\ldots+|V_{i_s}|$ additional switches.

The number of wires. We also count the number of wires necessary to connect switches in the circuit lattice. The number of wires in all vertical circuits is obviously the number of the lattice switches (5) plus the number of vertical circuits themselves. The latter value equals twice the number of tuples. In a horizontal circuit the type 2 switches are connected in parallel. So the number of wires is the number of type 1 switches plus twice the number of type 2 switches plus two. Therefore, the number of wires in all horizontal circuits is $3\sum_A(|V_i|+|V_j|)+2\sum_i|V_i|$. So the total number of wires should be

$$5\sum_{A}(|V_i|+|V_j|)+2\sum_{i}|V_i|+2\sum_{\text{tuples}}1. \tag{6}$$

For guessing s variables $x_1 \in X_{i_1}, \ldots, x_s \in X_{i_s}$ there should be also $|V_{i_1}|+\ldots+|V_{i_s}|+2s$ additional wires.

4 Reduced Circuit Lattices

In this Section we briefly discuss how to reduce the parameters of the device(the number of switches and wires) through using switches that control several circuits connected in parallel, and controlled themselves by any of several parallel circuits, see Fig.2.

First, we modify each horizontal circuit so that it now comprises only one 1-switch and one 2-switch. The same 1-switch now controls all vertical circuits that passed via 1-switches on a horizontal circuit in above Circuit Lattice(CL). Then the same 2-switch controls that horizontal circuit by any of vertical circuits passing via 2-switches in CL. So the horizontal circuit in Fig.3 now transforms into that in Fig.6. We keep all vertical circuits intact. The number of switches becomes $2\sum_i|V_i|$, while the number of wires is essentially $2\sum_A(|V_i|+|V_j|)$. We call the described device Reduced Circuit Lattice 1(RCL1). It operates similarly to how CL operates.

We will further reduce the device parameters by observing that one 2-switch can control several horizontal circuits. We keep one type 1 switch on each horizontal circuit as above. Particular $a \in V_i$ are in one-to-one correspondence with

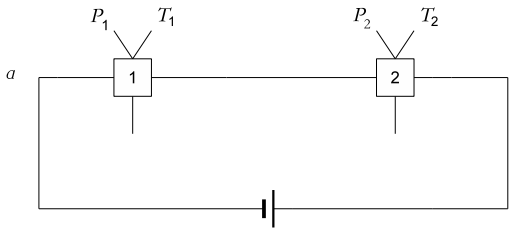


Fig. 6. The reduced horizontal circuit for a particular solution a.

1-switches. So that we say there is voltage in the horizontal circuit a if the related 1-switch is closed. However the connections in the vertical circuits related to a tuple are now as in Fig. 7, compare with that in Fig. 4. The number of switches now becomes $\sum_i |V_i| + 2\sum_{\text{tuples}} 1$, while the number of wires is essentially $2\sum_A(|V_i| + |V_j|)$. We call the described device Reduced Circuit Lattice 2(RCL2).

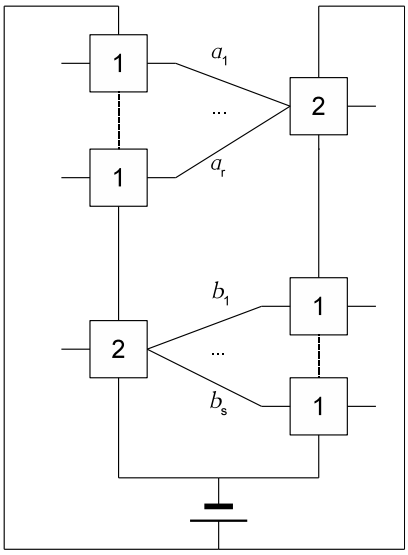


Fig. 7. The reduced vertical circuits defined by $\{a_1, \ldots, a_r; b_1, \ldots, b_s\}$.

5 Guessing the variable values

Equations from a cipher. The number of key variables is commonly very small if compared with all system variables. Guessing all key variables results in the

whole system collapses by any of the Agreeing Algorithms. This is a variant of the brute force attack. If Agreeing works faster than this cipher encryption, then an advantage over common brute force attack is observed. It might be well that a proper subset of key variables should be guessed before the system is solved with Agreeing, see this paper Introduction, where the issue is briefly discussed.

Random equations. Generally, s-variable guesses result in 2^s trials(Agreeing runs). However, in randomly generated sparse equations there is a more efficient approach based on Gluing [15]. Assume that an s-bit guess is enough for solving (1) or finding it inconsistent with Agreeing. Look at the gluing of some t equations:

$$(X(t), U_t) = (X_{i_1}, V_1) \circ (X_{i_2}, V_2) \circ \ldots \circ (X_{i_t}, V_t),$$

where $s = |X(t)|$ and $X(t) = X_{i_1} \cup X_{i_2} \cup \ldots \cup X_{i_t}$. In other words, U_t is the set of all common solutions to the equations $E_{i_1}, \ldots, E_{i_t}$. The number of vectors in U_t is 2^{s-t} on the average, see Lemma 4 in [18]. The vectors U_t are produced one after the other as in [18] with the cost per vector proportional to t. So it is not necessary to keep the whole set U_t. This is true for t smaller than some critical value $\frac{\alpha_0 n}{l}$, where $\alpha_0 = 2^{1/l} \ln(\frac{1-1/2}{1-(1/2)^{1/l}})$, see [18]. So the total complexity of solving is roughly proportional to 2^{s-t} of Agreeing runs.

6 DES and Triple DES equations

The DES and the Triple DES equation systems are constructed in Appendix, where input/output 64-bit blocks are considered variables too. So each equation comprises 20 variables and admits 2^{16} solutions. Table 2 provides with the equation system parameters as the number of equations, the number of variables, the number of edges of the adjacent graph with nonempty labels, the number of maximal edges and the number of tuples (3). Any of Circuit Lattices may be

Table 2. DES and Triple DES equations.

Nmbr of	eqns	vrbls	edges	mx.edges	tuples
DES	128	632	3545	1409	16636
TDES	384	1712	16831	3929	71320

.

used to compute the key for any given plain-texts and related cipher-texts. These are introduced into a Circuit Lattice similarly to the guessed key-bits. However plain-text, cipher-text bits are not changing during the whole computation. So any CL should have $2 \times 56 + 2 \times 128 = 368$ input contacts for the DES and $2 \times 112 + 2 \times 128 = 480$ input contacts for the Triple DES.

Tables 3 and 4 show main characteristics of Circuit Lattices for DES and Triple DES: the number of necessary switches, wires and input contacts, which are computed by formulas (5) and (6) and in Section 4.

Table 3. DES Circuit Lattice implementations.

Nmbr of	switches	wires	input contacts
CL	3.9×10^8	9.5×10^8	368
RCL1	1.7×10^7	3.9×10^8	368
RCL2	8.5×10^6	3.9×10^8	368

.

Table 4. TDES Circuit Lattice implementations.

Nmbr of	switches	wires	input contacts
CL	1.1×10^9	2.7×10^9	480
RCL1	5.1×10^7	1.1×10^9	480
RCL2	2.6×10^7	1.1×10^9	480

.

Two plain-text, cipher-text 64-bit blocks uniquely define 112-bit key in the Triple DES. So for the key search there should be two above described devices working in parallel. The speed of computation is determined by the time that a switch takes to turn. However, how many switch turns are necessary before the system is found inconsistent looks generally difficult to estimate. This is an open problem. Voltage expands in a highly parallel manner through several circuits which affect each other and many switches turn simultaneously. Fortunately, this is easy for round ciphers like DES or Triple DES. Assume guessing all key variables at once. Then all Type 1 switches in tuples related to pairs of equations in subsequent rounds turn simultaneously when voltage expands from one round to another. That makes related Type 2 switches turn too. This is so even if the Agreeing only runs through maximal edges of the adjacent graph. Therefore the time measured in switch turns that the solver takes to agree pairwise all equations is twice the number of rounds. In particular, to reject one wrong key in the Triple DES takes at most 2×48 switch turns.

7 Conclusion, open problems and discussion

The paper describes a hardware implementation of the Agreeing Algorithm aimed to find solutions to a system of sparse Boolean equations, e.g. coming from ciphers. Some variables guess is introduced into the device which signals out if the system is inconsistent after that guess. The device architecture implemented with a lattice of circuits is transparent. However, this is an open problem whether the circuit lattice for a real world cipher like DES or Triple DES is implementable within the current technology in computer industry.

There are several related problems:

1. The number of switches is the most important parameter of the solver. Table 3 and 4 data shows that the equation systems for the DES and the Triple DES require the number of switches which is within the number of transistors now available on one semiconductor crystal. For instance, Intel announced

Dual-Core Itanium2 processor with more than 1.7 billion transistors, see [9]. Obviously, a transistor is able to work as a switch.

2. Special purpose hardware to supply one after the other guesses on fixed variables is to be devised. Its speed should be comparable with that of the solver. The device is similarly constructed in wires and switches and controlled by the output signal from the solver. We do not discuss this in detail as its construction is rather obvious. It is easy to understand that its speed should be only 2 switch turns on the average.
3. The transistor speed(the speed of a turn) is constantly increasing. E.g., historical 17% year performance improvement is also predicted in [23] for the next decade. Then a new speed record for the world fastest transistor which is more than 1THz(1000GHz), see [10], was reported. However, to be on the safe side we assume available transistors with speed about 100GHz. Assume it is feasible to integrate one billion or so such transistors on one semiconductor chip as a Triple DES Circuit Lattice. Remark that Reduced Circuit Lattices require much low number of transistors, see Table 4 . Then average time for producing a guess on 112 key variables and finding the system's inconsistence is approximately $2 \times 48 + 2 = 98$ switch turns. So the key rejecting rate is approximately 1GHz in this case. It is compared favorably with what is currently achieved, about 0.034GHz.

References

1. M. Bardet, J.-C.Faugére, and B. Salvy, *Complexity of Gröbner basis computation for semi-regular overdetermined sequences over F_2 with solutions in F_2,* Research report RR–5049, INRIA, 2003.
2. R. Clayon and M.Bond, *Experience using a low-cost FPGA design to crack DES keys*, in CHES 2002, LNCS 2523, pp. 579–592, Springer-Verlag, 2002.
3. N. Courtois, A. Klimov, J. Patarin, and A. Shamir, *Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations,* in Eurocrypt 2000, LNCS 1807, pp. 392–407, Springer-Verlag, 2000.
4. N. T. Courtois and G. V. Bard, *Algebraic Cryptanalysis of the Data Encryption Standard*, Cryptology ePrint Archive: Report 2006/402.
5. N. T. Courtois, G. V. Bard, and D. Wagner, *Algebraic and Slide Attacks on KeeLoq*, Cryptology ePrint Archive: Report 2007/062.
6. W. Diffie and M. Hellman, *Exhaustive cryptanalysis of the NBS Data Encryption Standard*, Computer, 10(6), 1977, pp.74–84.
7. Electronic Frontier Foundation, *Cracking DES: Secrets of Encryption Research, Wiretap Politics and Chip Design*, O'Reilly and Assotiates Inc., 1998.
8. J.-C. Faugère, *A new efficient algorithm for computing Gröbner bases without reduction to zero (F5),* Proc. of ISSAC 2002, pp. 75 – 83, ACM Press, 2002.
9. http://www.intel.com
10. http://www.semiconductor.net/article/CA6514491.html
11. W. Geiselmann, K. Matheis and R. Steinwandt, *PET SNAKE: A Special Purpose Architecture to Implement an Algebraic Attack in Hardware,* Cryptology ePrint Archive, 2009/222.
12. K. Iwama, *Worst-Case Upper Bounds for kSAT,* The Bulletin of the EATCS, vol. 82(2004), pp. 61–71.

13. S. Kumar,C. Paar, J. Pelzl,G. Pfeiffer, and M. Schimmler, *Breaking ciphers with Copacobana-a cost-optimized parallel code breaker*, in CHES2006, LNCS 4249, pp. 101-118, 2006.
14. D.A. Osvik, E. Tromer, *Cryptologic applications of the Playstation3: Cell SPEED*, http://www.hyperelliptic.org/SPEED/slides/Osvik_cell-speed.pdf
15. H. Raddum and I. Semaev, *New technique for solving sparse equation systems*, Cryptology ePrint Archive, 2006/475.
16. H. Raddum and I. Semaev, *Solving Multiple Right Hand Sides linear equations*, Designs, Codes and Cryptography, vol. 49(2008), pp. 147–160, extended abstract in Proceedings of WCC'07, 16-20 April 2007, Versailles, France, INRIA, pp. 323–332.
17. G. Rouvroy, F.-X. Standaert, J.-J. Quisquater, and J.-D. Legat, *Design Strategies and Modified desciptions to optimize cipher FPGA implementations: fast and compact results for DES and Triple-DES,* in FPL2003, LNCS 2778, pp. 181–193, 2003.
18. I. Semaev, *On solving sparse algebraic equations over finite fields*, Designs, Codes and Cryptography, vol. 49(2008), pp. 47–60, extended abstract in Proceedings of WCC'07, 16-20 April 2007, Versailles, France, INRIA, pp. 361–370.
19. I. Semaev, *Sparse algebraic equations over finite fields*, to appear in SIAM Journal on Computing, 2009, see also in Cryptology ePrint Archive, 2007/280.
20. B.-Y. Yang, J-M. Chen, and N.Courtois, *On asymptotic security estimates in XL and Gröbner bases-related algebraic cryptanalysis,* in ICICS 2004, LNCS 3269, pp. 401–413, Springer-Verlag, 2004.
21. M.J. Wiener, *Efficient DES key search*, In Willam R.Stalling, editor, Practical Cryptography for Data Interworks, pp.31–79, IEEE Computer Society Press,1996.
22. A. Zakrevskij, I. Vasilkova, *Reducing large systems of Boolean equations,* 4th Int.Workshop on Boolean Problems, Freiberg University, September, 21–22, 2000.
23. P. Zeitzoff, *2007 International Technology Roadmap: MOSFET scaling challenges,* Solid State Technology Magazine, February 2008.

8 Appendix

In this Appendix we describe how to make the equation system from the DES algorithm. The similar equations are constructed for the Triple DES. The input and output applications of the permutation IP are ignored as well as the final swap between 32-bit sub-blocks. The 64-bit internal state of the cipher after the i-th round is denoted by (R_{i-1}, R_i). In particular, (R_{-1}, R_0) denotes the 64-bit plain-text block and (R_{15}, R_{16}) is the related cipher-text block. All these 128 bits are generally considered known constants. But we write them variables. So that when the Agreeing algorithm is being run, these 128 variables are substituted by constants as if for guessing. Therefore, 576 state variables are bits of $R_{-1}, R_0, R_1, \ldots, R_{15}, R_{16}$. They are numbered $-63, -62, \ldots, 512$. 56 key variables are numbered by $512 + j$, where $1 \leq j \leq 64$ and $j \neq 8, 16, \ldots, 64$.

At every round $i = 1, 2, \ldots, 16$, sub-blocks R_i are related as

$$R_i \oplus R_{i-2} = PS(\overline{R_{i-1}} \oplus K_i), \tag{7}$$

where $\overline{R_{i-1}}$ is the 48-bit expansion of the 32-bit R_{i-1} and K_i is the round key. P denotes the fixed permutation on 32 symbols and S is the transform implemented

by 8 S-boxes. The equation (7) is equivalent to 8 equations related to each of the S-boxes S_j:

$$(P^{-1}(R_i))_j \oplus (P^{-1}(R_{i-2}))_j = S_j((\overline{R_{i-1}})_j \oplus K_{i,j}), \tag{8}$$

where $R_{i,j}$ is a 4-bit sub-block of R_i, and $K_{i,j}$ is a 6-bit sub-block of K_i and $(T)_j$ denotes a 6(or 4)-bit sub-block of T. The equation (8) is denoted by $E_{i,j} = E_{j+8(i-1)}$. The full system of the DES equations consists of 128 equations E_t, $t = 1, 2, \ldots, 128$. One equation incorporates 20 variables. For instance, $E_{8,4} = E_{60}$ depends on 20 variables:

$$\begin{aligned}(P^{-1}(R_6))_4 &= (x_{161}, x_{170}, x_{180}, x_{186}),\\ (\overline{R_7})_4 &= (x_{204}, x_{205}, x_{206}, x_{207}, x_{208}, x_{209}),\\ (P^{-1}(R_8))_4 &= (x_{225}, x_{234}, x_{244}, x_{250}),\\ K_{8,4} &= (x_{514}, x_{529}, x_{538}, x_{539}, x_{556}, x_{561}).\end{aligned}$$

These variables compose the set X_{60}. For any values of the following 16 variables:

$$x_{204}, x_{205}, x_{206}, x_{207}, x_{208}, x_{209}, x_{225}, x_{234},$$
$$x_{244}, x_{250}, x_{514}, x_{529}, x_{538}, x_{539}, x_{556}, x_{561},$$

the values of $x_{161}, x_{170}, x_{180}, x_{186}$ are uniquely defined by (8). So 2^{16} vectors of length 20 compose the list V_{60}. That is all equations have 2^{16} solutions. Let $m \to E_K(m)$ denote the encryption function on plain-text blocks with the DES algorithm. Then the Triple DES implements the mapping:

$$m \to E_{K_1}(E_{K_2}(E_{K_1}(m))).$$

Therefore Triple DES equations are determined similarly to those for the DES.

Pollard Rho on the PlayStation 3

Joppe W. Bos[1], Marcelo E. Kaihara[1], and Peter L. Montgomery[2]

[1] EPFL IC LACAL, CH-1015 Lausanne, Switzerland
{joppe.bos, marcelo.kaihara}@epfl.ch
[2] Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
peter.montgomery@microsoft.com

Abstract. This paper describes a high-performance PlayStation 3 (PS3) implementation of the Pollard rho discrete logarithm algorithm on elliptic curves over prime fields. A record has been set using this implementation by solving an elliptic curve discrete logarithm problem (ECDLP) with domain parameters from a currently standardized elliptic curve over a 112-bit prime field. Solving this 112-bit ECDLP instance required 62.6 PS3 years. Arithmetic algorithms have been designed for the PS3 to exploit the SIMD architecture and the rich instruction set of its computational units. Though our implementation is targeted at a specific 112-bit modulus, most of our implementation strategies apply to other large moduli as well.

Keywords: Elliptic curve discrete logarithm, Pollard rho, Cell broadband engine, SIMD arithmetic

1 Introduction

Elliptic curve cryptography (ECC) [20, 23] is becoming increasingly popular since it allows smaller key-sizes [22] to obtain the same level of security as other widely used public-key cryptographic approaches such as RSA [30]. Government and industry have standardized the use of ECC in, for instance, the Digital Signature Standards (DSS) [38] and the Standards for Efficient Cryptography (SEC) [6]. Here, elliptic curves defined over prime fields ranging from 192 to 512 bits, and from 112 to 512 bits are standardized, respectively.

Processor development seems to be moving away from a single-core towards a multi-core design in order to scale performance through parallelism. The Cell broadband engine (Cell), with its unique heterogeneous architecture, is an interesting example. Its single instruction multiple data (SIMD) organization along with its rich instruction set makes it attractive for accelerating cryptographic operations [8, 2, 7, 3] and cryptanalysis [34, 35].

In this article, the security of ECC – using elliptic curves over prime fields – is evaluated using the relatively low-priced and broadly available multi-core Cell architecture, which is the heart of the video game console PlayStation 3 (PS3). For this purpose, high-performance SIMD arithmetic algorithms have been designed to exploit the features of the instruction set of the Cell. These

SIMD algorithms form the basis of our implementation of the Pollard rho [28] algorithm, the fastest known algorithm to solve the Elliptic Curve Discrete Logarithm Problem (ECDLP). Our implementation has been used to set a record by solving an ECDLP with parameters taken from a 112-bit standardized elliptic curve. Solving this problem required 62.6 PS3 years and ran on a PS3 cluster of more than 200 PS3s in the period January - July, 2009. When run continuously, using the latest version of our code, the calculation would have taken 3.5 months.

The rest of the paper is organized as follows. Section 2 gives a brief overview of the Cell broadband engine. Section 3 recalls the Pollard rho discrete logarithm algorithm together with some optimizations. Section 4 presents efficient arithmetic algorithms aimed at the 112-bit elliptic curve designed to exploit the features of the Cell architecture. Section 5 gives implementation details and performance results. Section 6 concludes the paper.

2 Cell Broadband Engine Architecture

The Cell architecture [16], developed by Sony, Toshiba and IBM, has as a main processing unit, a dual-threaded 64-bit Power Processing Element (PPE) which can offload work to the eight Synergistic Processing Elements (SPEs) [11, 36]. The SPEs are the workhorses of the Cell and the main interest in this article. The SPE consists of a Synergistic Processor Unit (SPU) and a Memory Flow Controller (MFC). Each SPU has a register file of 128 entries called vectors, or quad-words, of 128-bit length and to its own 256-kilobyte Local Store (LS) with room for instructions and data. The main memory can be accessed through explicit Direct Memory Access (DMA) requests to the MFC. The SPUs have a 128-bit SIMD organization allowing sixteen 8-bit, eight 16-bit or four 32-bit integer computations in parallel. The SPUs are asymmetric processors, having two pipelines, denoted as even and odd pipelines. This means that two instructions can be dispatched every clock cycle. Most of the arithmetic instructions are executed on the even pipeline and most of the memory instructions are executed on the odd pipeline. It is a challenge to fully utilize both pipelines always at the same time. The SPEs have no hardware branch-prediction. Instead, the programmer (or the compiler) can provide hints to the instruction fetch unit where a branch instruction will most likely jump to.

An additional advantage of the SPEs is the rich instruction set. For instance, among the available instructions all distinct binary operations $f : \{0,1\}^2 \to \{0,1\}$ are present. The SPEs are equipped with a 4-SIMD multiplier which can compute four 16-bit integer multiplications simultaneously per clock cycle. In addition, a multiply-and-add instruction which performs a 16×16-bit unsigned multiplication, and an addition of a 32-bit unsigned operand to the 32-bit product is provided and has the same time cost as a single 16×16-bit multiplication. This instruction requires the 16-bit operands to be placed in the higher positions of the 32-bit word elements of the vectors. Note that carries are not generated for these instructions.

One of the first applications of the Cell was to serve as the main processor for the Sony's PS3 video game console. The Cell contains eight SPEs, and in the PS3 one of them is disabled. One of the remaining SPEs is reserved by Sony's hypervisor (a software layer which is used to virtualize devices and other resources in order to provide a virtual machine environment to operating systems such as Linux OS). All in all, six SPEs can be accessed when the Linux operating system is installed on a PS3.

3 Preliminaries

3.1 Elliptic Curves over $\mathbb{F}_p$

Let $\mathbb{F}_p$ be a finite field of characteristic $p \neq 2, 3$ and $a, b \in \mathbb{F}_p$ satisfy the inequality $4a^3 + 27b^2 \neq 0$. Informally an elliptic curve $E(\mathbb{F}_p)$ is defined as the set of points $(x, y) \in \mathbb{F}_p \times \mathbb{F}_p$ which satisfy the affine Weierstrass equation [33]:

$$y^2 = x^3 + ax + b. \tag{1}$$

These points, together with a point at infinity, denoted as $\mathcal{O}$, form an abelian group where the group operation is point addition and the zero point is the point at infinity. Let $P, Q \in E(\mathbb{F}_p) \setminus \{\mathcal{O}\}$, where $P = (x_1, y_1)$ and $Q = (x_2, y_2)$. Then $-P = (x_1, -y_1)$. If $P \neq -Q$ then $P + Q = (x_3, y_3)$ where

$$x_3 = \mu^2 - x_1 - x_2, \quad y_3 = \mu(x_1 - x_3) - y_1 \text{ with } \mu = \begin{cases} \dfrac{y_2 - y_1}{x_2 - x_1} \text{ if } P \neq Q \\ \dfrac{3x_1^2 + a}{2y_1} \text{ if } P = Q. \end{cases} \tag{2}$$

3.2 The Pollard Rho Algorithm

Let E be an elliptic curve over $\mathbb{F}_p$, $P \in E(\mathbb{F}_p)$ a point of order n and $Q = lP \in \langle P \rangle$. Here p is prime and $l, n \in \mathbb{Z}$, in practice p and n are known. The most efficient algorithm in the literature to find $l \bmod n$ for generic curves is Pollard's rho algorithm [28]. The underlying idea of this method is to search for two distinct pairs $(c_i, d_i), (c_j, d_j) \in \mathbb{Z}/n\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z}$ such that

$$c_i P + d_i Q = c_j P + d_j Q.$$

Then, the discrete logarithm of Q to the base P, i.e. $l = \log_P Q$, can be obtained by computing

$$l \equiv (c_i - c_j)(d_j - d_i)^{-1} \bmod n.$$

This calculation might fail if the inverse of $(d_j - d_i)$ does not exist. In practice, n is prime since one first reduces the calculation of the discrete logarithm to the computation of the discrete logarithm in the prime order subgroups of $\langle P \rangle$ [27].

The occurrence of two such distinct pairs is called a *collision*. Given an iteration function $f : \langle P \rangle \to \langle P \rangle$, the Pollard rho method calculates a sequence of

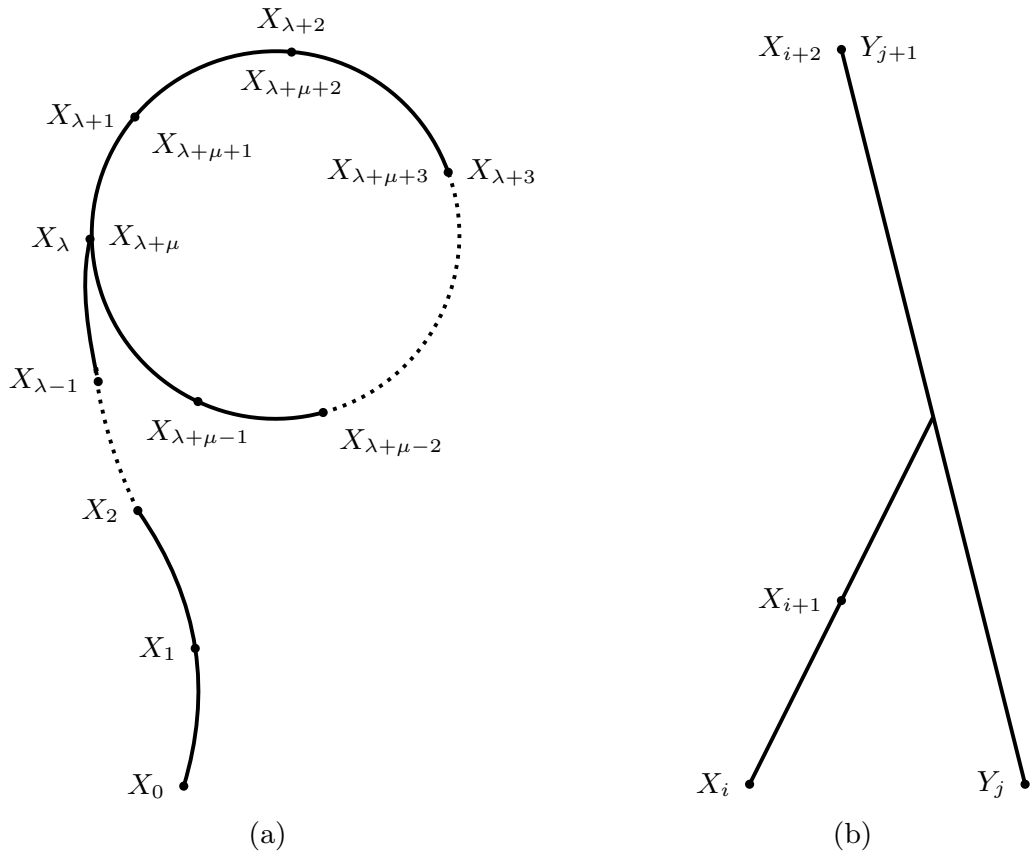


Fig. 1. *Representation of the ρ and λ shape of the single-instance Pollard rho 1(a) and the multi-instance Pollard rho method 1(b) respectively. The points X_i, Y_j represent distinguished points from two different walks.*

points $X_{i+1} = f(X_i)$, $i \geq 0$. The sequence of points represents a walk through the set of points $\langle P \rangle$. Given $X_i = c_i P + d_i Q$ and $c_i, d_i \in [0, n-1]$, f updates c_{i+1} and d_{i+1} and computes X_{i+1} as $X_{i+1} = c_{i+1} P + d_{i+1} Q$. The sequence is started from a random and known point $X_0 \in \langle P \rangle$ by selecting random values for c_0 and d_0. This sequence of points eventually collides (as operations are performed over a finite cyclic group). Let us denote λ and $\mu \geq 1$ as the smallest numbers such that $X_\lambda = X_{\lambda+\mu}$ holds. The value λ is called the tail and μ the cycle length, graphically the walk through the set of points forms a ρ shape: see Fig. 1(a). Assuming the iteration function is a random mapping of size $n = |\langle P \rangle|$, i.e. f is equally probable among all functions $F : \langle P \rangle \rightarrow \langle P \rangle$, Harris showed that the expected values of λ and μ are $\lambda = \mu = \sqrt{\frac{\pi n}{8}}$ when $n \rightarrow \infty$ [15]. The advantage of the Pollard rho method is that it uses a negligible amount of memory, by using Floyd's cycle finding method [19], compared to the baby-step-giant-step [32] method which has the same asymptotic run-time complexity.

3.3 Parallelization

In [39], van Oorschot and Wiener present a time-memory trade-off method based on the work by Quisquater and Delescaille [29]. In order to run many instances of the Pollard rho method on different processors, in order to speed up the calculation of the discrete logarithm, each instance starts with a unique value. The idea is to distinguish points in the walk using a specific property and share the distinguished points among all the processors by communicating them to a central database. Distinguished points (DTP) can be, for example, those with an x-coordinate that is divisible by 2^m, for some $m > 0$, after being normalized to $[0, p-1]$. The search for a collision among the DTPs is performed in this central database. This technique leads to a linear speed-up on the number of processors. Graphically, colliding walks form a λ shape: see Fig. 1(b).

3.4 Adding Walks

The iteration function proposed by Pollard in [28] divides $\langle P \rangle$ into three different partitions: one partition is used to double the current point while in the other two partitions a constant is added. Teske introduces in [37], based on the work by Schnorr and Lenstra [31], a class of walks for the iterating function of Pollard's rho method which achieves a similar performance, in terms of the number of iterations needed, compared to a random mapping. The main idea consists in dividing $\langle P \rangle$ into r different partitions using a partition function $h : \langle P \rangle \rightarrow [0, r-1]$.

To each partition a point is associated; for partition j the values m_j and n_j are randomly chosen in the initialization phase and $R_j = m_j P + n_j Q$ is associated with this partition. If the parallelized version of the Pollard rho method is used, the same m_j, n_j, h should be used in all instances.

The iteration function is defined as

$$X_{i+1} = f(X_i) = X_i + R_{h(X_i)}. \tag{3}$$

It is shown in [37] that values of $r \geq 16$ partitions provide performance comparable to the expected values from random mappings, overcoming a loss of approximately 20 percent of computation time that occurs when Pollard's original iteration function is used.

3.5 Montgomery's Simultaneous Inversion

Elliptic curves can be parameterized in different ways, resulting in different operation counts (cf. [1]). Since many independent walks can be processed conjointly, Montgomery's inversion technique [25], which enables to trade M inversions for $3(M-1)$ multiplications and one inversion, can be used. This places the affine Weierstrass coordinate system as the most suitable candidate. For a point addition, the cost of computing the x-coordinate is four multiplications, one squaring and $\frac{1}{M}^{\text{th}}$ inversion, when M group additions are processed in parallel. By reusing intermediate results of this computation, the y-coordinate can be computed with an additional cost of one field multiplication, see Equation (2).

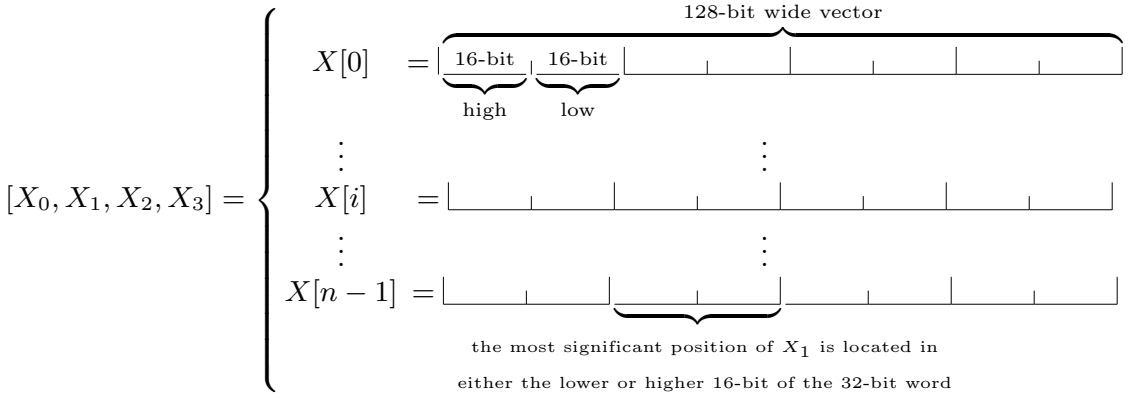


Fig. 2. *Four numbers arranged, in either all lower or higher parts.*

3.6 The Negation Map

The computation of the negative of a point $P = (x, y)$, i.e. $-P = (x, -y)$, is computationally cheap. This observation is used by Wiener and Zucherato [40] to reduce the search space by a factor of two. Given an equivalence relation $\sim$ on $\langle P \rangle$ the idea is to iterate over the set of equivalence classes $\langle P \rangle / \sim$. This can be accomplished by computing $\pm P$ and selecting the point with the smaller y-coordinate after being normalized to $[0, p-1]$. When the negation map is used, almost all equivalence classes have two elements, giving a theoretical speed-up factor of $\sqrt{2}$. This technique can be applied to all elliptic curves. In general, if most equivalence classes contain m points, the search space is reduced by a factor m. Hence, the total required number of iterations is reduced by a factor $\sqrt{m}$. Other examples of equivalence relations, aimed at anomalous binary curves, and more detailed information can be found in [40, 13, 10].

4 112-bit Elliptic Curve Domain Parameters over $\mathbb{F}_p$

As of 2009, the smallest standardized elliptic curve is over a 112-bit prime field. This elliptic curve is standardized in the Standard for Efficient Cryptography (SEC), SEC2: Recommended Elliptic Curve Domain Parameters [6] as curve *secp112r1* and in the Wireless Transport Layer Security Specification [12] as *curve number 6*.

4.1 Integer Representation in the Cell

For a high-performance implementation of arithmetic algorithms on the Cell, vectorization techniques (cf. [9]) are applied and data are represented using a 4-SIMD organization. If the radix of the number system is $r = 2^w$, with $0 < w \leq 32$, then a b-bit number is represented using $n = \lceil \frac{b}{w} \rceil$ digits. In the 4-SIMD representation, four b-bit numbers X_0, X_1, X_2, and X_3 are stored in n vectors.

Each vector $X[j]$, $0 \leq j < n$, holds four w-bit digits of the four numbers that correspond to the same digit position. The notation $[X_0, X_1, X_2, X_3]$ means that the four numbers X_0, X_1, X_2, and X_3 are grouped using 4-SIMD and operations are applied in parallel digit-wise (for the same digit positions) for all the four numbers. For modular multiplication, $w = 16$ is selected, cf. Section 4.2, and each of the n vectors is composed of four 32-bit word elements, where the 16-bit digits of four numbers are stored either in the higher or lower positions of these 32-bit word elements. Hence, each of the four b-bit numbers is represented as $X_i = \sum_{j=0}^{n-1} r^j \left(\left\lfloor \frac{X[j]}{r^{2 \cdot i + h}} \right\rfloor \bmod r \right)$ for $i \in \{0, 1, 2, 3\}$ and $h \in \{0, 1\}$, where h is 1 if data are placed in the higher bit positions and 0 otherwise. Fig. 2 depicts the data structure. For modular inversion, $w = 32$ and each of the four b-bit numbers is represented as $X_i = \sum_{j=0}^{n-1} r^j \left(\left\lfloor \frac{X[j]}{r^i} \right\rfloor \bmod r \right)$ for $i \in \{0, 1, 2, 3\}$ and adjusting the value n accordingly.

4.2 Arithmetic

The standardized elliptic curve *secp112r1* is over $\mathbb{F}_p$. Here p is prime and has the special form: $p = \frac{2^{128}-3}{11 \cdot 6949}$. In order to speed up modular multiplication and subtraction in the Pollard rho algorithm we use a redundant representation taking a larger modulus $\widetilde{p} = 2^{128} - 3 = 11 \cdot 6949 \cdot p$.

Modular Multiplication One computationally intensive operation in the point addition on the elliptic curve is modular multiplication. Furthermore, as Montgomery's simultaneous inversion technique is used to trade one modular inversion by approximately three modular multiplications, the performance of the Pollard rho algorithm highly depends on the performance of the modular multiplication.

In order to increase computation speed, operations are performed in a residue class of a larger modulus $\widetilde{p}$. This redundant representation significantly accelerates modular reduction and successive operations can be performed in this representation.

Let us define a reduction function $\mathfrak{R}$.

Definition 1. *Let $R = 2^{128}$ and $\widetilde{p} = R - 3$. Given an integer $0 \leq x < R^2$ represented in radix R; $x = x_h \cdot R + x_l$, define a map $\mathfrak{R} : \mathbb{Z}/R^2\mathbb{Z} \rightarrow \mathbb{Z}/R^2\mathbb{Z}$ such that*

$$y = \mathfrak{R}(x) = (x \bmod R) + 3 \cdot \left\lfloor \frac{x}{R} \right\rfloor$$

Note that, if $y = \mathfrak{R}(x)$ then $x \equiv y \bmod \widetilde{p}$ and $y \leq x$.

Furthermore, with high likelihood $\mathfrak{R}$ can be used to quickly reduce values modulo $\widetilde{p}$. Because $0 \leq \mathfrak{R}(x) < 4R$, for any x with $0 \leq x < R^2$, it follows that $0 \leq \mathfrak{R}(\mathfrak{R}(x)) < R + 9$. It is easily seen that $R + 9$ can be replaced by $R + 6$. Assuming that all values have more or less the same probability to occur, the result will actually most likely be $< \widetilde{p}$. Although counterexamples are simple to construct and we have no formal proof, we can confidently state the following.

Proposition 1. *For independent random* 128*-bit non-negative integers* x *and* y *there is overwhelming probability that* $0 \leq \mathfrak{R}(\mathfrak{R}(x \cdot y)) < \widetilde{p}$.

Computation of integer multiplication is performed using the data representation described in Section 4.1. In order to take advantage of the multiply-and-add instruction, we use the following property. If $0 \leq a, b, c, d < r$, then $a \cdot b + c + d < r^2$. Specifically, this property enables the addition of a 16-bit word to the result of a 16×16-bit product, used for the multi-precision multiplication and accumulation, and an extra addition of 16-bit word, which is used for carry propagation. The multi-precision products are calculated using the schoolbook method since the modulus is relatively small and the multiply-and-add instruction can be exploited. Our tests show that this approach is faster, for this particular size on this platform, compared to other methods such as Karatsuba multiplication [18].

Modular Subtraction The modular subtraction algorithm, which can be implemented as a subtraction with a conditional addition, is a basic operation. See for implementation details Section 5.1.

Modular Inversion We consider modular inversion of one positive integer x in the residue class of an odd modulus p. Taking into account the memory constrained environment of the PS3s, and the 4-SIMD organization of the SPEs, the most suitable algorithm seems to be the Montgomery algorithm for the classical modular inverse [17]. This algorithm computes modular inversion in two phases:

1. The computation of the almost Montgomery inverse $x^{-1} \cdot 2^k \bmod p$ for some known k.
2. A normalization phase where the factor $2^k \bmod p$ is removed.

In order to exploit the 4-SIMD organization, variables A_1, B_1, A_2 and B_2 are grouped and denoted as $[A_1, B_1, A_2, B_2]$. Then, the resulting 4-SIMD Extended Binary GCD algorithm is depicted in Algorithm 1.

In the algorithm, $A_1 >> t_1$ means that variable A_1 is shifted by t_1 bits towards the least significant bit position. Similarly, $B_1 << t_2$ means that variable B_1 is shifted to the most significant bit position by t_2 positions. Assignments such as $[A_1, B_1, A_2, B_2] := [A_1 >> t_1, B_1 << t_2, A_2 >> t_2, B_2 << t_1]$ mean that the four operations (shift operations in this case) are performed in parallel in SIMD. Note that, in the algorithm, operations $A_1 >> t_1$ and $A_2 >> t_2$ shift out only zero bits.

Algorithm 1 4-SIMD Extended Binary GCD

Input: $p : r^{n-1} < p < r^n$ and $\gcd(p, 2) = 1$
$x : 0 < x < r^n$ and $\gcd(x, p) = 1$
Output: $z \equiv \frac{1}{x} \bmod p$

1: $[A_1, B_1, A_2, B_2] := [p, 0, x, 1]$ and $[k_1, k_2] := [0, 0]$
2: **while** true **do**
3: /* *Start of shift reduction.* */
4: Find t_1 such that $2^{t_1} | A_1$
5: Find t_2 such that $2^{t_2} | A_2$
6: $[k_1, k_2] := [k_1 + t_1, k_2 + t_2]$
7: $[A_1, B_1, A_2, B_2] := [A_1 >> t_1, B_1 << t_2, A_2 >> t_2, B_2 << t_1]$
8:
9: /* *Start of subtraction reduction.* */
10: **if** $(A_1 > A_2)$ **then**
11: $[A_1, B_1, A_2, B_2] := [A_1 - A_2, B_1 - B_2, A_2, B_2]$
12: **else if** $(A_2 > A_1)$ **then**
13: $[A_1, B_1, A_2, B_2] := [A_1, B_1, A_2 - A_1, B_2 - B_1]$
14: **else**
15: **return** $z := B_2 \cdot (2^{-(k_1+k_2)}) \bmod p$
16: **end if**
17: **end while**

Let $g = \gcd(x, p)$ and y be a solution of $xy \equiv g \pmod{p}$. Algorithm 1 has invariants (for $j = 1, 2$)

$$\begin{aligned} A_j(2^{k_1+k_2}y) &\equiv B_j g \pmod{p}, \\ \gcd(A_1, A_2) &= g, \\ A_1B_2 - A_2B_1 &= p, \\ 2^{k_1}A_1 \le p, \quad 2^{k_2}A_2 &\le x, \\ B_1 \le 0 < B_2, \quad k_j \ge 0, \quad A_j &> 0. \end{aligned} \tag{4}$$

At line 15, a modular multiplication removes powers of 2 from the output. We can bound the exponent $k_1 + k_2$ by

$$2^{k_1+k_2} \le (2^{k_1}A_1)(2^{k_2}A_2) \le px.$$

We have $A_1 = A_2 = \gcd(A_1, A_2) = g$. If $A_2 > 1$ then we report an error to the caller. Otherwise $g = 1$. The output $z = B_2 \cdot (2^{-k_1-k_2})$ satisfies

$$z = zg \equiv B_2 \cdot (2^{-k_1-k_2})g \equiv (A_2 2^{k_1+k_2}y)2^{-k_1-k_2} \equiv A_2 y = y \pmod{p}.$$

If we pick t_1 and t_2 as large as possible during a shift reduction, then the new A_1 and A_2 will both be odd. The next subtraction and shift reductions will reduce $A_1 + A_2$ by at least a factor of 2.

The values of A_1 and A_2 are bounded by p and x, respectively. The invariant $p = A_1B_2 - A_2B_1 \ge B_2 - B_1$ bounds B_1 and B_2.

Operation	Estimated #cycles per operation	Quantity per iteration	Estimated #cycles per iteration
Modular multiplication	53	6	318
Modular subtraction	5	6	30
Montgomery reduction	24	1	24
Modular inversion	4941	$\frac{1}{400}$	12
Miscellaneous	69	1	69
Total			453

Table 1. *Estimated number of clock cycles for different operations of our Pollard rho implementation in one SPU.*

4.3 The Distinguished Point and Partition Determination

Each application of an r-adding iteration function requires the determination of the partition to follow for calculating the next point; see Section 3.4. Furthermore, the current unreduced point needs to be inspected for distinguishedness. Since we are performing arithmetic modulo $\widetilde{p}$, the coordinates of the elliptic curve point need to be reduced modulo p, i.e. this point cannot be used to uniquely determine either the partition number or the the distinguished point property. Given a point $\widetilde{P} = (\widetilde{x}, \widetilde{y})$, the idea is to compute a partial Montgomery reduction [24] instead of normalizing $\widetilde{x}$ modulo p which requires a full modular reduction at each iteration. This faster reduction computes $\widetilde{x} \cdot 2^{-16} \bmod p$, where the result of this operation is in $[0, p-1]$. The uniqueness of both the distinguished point property and the partition number is ensured.

5 Experimental Results

In this section, we present the performance analysis of our Pollard rho implementation using the techniques described in Section 4 and show how this implementation has set a record by solving a 112-bit ECDLP. The previous record in the computation of an ECDLP is for an elliptic curve over a 109-bit prime field with parameters taken from Certicom's ECC challenge [4]. That problem was solved in the year 2002 using "10^4 computers (mostly PCs) running 24 hours a day for 549 days" [5].

5.1 Implementation details

Our software implementation is optimized for the SPE-architecture of the Cell and uses all the techniques described in Section 3 with the exception of the negation map. This is because, the computational overhead for this technique, due to the conditional branches required to check for fruitless cycles [13], results (in our implementation on this architecture) in an overall performance degradation. As an iteration function a 16-adding walk is used. In order to take advantage of the Montgomery simultaneous inversion technique, 400 walks are processed

in parallel. The number of concurrent walks is adjusted to the local storage restrictions of the PS3s. At the cost of 16 counters of 32 bits each per process, updating the values c_{i+1}, d_{i+1} can be postponed until a distinguished point is found.

Note that, in our implementation, several things can go wrong: we may have dropped off the curve because we should have used curve doubling (in the unlikely case that $X_i = R_{h(X_i)}$, or in the unlikely case of incorrect reduction modulo $\widetilde{p}$, cf. Prop. 1), or a wrong point may by accident again have landed on the curve, and have nonsensical c_i, d_i values. Just as the correct iterations, these wrong points will after a while end up as distinguished points. Thus, whenever a point is distinguished, we check that it indeed lies on the curve and that the equation $X_i = c_i P + d_i Q$ holds for the alleged c_i and d_i. Only correct distinguished points are collected. If we hit upon a process that has gone off-track, all 400 concurrent processes on that SPU are terminated and restarted, each with a fresh startpoint. This type of error-acceptance leads to enormous timesavings and code-size reduction at negligible cost: we have not found even a single incorrect distinguished point during in the process of solving this ECDLP instance.

A summary of estimated clock cycles needed for each operation is detailed in Table 1. The 69 miscellaneous clock cycles stated in this table include the cost for fetching the constant for the 16-adding walk, checking if a point is distinguished and if so perform sanity checks and the overhead of conditional branches in the main and the simultaneous inversion loop.

Modular Multiplication Our implementation of the modular multiplication method, which is an 128-bit modular multiplication since we work with integers reduced modulo $\widetilde{p}$, as described in Section 4.2, is aimed at filling both the odd and even pipelines, to reduce the overall latency. The 4-SIMD multiplication is done by using the multiply-and-add instruction. Extraction of higher and lower 16-bit parts of a 32-bit word elements is done by using two shuffle operations which are performed in the odd pipeline.

Fast modular reduction, cf. Prop. 1, is implemented using eight multiply-and-add instructions, seven additions, eight extractions of the lower parts and seven extractions of the higher parts for the first reduction phase. For the second reduction, only one multiply-and-add instruction is used since the maximum number that can be added in the second reduction is 4. Most likely no further carries are generated and modular reduction is complete. This condition is checked using an conditional "if" branch with a branch-hint. In the unlikely case that carries are generated, a penalty is paid and the remaining part of the reduction code is executed.

The number of clock cycles needed for a modular multiplication is 53, as shown in Table 1. This number is an average over a long benchmark run using input data from the Pollard rho algorithm.

Modular Subtraction Modular subtraction is performed using operands represented in 4-SIMD with radix 2^{32}. A multi-word subtraction (four extended

subtractions and four generate borrow instructions), comparison (one comparison of the borrow), mask (four AND instructions) and addition (four extended additions and three extended carry generation instructions) are performed in order to avoid expensive branches. Conversions back and forth from representations using radix 2^{16} and 2^{32}, in 4-SIMD, are performed using eight shuffle operation for each conversion.

All in all, 16 instructions in the odd pipeline and 20 instructions in the even pipeline are needed for four modular subtractions. The number of clock cycles required for a single modular subtraction in practice is roughly five (see Table 1).

Modular Inversion The proposed modular inversion algorithm performs one single modular inversion using the SIMD instructions of the SPE, with either two or four active computations at a time. The 128-bit values of A_1, B_1, A_2, and B_2 are stored using the data representation described in Section 4.1 with $w = 32$. The initializations $A_1 = p$, $B_1 = 0$, $B_2 = 1$ do not depend on x. The initialization of $A_2 = x$ requires eight load and four shuffle instructions to convert the input.

A shift reduction always starts with at least one of A_1 and A_2 being odd, by Equation (4). We do not know which of these might be even, but can examine both, in a SIMD fashion. The trailing zero bit count of a positive integer A is the population count of $\overline{A} \wedge (A - 1)$. The PS3's population count instruction acts only on 8-bit data, so our t_1 and t_2 may not be maximal. The PS3 lets each vector element have its own shift or rotate count, although a single instruction cannot rotate some elements while others shift.

Within the subtraction reduction, the four 128-bit differences $A_1 - A_2$, $B_1 - B_2$, $A_2 - A_1$, and $B_2 - B_1$ are evaluated in parallel. We exit the loop if neither $A_1 - A_2$ nor $A_2 - A_1$ needs a borrow. Otherwise we update $[A_1, B_1, A_2, B_2]$ appropriately. Subtracting 1 from each element of the borrow vector gives masks of -1 or 0 depending on the sign of $A_1 - A_2$ or $A_2 - A_1$. A shuffle of these masks builds a selector which determines which parts of $[A_1, B_1, A_2, B_2]$ are updated.

The final multiplication with 2^{-k} is done by first looking up this value in a table and next computing the modular multiplication as outlined in Prop. 1. Hence, the modular inversion implementation takes as input an 128-bit integer x and outputs an 128-bit integer $z \equiv \frac{1}{x} \bmod p$ with $0 \leq x, z < \widetilde{p}$.

5.2 Performance Comparison

In the paper by Güneysu et al. [14] a field-programmable gate array (FPGA)-based multi-processing hardware architecture for the Pollard rho method targeted at elliptic curves over prime fields is described. Performance details are stated for a hardware implementation using XC3S1000 FPGAs targeted at field sizes of varying bit lengths.

Our PS3 implementation is targeted at an elliptic curve over a 112-bit prime field. We use 128-bit multiplication with fast reduction modulo the 128-bit special modulus $\widetilde{p}$. The inversion of 128-bit values is performed modulo the 112-bit

prime. Experimental results show that modular multiplication using fast reduction (see Prop. 1) is roughly 20 percent faster compared to an implementation of Montgomery multiplication on the SPE architecture. Because 128-bit reduction is used, we compare our performance results to the FPGA-results of elliptic curves over 96- and 128-bit generic prime fields [14]. These results are given for the cost-efficient parallel architecture called COPACABANA [21]. This architecture can host up to 120 FPGAs at a total cost of approximately US$ $10,000$ including material and production costs. Using this setup, a performance of $3.97 \cdot 10^7$ and $2.08 \cdot 10^7$ iterations per second can be achieved for the 96- and 128-bit versions respectively.

The current price for a PS3, as stated on large web-stores, is around US$ 300. Hence, for the price of one COPACABANA, 33 PS3s can be purchased. The resulting cluster of PS3s is able to compute $1.4 \cdot 10^9$ iterations per second. The performance results reported in [14] are dated from 2007. We scale the performance results according to Moore's law [26], i.e. the performance is doubled. The implementations by Güneysu et al. use the negation map optimization, leading to a $\sqrt{2}$ speed-up. The use of this technique results in some overhead related to the detection and handling of fruitless cycles; the reason why we decided not to use this technique in our SPE-implementation. Unfortunately, no details are given related to this overhead. After scaling the COPACABANA performance numbers by a factor two, due to Moore's law, and assuming that the negation map optimization technique is used, leading to a speed-up of a factor $\sqrt{2}$, the PS3 cluster outperforms the COPACABANA machine by a factor 12.4 and 23.8 compared to the 96- and 128-bit versions respectively.

5.3 Solving a 112-bit ECDLP

We solved the ECDLP using the parameters of the standardized curve over a 112-bit prime field using the methods and implementation as explained in this article. The expected number of iterations is $\sqrt{\frac{\pi \cdot n}{2}} \approx 8.4 \cdot 10^{16}$, where n is the prime order of the base point P as specified in the standard, assuming the negation map optimization is not used. The real number of required iterations to solve this ECDLP was only two percent higher. The calculation has been performed on a PS3 cluster of more than 200 PS3s and started on January 13, 2009 and finished on July 8, 2009. It ran on and off, occasionally interrupted by other cryptanalytic projects. When run continuously using the latest version of our code, the same calculation would have taken 3.5 months.

By selecting a DTP property with occurrence of approximately once every 2^{24} points, we needed to store $\approx 5.0 \cdot 10^9$ DTPs. Storage of a DTP $X = (x, y)$ together with the values c and d such that $X = cP + dQ$, requires $4 \cdot 112$ bits when storing in an uncompressed format. Hence, the total required storage sums up to $4 \cdot 112 \cdot 5.0 \cdot 10^9$ bits ≈ 260 gigabyte. To facilitate collision finding using standard unix commands the DTPs were stored in plaintext format increasing the required storage to 615 gigabyte.

We solved the discrete logarithm with respect to P for the point Q. The point P of order n are given in the standard and the x-coordinate of Q was chosen as $\lfloor(\pi - 3)10^{34}\rfloor$. The points P, Q and the solution to $Q = lP$ are given here:

$P =$ (188281465057972534892223778713752, 341987549103317082716786189608268 8)
$Q =$ (141592653589793238462643383279502 8, 384675960649470672428613962388554 4)
$n =$ 44516852250937147764918915425489 33
$Q =$ 31252163601477247716176735185669 9 $\cdot P$

6 Conclusions

We have presented a high-performance PlayStation 3 (PS3) implementation of the Pollard rho discrete logarithm algorithm on elliptic curves over prime fields. Arithmetic algorithms have been designed for the SIMD-like architectures such as the PS3. Using this implementation a record has been set by solving a 112-bit ECDLP where the parameters are taken from a standardized curve. The time required to solve this ECDLP instance is 62.6 PS3 years. This shows that given the easy accessibility and the relatively low price of these game consoles, solving ECDLPs for this bit-size is practical.

References

1. D. J. Bernstein and T. Lange. Analysis and optimization of elliptic-curve single-scalar multiplication. In *Finite Fields and Applications*, volume 461 of *Contemporary Mathematics Series*, pages 1–19, 2008.
2. J. W. Bos, N. Casati, and D. A. Osvik. Multi-stream hashing on the PlayStation 3. *PARA 2008*, 2008. To appear.
3. J. W. Bos and M. E. Kaihara. Montgomery multiplication on the Cell. *PPAM 2009*, 2009. To appear.
4. Certicom. Certicom ECC Challenge. See http://www.certicom.com/images/pdfs/cert_ecc_challenge.pdf, 1997.
5. Certicom. Press release: Certicom announces elliptic curve cryptosystem (ECC) challenge winner. See http://www.certicom.com/index.php/2002-press-releases/38-2002-press-releases/340-notre-dame-mathematician-solves-eccp-109-encryption-key-problem-issued-in-1997, 2002.
6. Certicom Research. Standards for Efficient Cryptography 2: Recommended Elliptic Curve Domain Parameters. Standard SEC2, Certicom, 2000.
7. N. Costigan and P. Schwabe. Fast elliptic-curve cryptography on the Cell broadband engine. In *Africacrypt 2009*, volume 5580 of *LNCS*, pages 368–385, 2009.
8. N. Costigan and M. Scott. Accelerating SSL using the vector processors in IBM's Cell broadband engine for Sony's Playstation 3. Cryptology ePrint Archive, Report 2007/061, 2007. http://eprint.iacr.org/.
9. B. Dixon and A. K. Lenstra. Fast massively parallel modular arithmetic. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 99–110, 1993.
10. I. M. Duursma, P. Gaudry, and F. Morain. Speeding up the discrete log computation on curves with automorphisms. In *Asiacrypt 1999*, volume 1716 of *LNCS*, pages 103–121, 1999.

11. B. Flachs, S. Asano, S. Dhong, P. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H. Oh, S. M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, and N. Yano. A streaming processor unit for a Cell processor. *IEEE International Solid-State Circuits Conference*, pages 134–135, February 2005.
12. W. A. P. Forum. Wireless transport layer security specification. See http://www.openmobilealliance.org/tech/affiliates/wap/wap-261-wtls-20010406-a.pdf, 2001.
13. R. P. Gallant, R. J. Lambert, and S. A. Vanstone. Improving the parallelized Pollard lambda search on anomalous binary curves. *Mathematics of Computation*, 69(232):1699–1705, 2000.
14. T. Güneysu, C. Paar, and J. Pelzl. Special-purpose hardware for solving the elliptic curve discrete logarithm problem. *ACM Transactions on Reconfigurable Technology and Systems*, 1(2):1–21, 2008.
15. B. Harris. Probability distributions related to random mappings. *The Annals of Mathematical Statistics*, 31:1045–1062, 1960.
16. H. P. Hofstee. Power efficient processor architecture and the Cell processor. In *HPCA 2005*, pages 258–262, 2005.
17. B. S. Kaliski Jr. The Montgomery inverse and its applications. *IEEE Transactions on Computers*, 44(8):1064–1065, 1995.
18. A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. Number 145 in Proceedings of the USSR Academy of Science, pages 293–294, 1962.
19. D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, third edition, 1997.
20. N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.
21. S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler. Breaking ciphers with COPACOBANA - a cost-optimized parallel code breaker. In *CHES 2006*, volume 4249 of *LNCS*, pages 101–118, 2006.
22. A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, 2001.
23. V. S. Miller. Use of elliptic curves in cryptography. In *Crypto 1985*, volume 218 of *LNCS*, pages 417–426, 1986.
24. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.
25. P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48:243–264, 1987.
26. G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38:8, 1965.
27. S. C. Pohlig and M. E. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Transactions on Information Theory*, 24:106–110, 1978.
28. J. M. Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of Computation*, 32:918–924, 1978.
29. J.-J. Quisquater and J.-P. Delescaille. How easy is collision search. new results and applications to DES. In *Crypto 1989*, volume 435 of *LNCS*, pages 408–413, 1989.
30. R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.
31. C. P. Schnorr and H. W. Lenstra, Jr. A Monte Carlo factoring algorithm with linear storage. *Mathematics of Computation*, 43(167):289–311, 1984.

32. D. Shanks. Class number, a theory of factorization, and genera. In *Symposia in Pure Mathematics*, volume 20, pages 415–440, 1971.
33. J. H. Silverman. *The Arithmetic of Elliptic Curves*, volume 106 of *Gradute Texts in Mathematics*. Springer-Verlag, 1986.
34. M. Stevens, A. K. Lenstra, and B. de Weger. Predicting the winner of the 2008 US presidential elections using a Sony PlayStation 3. http://www.win.tue.nl/hashclash/Nostradamus/.
35. M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, D. Molnar, D. A. Osvik, and B. de Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In *Crypto 2009*, volume 5677 of *LNCS*, pages 55–69, 2009.
36. O. Takahashi, R. Cook, S. Cottier, S. H. Dhong, B. Flachs, K. Hirairi, A. Kawasumi, H. Murakami, H. Noro, H. Oh, S. Onish, J. Pille, and J. Silberman. The circuit design of the synergistic processor element of a Cell processor. In *ICCAD 2005*, pages 111–117. IEEE Computer Society, 2005.
37. E. Teske. On random walks for Pollard's rho method. *Mathematics of Computation*, 70(234):809–825, 2001.
38. U.S. Department of Commerce/National Institute of Standards and Technology. Digital Signature Standards (DSS). FIPS-186-2, Certicom Corp., 2000. See http://csrc.nist.gov/publications/PubsFIPS.html.
39. P. C. van Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.
40. M. J. Wiener and R. J. Zuccherato. Faster attacks on elliptic curve cryptosystems. In *Selected Areas in Cryptography*, volume 1556 of *LNCS*, pages 190–200, 1998.

The Certicom Challenges ECC2-X

Daniel V. Bailey[1], Brian Baldwin[2], Lejla Batina[3], Daniel J. Bernstein[4], Peter Birkner[5], Joppe W. Bos[6], Gauthier van Damme[3], Giacomo de Meulenaer[7], Junfeng Fan[3], Tim Güneysu[8], Frank Gurkaynak[9], Thorsten Kleinjung[6], Tanja Lange[5], Nele Mentens[3], Christof Paar[8], Francesco Regazzoni[7], Peter Schwabe[5], and Leif Uhsadel[3] ⋆

[1] RSA, the Security Division of EMC, USA
dbailey@rsa.com
[2] Claude Shannon Institute for Discrete Mathematics, Coding and Cryptography.
Dept. of Electrical & Electronic Engineering, University College Cork, Ireland
brianb@rennes.ucc.ie
[3] ESAT/SCD-COSIC, Katholieke Universiteit Leuven and IBBT
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
[4] Department of Computer Science
University of Illinois at Chicago, Chicago, IL 60607–7045, USA
djb@cr.yp.to
[5] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, Netherlands
p.birkner@tue.nl, tanja@hyperelliptic.org, peter@cryptojedi.org
[6] EPFL IC IIF LACAL, Station 14, CH-1015 Lausanne, Switzerland
{joppe.bos, thorsten.kleinjung}@epfl.ch
[7] UCL Crypto Group, Place du Levant, 3, B-1348 Louvain-la-Neuve, Belgium
{giacomo.demeulenaer, francesco.regazzoni}@uclouvain.be
[8] Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
{gueneysu, cpaar}@crypto.rub.de
[9] Microelectronics Design Center, ETH Zürich, Switzerland
kgf@ee.ethz.ch

Abstract. To encourage research on the hardness of the elliptic-curve discrete-logarithm problem (ECDLP) Certicom has published a series of challenge curves and DLPs.
This paper analyzes the costs of breaking the Certicom challenges over the binary fields $\mathbf{F}_{2^{131}}$ and $\mathbf{F}_{2^{163}}$ on a variety of platforms. We describe details of the choice of step function and distinguished points for the Koblitz and non-Koblitz curves. In contrast to the implementations for the previous Certicom challenges we do not restrict ourselves to software and conventional PCs, but branch out to cover the majority of available platforms such as various ASICs, FPGAs, CPUs and the Cell Broadband Engine. For the field arithmetic we investigate polynomial and normal basis arithmetic for these specific fields; in particular for the challenges on Koblitz curves normal bases become more attractive on ASICs and

⋆ This work has been supported in part by the National Science Foundation under grant ITR-0716498 and in part by the European Commission through the ICT Programme under Contract ICT–2007–216676 ECRYPT II.

FPGAs.

Keywords: ECC, binary fields, Certicom challenges

1 Introduction

In 1997, Certicom published several challenges [Cer97a] to solve the Discrete Logarithm Problem (DLP) on elliptic curves. The challenges cover curves over prime fields and binary fields at several different sizes. For the binary curves, each field size has two challenges: a Koblitz curve and a random curve defined over the full field.

For small bit sizes the challenges were broken quickly — the 79-bit challenges fell already in 1997, those with 89 bits in 1998 and those with 97 bits in 1998 and 1999; Certicom described these parameter sizes as training exercises. In April 2000, the first Level I challenge (the Koblitz curve ECC2K-108) was solved by Harley's team in a distributed effort [Har] on a multitude of PCs on the Internet. After that, it took some time until the remaining challenges with 109 bit fields were tackled. The ECCp-109 (elliptic curve over a prime field of 109 bits) was solved on November 2002 and the ECC2-109 (random elliptic curve over a binary field with 109 bits) was solved in April 2004; both efforts were organized by Chris Monico. The gap of more than one year between the results is mostly due to the Koblitz curves offering less security per bit than curves defined over the extension field or over prime fields. The Frobenius endomorphism can be used to speed up the protocols using elliptic curves — the main reason Koblitz curves are attractive in practice — but it also gives an advantage to the attacker. In particular, over $\mathbf{F}_{2^n}$ the attack is sped up by a factor of approximately $\sqrt{n}$.

Since 2004 not much was heard about attempts to break the larger challenges. Certicom's documentation states "The 109-bit Level I challenges are feasible using a very large network of computers. The 131-bit Level I challenges are expected to be infeasible against realistic software and hardware attacks, unless of course, a new algorithm for the ECDLP is discovered. The Level II challenges are infeasible given today's computer technology and knowledge."

In this paper we analyze the cost of breaking the binary Certicom challenges: ECC2K-130, ECC2-131, ECC2K-163 and ECC2-163. We collect timings for field arithmetic in polynomial and normal basis representation for several different platforms which the authors of this paper have at their disposal and outline the ways of computing discrete logarithms on these curves. For Koblitz curves, the Frobenius endomorphism can be used to speed up the attack by working with classes of points. The step function in Pollard's rho method has to be adjusted to deal with classes. Per step a few more squarings are needed but the overall savings in the number of steps is quite dramatic.

The SHARCS community has already had some analysis of the costs of breaking binary ECC on FPGAs at SHARCS'06 [BMdDQ06]. Our analysis is an update of those results for current FPGAs and covers the concrete challenges. Particular emphasis is placed on the FPGA used in the Copacobana FPGA cluster. This way, part of the attack can be run on this cluster. Our research goes

further than [BMdDQ06] by dealing with other curves, considering many other platforms and analyzing the best methods for how these platforms can work together on computing discrete logarithms.

Our main conclusion is that the "infeasible" ECC2K-130 challenge is in fact feasible. For example, our implementations can break ECC2K-130 in an expected time of a year using only 4200 Cell processors, or using only 220 ASICs. For comparison, [BMdDQ06] and [MdDBQ07] estimated a cost of nearly $20000000 to break ECC2K-130 in a year.

As validation of the designs and the performance estimates we reimplemented and reran the ECC2K-95 challenge, using 30 2.4GHz cores on a Core 2 cluster for a few days to re-break the ECC2K-95 challenge. Each core performed 4.7 million iterations per second and produced distinguished points at the predicted speed. For comparison, Harley's original ECC2K-95 solution took 25 days on 200 Alpha workstations, the fastest being 0.6GHz cores performing 0.177 million iterations per second. The improvement is due not only to increased processor speeds but also to improved implementation techniques described in this paper.

The project partners are working on collecting enough hardware to actually carry out the ECC2K-130 attack. Available resources include KU Leuven's VIC cluster (`https://vscentrum.be/vsc-help-center/reference-manuals/vic-user-manual` and `vic3-user-manual`); several smaller clusters such as TU Eindhoven's CCCC cluster (`http://www.win.tue.nl/cccc/`); several high-end GPUs (not yet covered in this paper); some FPGA clusters; and possibly some ASICs. This is the first time that one of the Certicom challenges is tackled with a broad mix of platforms. This set-up requires extra considerations for the choice of the step function and the distinguished points so that all platforms can cooperate in finding collisions despite different preferences in point representation.

2 The Certicom challenges

Each challenge is to compute the ECC private key from a given ECC public key, i.e. to solve the discrete-logarithm problem in the group of points of an elliptic curve over a field $\mathbf{F}_{2^n}$. The complete list of curves is published online at [Cer97b]. In the present paper, we tackle the curves ECC2K-130, ECC2-131, ECC2K-163, and ECC2-163, the parameters of which are given below.

The parameters are to be interpreted as follows: The curve is defined over the finite field represented by $\mathbf{F}_2[z]/F(z)$, where $F(z)$ is the monic irreducible polynomial of degree n given below for each challenge. Field elements are given as hexadecimal numbers which are interpreted as bit strings giving the coefficients of polynomials over $\mathbf{F}_2$ of degree less than n. The curves are of the form $y^2+xy = x^3 + ax^2 + b$, with $a, b \in \mathbf{F}_{2^n}$. For the Koblitz curve challenges the curves are defined over $\mathbf{F}_2$, i.e. $a, b \in \mathbf{F}_2$. The points P and Q are given by their coordinates $P = (\texttt{P\_x}, \texttt{P\_y})$ and $Q = (\texttt{Q\_x}, \texttt{Q\_y})$. The group order is $h \cdot \ell$, where ℓ is a prime and h is the cofactor.

– ECC2K-130 ($F = z^{131} + z^{13} + z^2 + z + 1$)

```
a = 0, b = 1
P_x = 05 1C99BFA6 F18DE467 C80C23B9 8C7994AA
P_y = 04 2EA2D112 ECEC71FC F7E000D7 EFC978BD
h = 04, l = 2 00000000 00000000 4D4FDD57 03A3F269
Q_x = 06 C997F3E7 F2C66A4A 5D2FDA13 756A37B1
Q_y = 04 A38D1182 9D32D347 BD0C0F58 4D546E9A
```

– ECC2-131 ($F = z^{131} + z^{13} + z^2 + z + 1$)

```
a = 07 EBCB7EEC C296A1C4 A1A14F2C 9E44352E
b = 00 610B0A57 C73649AD 0093BDD6 22A61D81
P_x = 00 439CBC8D C73AA981 030D5BC5 7B331663
P_y = 01 4904C07D 4F25A16C 2DE036D6 0B762BD4
h = 02, l = 04 00000000 00000002 6ABB991F E311FE83
Q_x = 06 02339C5D B0E9C694 AC890852 8C51C440
Q_y = 04 F7B99169 FA1A0F27 37813742 B1588CB8
```

– ECC2K-163 ($F = z^{163} + z^8 + z^2 + z + 1$)

```
a = 1, b = 1
P_x = 02 091945E4 2080CD9C BCF14A71 07D8BC55 CDD65EA9
P_y = 06 33156938 33774294 A39CF6F8 C175D02B 8E6A5587
h = 02, l = 04 00000000 00000000 00020108 A2E0CC0D 99F8A5EF
Q_x = 00 7530EE86 4EDCF4A3 1C85AA17 C197FFF5 CAFECAE1
Q_y = 07 5DB1E80D 7C4A92C7 BBB79EAE 3EC545F8 A31CFA6B
```

– ECC2-163 ($F = z^{163} + z^8 + z^2 + z + 1$)

```
a = 02 5C4BEAC8 074B8C2D 9DF63AF9 1263EB82 29B3C967
b = 00 C9517D06 D5240D3C FF38C74B 20B6CD4D 6F9DD4D9
P_x = 02 3A2E9990 4996E867 9B50FF1E 49ADD8BD 2388F387
P_y = 05 FCBFE409 8477C9D1 87EA1CF6 15C7E915 29E73BA2
h = 02, l = 04 00000000 00000000 0001E60F C8821CC7 4DAEAFC1
Q_x = 04 38D8B382 1C8E9264 637F2FC7 4F8007B2 1210F0F2
Q_y = 07 3FCEA8D5 E247CE36 7368F006 EBD5B32F DF4286D2
```

The curves denoted ECC2K-X are binary Koblitz curves. This means that their equation is defined over $\mathbf{F}_2$ which, in turn, implies that the Frobenius endomorphism σ operates on the set of points over $\mathbf{F}_{2^n}$. Because σ commutes with scalar multiplication, it operates on prime-order subgroups as a group automorphism. Consequently, there exists an integer s which is unique modulo ℓ so that $\sigma(P) = [s]P$ for all points P in the subgroup of order ℓ. The value of s is computed as $T - s = \gcd(T^n - 1, T^2 + (-1)^a T + 2)$ in $\mathbf{F}_\ell[T]$.

3 Parallelized Pollard's rho algorithm

In this section, we explain the parallelized version of Pollard's rho method. First, we describe the single-instance version of the method and then show how to parallelize it with the distinguished-point method as done by van Oorschot and Wiener in [vOW99]. Note that these descriptions give the "school-book versions" for background; details on our actual implementation are given in Section 6.

3.1 Single-instance Pollard rho

Pollard's rho method is an algorithm to compute the discrete logarithm in generic cyclic groups. It was originally designed for finding discrete logarithms in $\mathbf{F}_p^*$ [Pol78] and is based on Floyd's cycle-finding algorithm and the birthday paradox. In the following, let G be a cyclic group, using additive notation, of order ℓ with a generator P. Given $Q \in G$, our goal is to find an integer k such that $[k]P = Q$.

The idea of Pollard's rho method is to construct a sequence of group elements with the help of an iteration function $f : G \to G$. This function generates a sequence

$$P_{i+1} = f(P_i),$$

for $i \geq 0$ and some initial point P_0. We compute elements of this sequence until a collision of two elements occurs. A collision is an equality $P_q = P_m$ with $q \neq m$. Let us assume we know how to write the elements P_i of the sequence as $P_i = [a_i]P \oplus [b_i]Q$, then we can compute the discrete logarithm $k = \log_P(Q) = \frac{a_q - a_m}{b_m - b_q}$ if a collision $P_q = P_m$ with $b_m \neq b_q$ has occurred. We show later how to obtain such sequences.

Assuming the iteration function is a random mapping of size ℓ, i.e. f is equally probable among all functions $G \to G$, Harris in [Har60] showed that the expected number of steps before a collision occurs is approximately $\sqrt{\pi\ell/2}$. The sequence $(P_i)_{i\geq 0}$ is called a *random walk* in G.

A pictorial description of the rho method can be given by drawing the Greek letter ρ representing the random walk and starting at the tail at P_0. "Walking" along the line means going from P_i to P_{i+1}. If a collision occurs at P_t, then $P_t = P_{t+s}$ for some integer s, and the elements $P_t, P_{t+1}, \ldots, P_{t+s-1}$ form a loop. See Figure 1, and see Figure 2 for an example of how this picture occurs inside the complete graph of a function.

In the original paper by Pollard it is proposed to find a collision with Floyd's cycle-finding algorithm. The idea of this algorithm is to walk along the sequence at two different speeds and wait for a collision. This is usually realized by using the two sequences P_i and P_{2i}. Doing a step means to increase i by 1. If $P_i = P_{2i}$ for some i, then we have found a collision.

To benefit from this method it is necessary to construct walks on G that behave like random mappings and for which for each element a representation as $P_i = [a_i]P \oplus [b_i]Q$ is known. An example of a such a class of walks is the r-adding walk as studied by Teske [Tes01]. The group G is divided in r partitions using a partition function $\psi : G \to [0, r-1]$. An element $R_j = [c_j]P \oplus [d_j]Q$, for

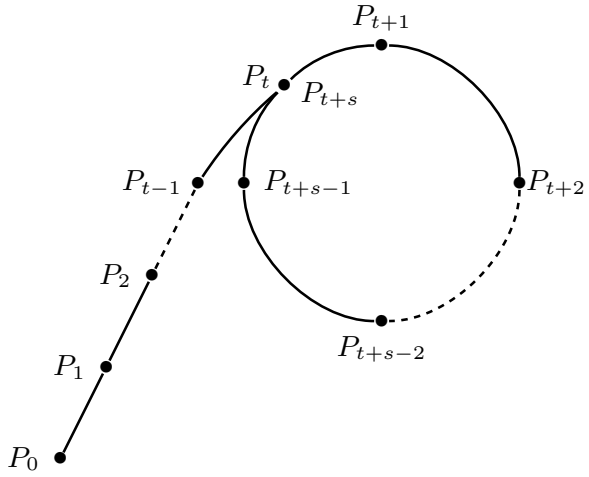


Fig. 1. Abstract diagram of the rho method.

random c_j and d_j, is associated to each partition and the iteration function is defined as

$$P_{i+1} = f(P_i) = P_i \oplus R_{\psi(P_i)},$$

and the values of a_i and b_i are updated as $a_{i+1} = a_i + c_j,\ b_{i+1} = b_i + d_j$. When a collision $P_q = P_m$ for $q \neq m$ is found, then we obtain

$$[a_q]P \oplus [b_q]Q = [a_m]P \oplus [b_m]Q,$$

which implies $(a_q - a_m)P = (b_m - b_q)Q$ and hence $k = \log_P(Q) = \frac{a_q - a_m}{b_m - b_q}$. This solves the discrete-logarithm problem. With negligible probability the difference $b_m - b_q = 0$; in this case the computation has to be restarted with a different starting point P_0. Teske showed that choosing $r = 20$ and random values for the c_j and d_j approximates a random walk sufficiently well for the purpose of analyzing the function. For implementations, a power of 2 such as $r = 8$ or $r = 16$ or $r = 32$ is more practical.

3.2 Parallelized version and distinguished points

When running N instances of Pollard's rho method concurrently, a speed-up of $\sqrt{N}$ is obtained. To get a linear speedup, i.e. by a factor N, and thus to parallelize Pollard's rho method efficiently, van Oorschot and Wiener [vOW99] proposed the distinguished-points method. It works as follows: One defines a subset D of G such that D consists of all elements that satisfy a particular condition. For example, we can choose D to contain all group elements whose s least significant bits are zero, for some positive integer s. This allows to easily check if a group element is in D or not.

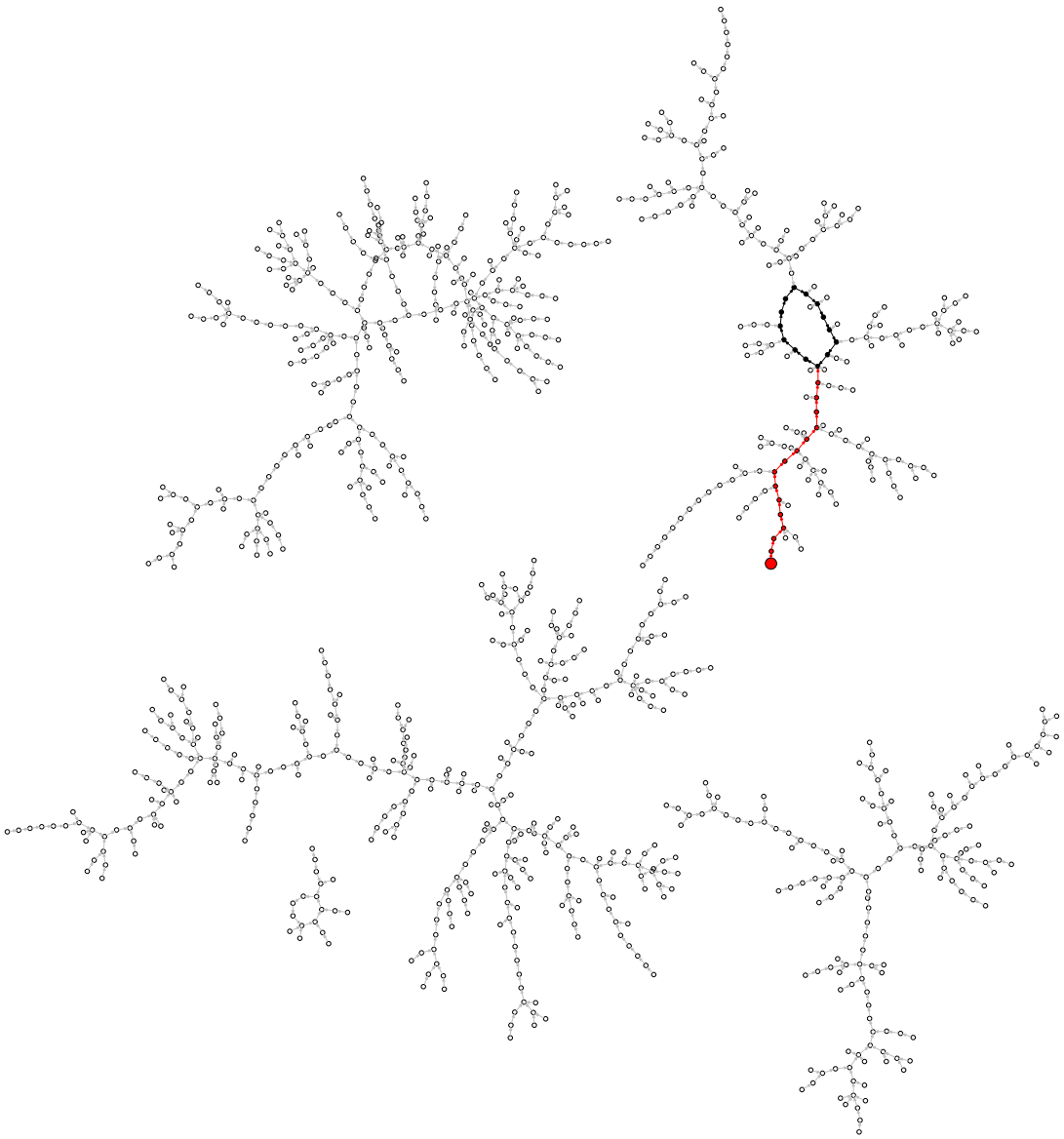


Fig. 2. Example of the rho method. There are 1024 circles representing elements of a set of size 1024. Each circle has an arrow to a randomly chosen element of the set; the positions of circles in the plane are chosen (by the `neato` program) so that these arrows are short. A walk begins at a random point indicated by a large red circle. The walk stops when it begins to cycle; the cycle is shown in black. This walk involves 29 elements of the set.

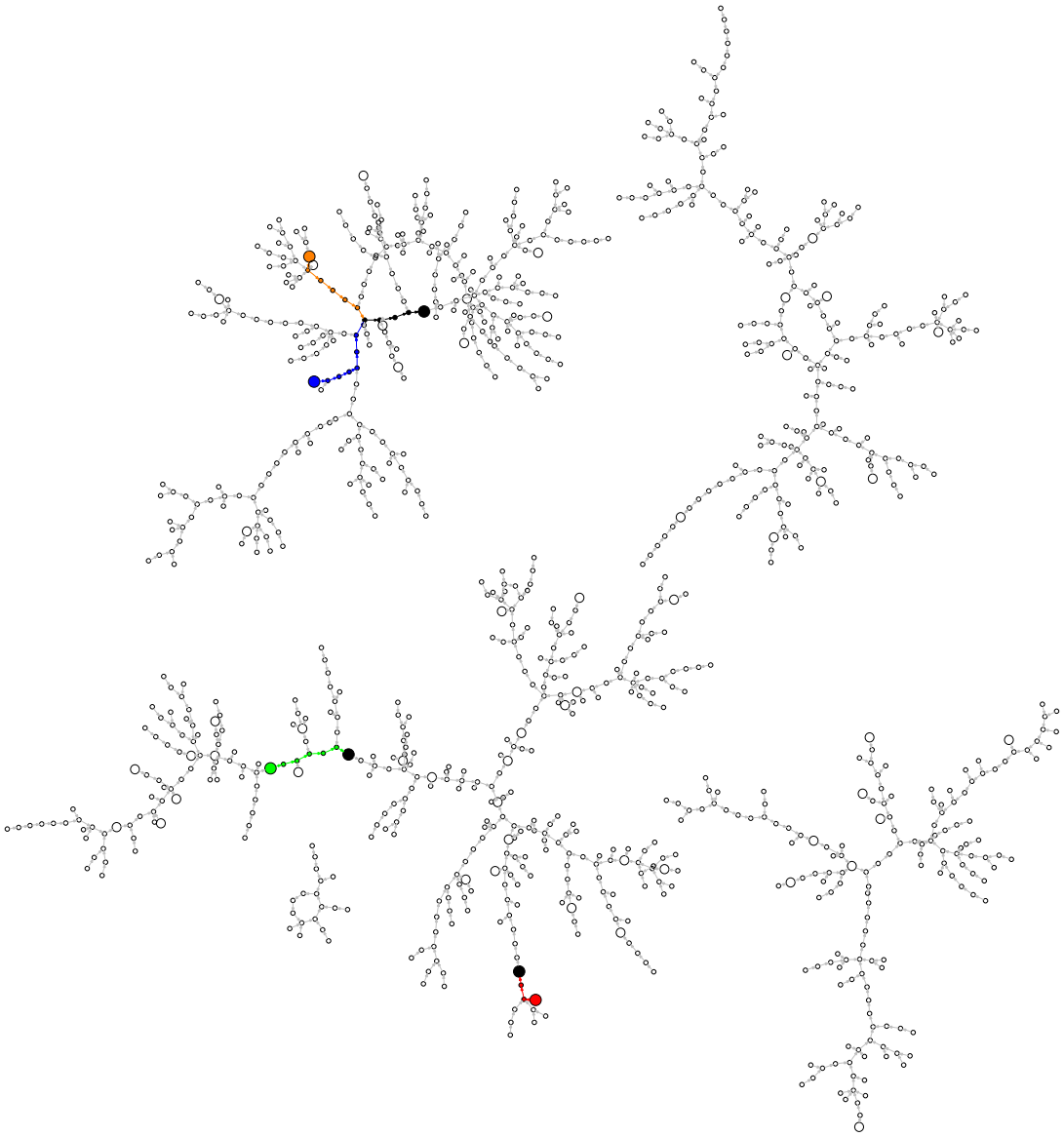


Fig. 3. Example of the parallelized rho method. Arrows represent the same randomly generated function used in Figure 2. Each circle is, with probability 1/16, designated a distinguished point and drawn as a large hollow circle. Four walks begin at random points indicated by large red, green, blue, and orange circles. Each walk stops when it reaches a distinguished point; those distinguished points are shown in black. The blue and orange walks collide and find the same distinguished point.

Each instance starts at a new linear combination $[a_0]P \oplus [b_0]Q$ and performs the random walk until a distinguished point is found. The distinguished point, together with the coefficients a_i, b_i that lead to it, is then stored on a central server. The server checks for collisions within the set of distinguished points and can solve the DLP once one collision is found. As before, the difference $b_m - b_q$ can be zero. In this case the distinguished point P_q is discarded and the search continues; note that the other stored distinguished points can lead to collisions since all processes are started independently at random.

See Figure 3 for an illustration of the distinguished-points method.

4 Automorphisms of small order

Elliptic-curve groups allow very fast negation. On binary curves such as the Certicom challenge curves, the negative of $P = (x, y)$ is $-P = (x, y + x)$. One can speed up the rho method by a factor of $\sqrt{2}$ (cf. [WZ98]) by choosing an iteration function defined on sets $\{P, -P\}$. For example, one can take any iteration function g, and define $f(x, y)$ as $g(\min\{(x, y), (x, y + x)\})$, ensuring that $f(-P) = f(P)$. Here min means lexicographic minimum. Special care has to be taken to avoid fruitless short cycles; see method 1 below for details.

The ECC2-131 challenge has group size $n \approx 2^{130}$; up to negation there are $n/2 \approx 2^{129}$ sets $\{P, -P\}$. In this case the expected number of steps is approximately $\sqrt{\pi n/4} \approx 2^{64.8}$. The ECC2-163 challenge has group size $n \approx 2^{162}$; in this case $\sqrt{\pi n/4} \approx 2^{80.8}$.

The ECC2K-130 challenge has group size $n \approx 2^{129}$. The DLP is easier than for the ECC2-131 challenge because this curve is a Koblitz curve allowing a very fast Frobenius endomorphism. Specifically, if (x, y) is on the ECC2K-130 curve then $\sigma(x, y) = (x^2, y^2), \sigma^4(x, y) = (x^4, y^4)$ and so on through $(x^{2^{130}}, y^{2^{130}})$ are on the ECC2K-130 curve; note that $(x^{2^{131}}, y^{2^{131}}) = (x, y)$. One can speed up the rho method by an additional factor of $\sqrt{131}$ by choosing an iteration function defined on sets $\{\pm\sigma^i(x, y) | 0 \leq i < n\}$; there are only about $n/262 \approx 2^{121}$ of these sets. In this case the expected number of rho iterations is approximately $\sqrt{\pi n/524} \approx 2^{60.8}$. Similarly, the expected number of iterations for the ECC2K-163 challenge is approximately $2^{77.2}$.

The remainder of this section focuses on Koblitz curves, i.e. curves defined over $\mathbf{F}_2$ that are considered over extension fields $\mathbf{F}_{2^n}$ and considers how to define walks on classes under the Frobenius endomorphism.

Speedups for solving the DLP on elliptic curves with automorphisms were studied by several groups independently. The following text summarizes the contributions by Wiener and Zuccherato [WZ98], Gallant, Lambert, and Vanstone [GLV00], and Harley [Har]. In the following we describe two methods to define a (pseudo-)random walk on the classes of points, where the class of a point P also contains $-P, \pm\sigma(P), \pm\sigma^2(P), \ldots, \pm\sigma^{n-1}(P)$.

4.1 Method 1

This method was discovered by Wiener and Zuccherato [WZ98] and Gallant, Lambert, and Vanstone [GLV00]. To apply the parallel Pollard rho method, the iteration function (or step function) in the first method uses a r-adding walk (see [Tes01]), i.e. we have r pre-defined random points $R_0, \ldots, R_{r-1}$ on the curve. To perform a step of the walk, we add one of the R_j's to the current point P_i to obtain P_{i+1}. To determine which point to add to P_i, we partition the group of points on the curve into r sets and define the map

$$\psi : E(\mathbf{F}_{2^n}) \mapsto \{0, \ldots, r-1\}, \tag{1}$$

which assigns to each point on the curve one of the r partitions. With this map, we could define the walk and the iteration function f as follows:

$$P_{i+1} = f(P_i) = P_i \oplus R_{\psi(P_i)}. \tag{2}$$

However, more care is required to avoid fruitless, short cycles. Assume that P_i is such that the unique representative of the class of $P_{i+1} = P_i \oplus R_{\psi(P_i)}$ is $-P_{i+1}$; this happens with probability $1/(2n)$. With probability $1/r$ the next added point $R_{\psi(-P_{i+1})}$ equals $R_{\psi(P_i)}$ and thus $P_{i+2} = P_i$ and the walk will never leave this cycle. The probability of such cycles is $1/(2rn)$ and is thereby rather high. See [WZ98] for a method to detect and deal with such cycles.

We define the unique representative per class by lexicographically ordering all x-coordinates of the points in the class and choosing the "smallest" element in that order. This element has most zeros starting from the most significant bit. Of the two possible y-coordinates chose the one with lexicographically smaller value. Given that y and $y + x$ are the candidate values and that the number of leading zeros of x is known already, say i, it is easy to grab the distinguishing bit in the y-coordinate as the $(i+1)$-th bit starting from the most significant bit. Let $\Phi(P)$ be this unique representative of the class containing P and let $(b_4, b_3, b_2, b_1, 1)$ be the five least significant bits of $\Phi(P)$. Let $j = (b_4, b_3, b_2, b_1)_2$. Then we can define the value of $\psi(P)$ to be $j \in \{0, \ldots, 15\}$. The iteration function is then given by

$$P_{i+1} = f(P_i) = \Phi(P_i) \oplus R_{\psi(P_i)}.$$

To parallelize Pollard rho, we also need to define distinguished points (or rather distinguished classes in the present case). For each class, we use the m most significant bits of $x(\Phi(P))$. If these bits are all zero, then we define this point (class) to be a distinguished one. With this, we can apply the methods from Section 3.

4.2 Method 2

A method similar to the second method was described by Gallant, Lambert, and Vanstone [GLV00]; Harley [Har] uses a similar method in his code to attack the earlier Certicom challenges.

This method does not need any precomputed random points R_i on the curve. Instead, we define the walk and the iteration function f as

$$P_{i+1} = f(P_i) = P_i \oplus \sigma^j(P_i), \tag{3}$$

where j is the Hamming weight of the binary representation of $x(P_i)$ in normal basis representation and σ the Frobenius automorphism. Note that if the computations are not carried out in normal-basis representation it is necessary to change the representation of $x(P)$ from polynomial basis to normal basis at every step. Note that in normal-basis representation the Hamming weight of $x(P_i)$ is equal to the Hamming weight of $x(\pm\sigma^k(P_i))$ for all $k \in \{0, \ldots, 130\}$. Note also that

$$\pm\sigma(P_{i+1}) = \pm\sigma(P_i) \pm \sigma^j(\sigma(P)) = f(\pm\sigma(P))$$

and thus the step function is well defined on the classes.

For parallel Pollard rho, we also need to define distinguished points (classes). For example, we can say that a class is a distinguished one if the Hamming weight of $x(P)$ is less than or equal to w for some fixed value of w. Note that the value of a determines the parity of the Hamming weight since for all points $\mathrm{Tr}(x(P)) = \mathrm{Tr}(a)$. This means that $(\binom{n}{w} + \binom{n}{w-2} + \cdots)/2^{n-1}$ approximates the probability of distinguished points as only about 2^{n-1} different values can occur as x-coordinates.

The proposed version by Gallant, Lambert, and Vanstone [GLV00] does not use the Hamming weight to define j. Instead they use a unique representative per class, e.g. the point with lexicographically smallest x-coordinate, and put $j = \mathrm{hash}(x(P))$ for hash a hash function from $\mathbf{F}_{2^n}$ to $\{0, 1, \ldots, n-1\}$. Using the Hamming weight of $x(P)$ instead avoids the computation of the unique representative at the expense of a somewhat less random walk. There are many more points with a Hamming weight around $n/2$ than there are around the extremal values 0 and $n-1$. See Section 6.2 for analysis of the loss of randomness.

Harley included an extra tweak to method 2 by using the Hamming weight for updating the points but restricting the maximal exponent of σ in

$$P_{i+1} = f(P_i) = P_i \oplus \sigma^{\bar{j}}(P_i), \tag{4}$$

by taking $\bar{j}$ as essentially the remainder of j modulo 7. He observed that for $n = 109$ the equality $(1 + \sigma^3) = -(1 + \sigma)$ holds. He settled for scalars $(1 + \sigma^1), (1 + \sigma^2), (1 + \sigma^4), (1 + \sigma^5), (1 + \sigma^6), (1 + \sigma^7)$, and $(1 + \sigma^8)$. This limits the number of times σ has to be applied per step. For sizes larger than 109 somewhat larger exponents should be used. The walks resulting from this method are even less resembling random walks but the computation is sped up by requiring fewer squarings.

5 Cost estimates

In this section we apply Pollard's rho method to elliptic curves and give rough cost estimates in terms of field operations. The next sections will translate these

estimates to the different computation platforms we use in our attack. The fine tuning for the Certicom challenge ECC2K-130 is given in Section 6. In the following we use **I** to denote the cost of a field inversion, **M** to denote the cost of a field multiplication, and **S** to denote the cost of a field squaring.

5.1 Point representation and addition

Most high-speed implementations of elliptic-curve cryptography use inversion-free coordinate systems for the scalar multiplication, i.e. they use a redundant representation $P = (X_P : Y_P : Z_P)$ to denote the affine point $(X_P/Z_P, Y_P/Z_P)$ (for $Z_P \neq 0$). In Pollard's rho method it is important that P_{i+1} is uniquely determined by P_i. Thus we cannot use a redundant representation but have to work with affine points. For ordinary binary curves $y^2 + xy = x^3 + ax^2 + b$ addition of $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ with $x_P \neq x_Q$ is given by

$$(x_R, y_R) = (\lambda^2 + \lambda + a + x_P + x_Q, \lambda(x_R + x_P) + y_P + x_R), \text{ where } \lambda = \frac{y_P + y_Q}{x_P + x_Q}.$$

Each addition needs 1**I**, 2**M**, and 1**S**. Note that one of the multiplications could be combined with the inversion to a division. However, we use a different optimization to reduce the expense of the inversion, the by far most expensive field operation.

5.2 Simultaneous inversion

All machines will run multiple instances in parallel. This makes it possible to reduce the cost of the inversion by computing several inversions simultaneously using a trick due to Montgomery [Mon87]. Montgomery's trick is easiest explained by his approach to simultaneously inverting two elements a and b.

One first computes the product $c = a \cdot b$, then inverts c to $d = c^{-1}$ and obtains $a^{-1} = b \cdot d$ and $b^{-1} = a \cdot d$. The total cost is 1**I** and 3**M** instead of 2**I**. By extending this trick to N inputs one can obtain inverses of N elements at the cost of 1**I** and $3(N-1)$**M**.

If N processes are running in parallel on one machine and the implementation uses Montgomery's trick to simultaneously invert all N denominators appearing in the λ's above, the cost per addition decreases to $(1/N)$**I**, $(2+3(N-1)/N)$**M**, and 1**S**; for large N this can be approximated by 5**M** and 1**S**.

5.3 Inversion

Inversion is by far the most costly of the basic arithmetic operations. Most implementations use one of two algorithms: the Extended Euclidean Algorithm (EEA), published in many papers and books [MvOV96], [ACD+06] and Itoh-Tsujii's method [IT88] which essentially is using Fermat's little theorem.

Let $\mathbf{F}_{2^n}$ be represented in polynomial basis. Each field element f can be regarded as a polynomial over $\mathbf{F}_2$ of degree less than n. The EEA finds elements

u and v such that $fu + Fv = 1$. Then $uf \equiv 1 \bmod F$, $\deg u < n$ and thus u represents the multiplicative inverse of f. The classical EEA performs a full division at each step. In practice for $\mathbf{F}_{2^n} \cong \mathbf{F}_2[X]/F(X)$, implementers often choose a version of the EEA that replaces the divisions with division by X (a right shift of the operand). Although this approach requires more iterations, it is generally faster in practice. This is the approach most commonly seen when elements of $\mathbf{F}_{2^n}$ are represented in polynomial basis but special implementation strategies such as bit slicing are more suited for Itoh-Tsujii.

Although variants of EEA have been developed for normal basis representation [Sun06], the most efficient approach is generally based on Itoh-Tsujii. This algorithm proceeds from the observation that in $\mathbf{F}_{2^n}$, $f^{2^n} = f, f^{2^n-1} = 1$, and therefore $f^{2^n-2} = f^{-1}$. Instead of performing divisions as in EEA, this algorithm computes the $2(2^{n-1} - 1)$th power. If f is represented in normal basis, squarings are simply a left shift of the operand. The exponent is fixed throughout the computation, so addition chains can be used to minimize the number of multiplications. For $n = 131$ a minimum-length addition chain to reach 130 is $1, 2, 4, 8, 16, 32, 64, 128, 130$ and the corresponding addition chain on the exponents is $2^1 - 1 = 1, 2^2 - 1, 2^4 - 1, 2^8 - 1, 2^{16} - 1, 2^{32} - 1, 2^{64} - 1, 2^{128} - 1, 2^{130} - 1$.

5.4 Field representation

Typically fields are represented in a polynomial basis using the isomorphism $\mathbf{F}_{2^n} \cong \mathbf{F}_2[z]/F(z)$, where $F(z)$ is an irreducible polynomial of degree n. A basis is given by $\{1, z, z^2, \ldots, z^{n-1}\}$. In this representation addition is done component wise and multiplication is done as polynomial multiplication modulo F. Squaring is implemented as a special case of multiplication; since all mixed terms disappear in characteristic 2 the cost of a squaring is basically that of the reduction modulo F. The Certicom challenges (see Section 2) are given in polynomial-basis representation with F an irreducible trinomial or pentanomial.

An alternative representation of finite fields is to use normal bases. A normal basis is a basis of the form $\{\theta, \theta^2, \theta^{2^2}, \ldots, \theta^{2^{n-1}}\}$ for some $\theta \in \mathbf{F}_{2^n}$. Also in this representation addition is done component wise. Squaring is very easy as it corresponds to a cyclic shift of the coefficient vector: if $c = \sum_{i=0}^{n} c_i \theta^{2^i}$ then $c^2 = \sum_{i=0}^{n} c_i \theta^{2^{i+1}} = \sum_{i=0}^{n} c_{i-1} \theta^{2^i}$, where $c_{-1} = c_n$. Multiplications are significantly more complicated — to express $\theta^{2^i+2^j}$ in the basis usually a multiplication matrix is given explicitly. If this matrix has particularly few entries then multiplications are faster. The minimal number is $2n-1$; bases achieving this number are called *optimal normal bases*. For $n = 131$ such a basis exists but for $n = 163$ it does not. Alternatives are to use Gauss periods and redundant representations to represent field elements.

Converting from one basis to the other is done with help of a transition matrix, an $n \times n$ matrix over $\mathbf{F}_2$.

5.5 Cost of one step

In the following sections we will consider implementations on various platforms in different representations — in particular looking at polynomial and normal basis representations. When using distinguished points within the random walk it is important that the walk is defined with respect to a *fixed* field representation. So, if different platforms decide to use different representations it is important to change between bases to find the next step.

The following sections collect the raw data for the cost of one step, in Section 6 we go into details of how the attack against ECC2K-130 is implemented and give justification.

For the Koblitz curves when using method 1, each step takes 1 elliptic curve addition, $(n-1)\mathbf{S}$ (to find the lexicographically smallest representative), and $1\mathbf{S}$ to a variable power of the y-coordinate. In particular if $x(P)^{2^m}$ gave the unique representative, one needs to compute $y(P)^{2^m}$. Note that the intermediate powers of $y(P)$ are not needed and special routines can be implemented. We report these figures as m-squarings, costing $\mathbf{mS}$. We do not count the costs for updating the counters a_i and b_i.

When using method 2 each step takes 1 elliptic curve addition and 2 $\mathbf{mS}$. If the computations are done in polynomial basis, then also the cost for conversion to normal basis need to be considered. If Harley's speedup is used, then the m in the m-squarings is significantly restricted.

6 Fine-tuning of the attack for Certicom ECC2K-130

In this section, we describe the concrete approach we take to attack the DLP on ECC2K-130 defined in Section 2.

6.1 Choice of step function

Of course we use the Frobenius endomorphism and define the walk on classes under Frobenius and negation. Of the two methods described in Section 4 we prefer the second. Advantages are that it can be applied in polynomial basis as well as in normal basis, that it automatically avoids short, fruitless cycles and thus does not require special routines, that there is no need to store precomputed points, and that it avoids computing the unique representative in the step function (computing the Hamming weight is faster than $130\mathbf{S}$). If the main computation is done in polynomial-basis representation, a conversion to normal basis is required.

We analyzed the orders of $(1 + \sigma^j)$ modulo the group order for all small values of j. For $j \geq 3$ no small orders appear. To get sufficient randomness and to have an easily computable step function we compute the Hamming weight of $x(P)$, divide it by 2, reduce the result modulo 8 and add 3. So the updates are $(1 + \sigma^j)$ for $j \in \{3, 4, \ldots, 10\}$. For this set we additionally checked that there does not exist any linear relation with small coefficients between the discrete

logarithms of $(1+\sigma^j)$ modulo ℓ. The shortest vector in the lattice spanned by the logarithms has four-digit coefficients. This means that fruitless collisions are highly unlikely.

6.2 Heuristic analysis of non-randomness

Consider an adding walk on ℓ group elements that maps P to $P+R_0$ with probability p_0, $P+R_1$ with probability p_1, and so on through $P+R_{r-1}$ with probability p_{r-1}, where $R_0, R_1, \ldots, R_{r-1}$ are distinct group elements and $p_0 + p_1 + \cdots + p_{r-1} = 1$.

If Q is a group element, and P, P' are two independent uniform random group elements, then the probability that P, P' both map to Q without having $P = P'$ is $(1-p_0^2-p_1^2-\cdots-p_{r-1}^2)/\ell^2$. Indeed, if $P = Q-R_i$ and $P' = Q-R_j$, with $i \neq j$, then P maps to Q with probability p_i and P' maps to Q with probability p_j; there is chance $1/\ell^2$ that $P = Q - R_i$ and $P' = Q - R_j$ in the first place; overall the probability is $\sum_{i\neq j} p_i p_j/\ell^2 = (\sum_{i,j} p_i p_j - \sum_j p_j^2)/\ell^2 = (1 - \sum_j p_j^2)/\ell^2$. Adding over the ℓ choices of Q one sees that the probability of an immediate collision from P, P' is at least $(1 - \sum_j p_j^2)/\ell$.

In the context of Pollard's rho algorithm (or its parallelized version), after a total of T iterations, there are $T(T-1)/2$ pairs of iteration outputs. The inputs are not uniform random group elements, and the pairs are not independent, but one might nevertheless guess that the overall success probability is approximately $1-(1-(1-\sum_j p_j^2)/\ell)^{T(T-1)/2}$, and that the average number of iterations before success is approximately $\sqrt{\pi\ell/(2(1-\sum_j p_j^2))}$. This is a special case of the variance heuristic introduced by Brent and Pollard in [BP81].

For example, if $p_0 = p_1 = \cdots = p_{r-1} = 1/r$, then this heuristic states that ℓ is effectively divided by $1-1/r$, increasing the number of iterations by a factor of $1/\sqrt{1-1/r} \approx 1 + 1/(2r)$, as discussed by Teske [Tes01].

The same heuristic applies to a multiplicative walk that maps P to $s_j P$ with probability p_j: the number of iterations is multiplied by $1/\sqrt{1-\sum_j p_j^2}$. In particular, for the ECC2K-130 challenge, the Hamming weight of $x(P)$ is congruent to $0, 2, 4, 6, 8, 10, 12, 14$ modulo 16 with respective probabilities approximately

$$0.1443, 0.1359, 0.1212, 0.1086, 0.1057, 0.1141, 0.1288, 0.1414,$$

so our walk multiplies the number of iterations by approximately 1.069993. Note that any walk determined by the Hamming weight will multiply the number of iterations by at least $1/\sqrt{1-\sum_j \binom{131}{2j}^2/2^{260}} \approx 1.053211$.

6.3 Choice of distinguished points

In total about $2^{60.9}$ group operations are needed to break the discrete logarithm problem. If we choose Hamming weight 28 for distinguished points then there will be an average length of $2^{35.4}$ steps before hitting a distinguished point,

since $(\binom{131}{28} + \binom{131}{26} + \cdots)/2^{130}$ is approximately $2^{-35.4}$, and an expected number of $2^{25.5}$ distinguished points. Note that for this curve $a = 0$ and thus each x-coordinate has trace 0 and so the Hamming weight is even for any point. If we instead choose Hamming weight 32 for distinguished points then there will be an average length of $2^{28.4}$ steps before hitting a distinguished point and an expected number of $2^{32.5}$ distinguished points.

6.4 Implementation details

Most computations will not lead to a collision. Our implementation does not compute the intermediate scalars a_i and b_i nor does it store a list of how often each of the exponents j appeared. Instead the starting point of each walk is computed deterministically from a salt S. When a distinguished point is found, the salt together with the distinguished point is reported to the central server.

When distinguished points are noticed, $x(P)$ is given in normal basis representation, so it makes sense to keep them in normal basis. To search for collisions it is necessary to have a unique representative per class. We choose the lexicographically smallest value of $x(P), x(P)^2, \ldots, x(P)^{2^{130}}$. In normal basis representation this is easily done by inspecting all cyclic shifts of $x(P)$. To save bandwidth a 64-bit hash of this unique representative is reported to the server along with the original 64-bit seed.

If the server detects a collision on the 64-bit hash, it recovers the two starting points from the salt values and follows the random walk from the initial points, this time keeping track of how often each of the powers j appears. Like in the initial computation the distinguished point is noticed by a small Hamming weight of the normal basis representation of the x-coordinate. If the x-coordinates coincide up to cyclic shifts, i.e. the partial collision gave rise to a complete one, the unique representative per class is computed. Note that this time also the y-coordinate needs to be transformed to normal basis to obtain the correct result.

Finally the equivalence of σ and $s \bmod \ell$ is used to compute the scalars on both sides and thereby (eventually) the $\mathrm{DLP}_P(Q)$.

It is possible for a 64-bit hash collision to occur by accident, rather than from a distinguished-point collision. We could increase the 64-bit hash to 96 bits or 128 bits to eliminate these accidents. However, a larger hash costs bandwidth and server storage, and the benefit is small. Even with as many as 2^{32} distinguished points, accidental 64-bit collisions are unlikely to occur more than a few times, and disposing of the accidents has negligible cost.

We plan to use several servers storing disjoint lists of distinguished points: e.g., server i out of 8 receives the distinguished points where the first 3 hash bits are equal to i in binary. This initial routing means that each server then has to handle only 1/8 of the total volume of distinguished points.

7 FPGA implementations

The attacks presented in this section are developed for the latest version of COPACOBANA [KPP+06], which is a tightly integrated, reconfigurable hardware cluster.

The 2009 version of COPACOBANA offers 128 Xilinx Spartan-3 XC3S5000-4FG676 FPGAs, laid out on 16 plug-in modules each hosting 8 chips. Each chip is configured with 32 MB of dedicated off-chip local RAM. Serial data lines connect the FPGAs as a systolic array on the backplane, passing data packets from one device to the next in round robin fashion before they finally reach their destination. Up to two separate controller units interface the systolic array (via PCIe) to the mainboard of a low-profile PC that is integrated within the same case as COPACOBANA. In addition to data routing, the PC can perform post-processing before it stores or distributes the results to other nodes.

This section presents preliminary results comparing two choices for the underlying field arithmetic on COPACOBANA. The first approach performs arithmetic operations on elements represented in polynomial basis, converting to normal basis as needed, while the second operates directly on elements represented in normal basis.

7.1 Polynomial basis implementation

The cycle counts and delay are based on the results of implementations on FPGA (Xilinx XC3S5000-4FG676). For multiplication, digit-serial multiplier is implemented. The choice of digit-size is based on a preliminary exploration of the trade-off between speed and area. Here we choose digit-size $d = 22$ and $d = 24$ for $\mathbf{F}_{2^{131}}$ and $\mathbf{F}_{2^{163}}$, respectively.

For inversion, we used the Itoh-Tsujii algorithm. Although EEA runs faster than Itoh-Tsujii, it requires its own data path, adding extra cost in area. Using Montgomery's trick for batch inversion lowers the amortized cost to $(3-1/N)\mathbf{M}+(1/N)\mathbf{I}$ per inversion. The selection of $N = 64$ is a trade-off between performance and area. One inversion in $\mathbf{F}_{2^{131}}$ takes 212 cycles, and one multiplication takes 8 cycles. Choosing $N = 64$, each inversion on average takes 28 cycles, and the whole design takes 8 out of 104 BRAMs on the target FPGA. If we increase N to 128, the cost of one inversion drops to 26 cycles. However, the whole design requires 16 BRAMs. Since the current design uses around %11 of the LUTs of the FPGA, we can put 8$\sim$9 copies of current design onto this FPGA if we do not use more than %11 BRAMs for each.

Tab. 1 shows the cycle counts of each field operation. The cost of memory access is included.

In order to check the Hamming weight of x-coordinate, we convert the x-coordinate to normal basis in each iteration. The basis conversion and population-count operations are performed in parallel and therefore do not add extra delay to the loop.

	additions	squarings	m-squarings	multiplications	inversions	batch-inv. ($N = 64$)
$\mathbf{F}_{2^{131}}$	2	2	m+1	8	212	28
$\mathbf{F}_{2^{163}}$	2	2	m+1	9	251	31

Table 1. Cycle counts for field operations in polynomial basis on XC3S5000-4FG676

The design is synthesized with Xilinx ISE 11.1. On Xilinx XC3S5000-4fg676 FPGA, our current design for $\mathbf{F}_{2^{131}}$ and $\mathbf{F}_{2^{163}}$ can reach a maximum frequency of 74.6 MHz and 74 MHz, respectively. Table 2 shows the delay of one iteration.

	ECC2K-130		ECC2-131	ECC2-163
	Method 1	Method 2		
	$1\mathbf{I} + 2\mathbf{M} + 131\mathbf{S} + 1\mathbf{mS}$	$1\mathbf{I} + 2\mathbf{M} + 1\mathbf{S} + 2\mathbf{mS}$	$1\mathbf{I} + 2\mathbf{M} + 1\mathbf{S}$	$1\mathbf{I} + 2\mathbf{M} + 1\mathbf{S}$
Cycles	371	71	38	51
Delay (ns)	4,971	951	509	714

Table 2. Cost of one iteration in polynomial basis on XC3S5000-4FG676

The current design for ECC2K-130 uses 3,656 slices, including 1,468 slices for multiplier, 75 slices for square, 1,206 slices for base conversion and 117 slices for Hamming Weight counting. Since a Xilinx XC3S5000 FPGA has 33,280 slices, taking input/output circuits into account, we estimate 9 copies of the polynomial-basis design can fit in one FPGA. Considering 2^{35} iterations are required on average to generate one distinguished point, each copy generates 2.6 distinguished points per day. A single chip can then be expected to yield 23.4 distinguished points per day.

The current design for ECC2-131 uses 2,206 slices. Note that circuits to search for the *smallest* $x(P)$ is not included. We estimate that 12 copies (limited by BRAM) can fit in one FPGA.

The current design for ECC2K-163 uses 4,446 slices, including 2,030 slices for multiplier, 94 slices for square and 1,209 slices for base conversion and 217 slices for Hamming Weight counting. We estimate that 7 copies can fit in one FPGA.

The current design for ECC2-163 uses 3,242 slices. Note that circuits to search for the *smallest* $x(P)$ is not included. We estimate that 9 copies can fit in one FPGA.

For ECC2-131, it is the number of BRAM that stops us putting more copies of ECC engine. We believe that the efficiency of BRAMs usage can be further improved, and more copies can be instantiated on one FPGA.

7.2 Normal basis implementation

These estimates are based on an initial implementation of a bit-serial normal-basis multiplier. At the conclusion of this section, we provide an estimate for the

digit-serial version currently under development. The bit-serial design computes a single bit of the product in each clock cycle. Because of this relatively slow performance, it consumes very little area. For $\mathbf{F}_{2^{131}}$, the multiplier takes 466 slices of the chip's available 66,560, or less than 1%, while $\mathbf{F}_{2^{163}}$ requires 679 slices, or around 1%.

	additions	squarings	m-squarings	multiplications	inversions	batch-inversions
$\mathbf{F}_{2^{131}}$	1	1	1	131	1065	
$\mathbf{F}_{2^{163}}$	1	1	1	163	1629	

Table 3. Cycle counts for field operations in normal basis on Xilinx XC3S5000-5FG1156

An implementation of the full Rho step would instantiate at least one multiplier expressly for the Itoh-Tsujii inversion routine and process several points simultaneously to keep the inversion unit busy. Our implementation results take this approach: each engine consists of one multiplier dedicated to inversion and eight for ordinary multiplication. The result for $\mathbf{F}_{2^{131}}$ is an area requirement of 1,915 slices while achieving a clock rate of 85.898 MHz.

The inversion unit is the bottleneck at 1,065 cycles required. We can use Montgomery's trick to perform many simultaneous inversions. So the design challenge becomes one of keeping the inversion unit busy, spreading the cost of the inverter across as many simultaneous Rho steps as possible.

Suppose we process 32 inversions simultaneously using one inversion unit. This operation introduces $8\mathbf{M}$ cycles of delay. Method 2 requires $1\mathbf{I}+2\mathbf{M}+1\mathbf{S}+1\mathbf{mS}$ to compute one step of the Rho method. Because squarings are free, we must compute two additional multiplications per step. On top of this cost are the $3(N-1) = 157$ multiplications needed for the pre- and post-computation to obtain individual inverses. Although we might be able to spread a particular step across several multipliers, we can safely assume that keeping the inversion unit busy this way requires up to a total of 32 additional multipliers. The chip has embedded storage for up to 1,664 points, far more than required. With this approach, these multipliers would be idle roughly five-eighths of the time; this estimate is meant to be conservative. At a cost of 466 slices each, we can expect an engine to consume a total of 14,912 slices. Instantiating all these multipliers has the advantage that we can complete 32 steps every 1,065 cycles, or one step every 34 cycles. At 85.898 MHz, that equates to 2,526,411 steps per second, or 6 distinguished points per day per engine. As a single chip has 66,360 slices available, four of these engines could be instantiated per chip, yielding 24 distinguished points per day per chip. This figure may underestimate the time required for overhead like memory access.

Mitigating this effect is the fact that the time-area product can surely be improved by modifying the multiplier to use a digit-serial approach as described in [NS06]. This time-area tradeoff processes d bits of the product simultaneously

at a cost of additional area. We are currently implementing this approach; the previous work indicates that this approach can improve the time-area product.

As of this writing, normal-basis results for $\mathbf{F}_{2^{163}}$ are still pending.

7.3 Comparison

Both polynomial and normal-basis implementations offer roughly the same performance today. Because the polynomial-basis implementation is in a more mature state, embodying the entire step of the Rho method, it represents less risk and uncertainty in terms of use in a practical attack. It achieves this performance despite the fact that this particular Certicom challenge would appear to be ideally suited to a normal-basis implementation. As a case in point, the polynomial version must pay the overhead to convert elements back to normal basis at the end of each step to check if a distinguished point has been reached.

As this paper represents work in progress, the normal-basis figures in this section are based on measurements only of the field arithmetic time and area. While the estimates may not fully account for overhead like memory access, they also do not capture the effect of migrating to a digit-serial multiplier. Because multiplication cost dominates the cost of inversion – and therefore the cost of a step of the Rho method, the normal-basis approach may ultimately offer higher performance because in this particular attack. Performance of the polynomial-basis version may also improve in unforeseen ways and it would undoubtedly outperform the normal-basis version in an attack on ECC2-131.

8 Hardware implementations

In this section we present and discuss the results achieved while targeting ASIC platforms. To obtain the estimates presented in this work, we have used the UMC L90 CMOS technology, using the Free Standard Cell library from Faraday Technology Corporation characterized for HS-RVT (high speed regular Vt) process option with 8 Metal layers. For synthesis results we have used design compiler 2008.09 from Synopsys, and for placement and routing we have used SoC Encounter 7.1-usr3 from Cadence Design Systems.

Both synthesis and post-layout analysis use typical corner values, and reasonable assumptions for constraints. The designs are for performance estimate and are not refined for actual production (i.e. they do not contain test structures, a practical I/O subsystem). During synthesis, multiple runs were made with different constraints, and the results with the best area time product have been selected.

8.1 Polynomial basis implementation

For the polynomial basis, the estimates are based on the work presented by Meurice de Dormale et al. in [MdDBQ07] and Bulens et al. in [BMdDQ06] where the authors proposed an implementation based on a pipelined architecture.

Addition The addition of n-bits will always be equal to $2n$-input XOR gates. No separate synthesis has been made for this circuit. In a practical circuit the interconnection between the adder the rest of the circuit would have a significant impact upon the delay and the area of the circuit. The delay for the addition is approximately 75 ps, while the required area is approximately $n{\cdot}2.5$ Gate Equivalents (GEs). No post-layout results are given for this circuit as a standalone implementation is not representative.

Squaring Squaring is performed with a parallel squarer. This operator is relatively inexpensive thanks to the use of a sparse irreducible polynomial, a pentanomial, which is hardwired. Each bit of the result is therefore computed using few XOR gates. The detailed results for ASIC implementation are reported in Table 4, where, due to the small size of this component, no post-layout analysis is provided.

	$\mathbf{F}_{2^{131}}$	$\mathbf{F}_{2^{163}}$
	Pre-layout	Pre-layout
Delay (ps) $\sim$	250	250
Frequency (GHz) $\sim$	4.0	4.0
Flip-Flop Number	131	163
Area (mm^2) $\sim$	0.003	0.004
GE $\sim$	960	1200

Table 4. Results for ASIC implementation of Squarer in polynomial basis

m-Squaring This operation can be implemented using a single squarer that iteratively computes the m squarings in m clock cycles.

Multiplication In order to perform the modular multiplication, we used a subquadratic Karatsuba parallel multiplier. As for squaring, the modular reduction step is easily performed with a few XOR gates for each bit of the result. The modular multiplier has a pipeline depth of 8 clock cycles in both $\mathbf{F}_{2^{131}}$ and $\mathbf{F}_{2^{163}}$ and produces a result at each clock cycle. The detailed results for the modular multiplication are reported in Table 5

	$\mathbf{F}_{2^{131}}$		$\mathbf{F}_{2^{163}}$	
	Pre-layout	Post-layout	Pre-layout	Post-layout
Delay (ps) $\sim$	470	585	500	575
Frequency (GHz) $\sim$	2.1	1.7	2.0	1.7
Flip-Flop Number	4608	4608	5768	5768
Area (mm^2) $\sim$	0.175	0.185	0.227	0.250
GE $\sim$	55000	59000	72500	80000

Table 5. Results for ASIC implementation of Multiplication in polynomial basis

Inversion The inversion is based on Fermat's little theorem with the multiplication chain technique by Itoh and Tsujii. This method is preferred to the extended Euclidean algorithm according to the comparison of both methods performed in the work by Bulens et al. [BMdDQ06]. An inversion requires 130 squarings and 8 multiplications in $\mathbf{F}_{2^{131}}$ and 162 squarings and 9 multiplications in $\mathbf{F}_{2^{163}}$. The inverter uses the parallel multiplier described above and a block of several pipelined consecutive squarers. This block allows to speed up the consecutive squarings required in the inversion using the Itoh-Tsujii technique. It is done by putting several squarers in a serial way, so that every bit of the result is now computed with a larger number of XOR gates, depending on the number of squarings to be performed. Within the inverter, the block of consecutive squarers actually allows to compute several possible numbers of consecutive squarings (the numbers of consecutive squarings specified by the technique of Itoh and Tsujii). The inversion is done by looping over the multiplier and the squarer according to the technique of Itoh and Tsujii. For this purpose, extra shift registers are required. The inverter has a pipeline depth of 16 and achieves an inversion with a mean delay of 10 and 11 clock cycles in $\mathbf{F}_{2^{131}}$ and $\mathbf{F}_{2^{163}}$ respectively.
A lower bound on the area requirement of the inverter can be found by gathering the results for the squarer and the multiplier. This leads to around 60000 GE for $\mathbf{F}_{2^{131}}$ and 81200 for $\mathbf{F}_{2^{163}}$. However, this assumes an iterative use of the squarer. For the parallel pipelined inverter described above, a better estimate can be obtained by approximating the block of consecutive squarers as a series of single squarers. The number of these consecutive squarers is given by the maximum number of consecutive squarings in the technique of Itoh and Tsujii on $\mathbf{F}_{2^{131}}$ and $\mathbf{F}_{2^{163}}$, i.e. 64 in both cases. As a result, the area cost of the block of consecutive squarers should be around 64000 and 76800 GE for $\mathbf{F}_{2^{131}}$ and $\mathbf{F}_{2^{163}}$ respectively, leading to an area cost of 123000 GE and 156800 for the full parallel inverter on $\mathbf{F}_{2^{131}}$ and $\mathbf{F}_{2^{163}}$ respectively.
The inverter has a much higher area cost and lower throughput with respect to the multiplier. From an area-time point of view, it is interesting to combine several inversions using the Montgomery trick instead of replicating several inverters when trying to increase the throughput of inversions. As each combined inversion requires 3 multiplications to be performed, it seems to be always interesting to use the Montgomery trick instead of replicating several inverters. In practice, the gain of using Montgomery's trick can be smaller than expected, as stated in [BMdDQ06]. Simultaneous inversion does not require a specific operator as it can be built upon the multiplier and the inverter.

Conversion A conversion from polynomial to normal basis is required when using method 2 as described in Section 4.2. The detailed results for this operation are reported in Table 6.

	$\mathbf{F}_{2^{131}}$	$\mathbf{F}_{2^{163}}$
	Pre-layout	Pre-layout
Delay (ps) $\sim$	410	460
Frequency (GHz) $\sim$	2.4	2.15
Flip-Flop Number	0	0
Area (mm^2) $\sim$	0.044	0.061
GE $\sim$	13900	19400

Table 6. Results for ASIC implementation of Polynomial to Normal Basis converter

	additions	squarings	m-squarings	multiplications	inversions
$\mathbf{F}_{2^{131}}$	1	1	m	1	10
$\mathbf{F}_{2^{163}}$	1	1	m	1	11

Table 7. Cycle counts for field operations in polynomial basis on ASIC

8.2 Normal basis implementation

Addition For ASIC, the addition is performed in a way similar to the one of polynomial bases.

Squaring In normal basis, the squaring is equivalent to a rotation, thus the impact on the area and delay of this component will be mainly due to the interconnections.

m-Squaring This operation can be achieved by an iterative use of a single squarer as for the polynomial basis. However, since a squaring is equivalent to a rotation, one can anticipate the m rotations and perform this operation in one clock cycle.

Multiplication The proposed implementation is based on a bit-serial multiplier which calculates a single bit product every clock cycle. The detailed results for the modular multiplication are reported in Table 8.

	$\mathbf{F}_{2^{131}}$		$\mathbf{F}_{2^{163}}$	
	Pre-layout	Post-layout	Pre-layout	Post-layout
Delay (ps) $\sim$	510	550	485	576
Frequency (GHz) $\sim$	2.0	1.8	2.0	1.75
Flip-Flop Number	402	402	661	611
Area (mm^2) $\sim$	0.015	0.016	0.025	0.027
GE $\sim$	5000	5250	7900	8800

Table 8. Results for ASIC implementation of Field element multiplier in normal basis

Inversion Also in normal bases, the inversion is implemented using Fermat's little theorem, thus the considerations of Section 8.1 hold also in this case. The number of cycle counts for the operations in normal basis on ASIC are shown in Table 9. The area requirement of the inverter in normal basis

should not be significantly higher than the cost of the multiplier since the multiple squarings can be performed with interconnections only. Therefore, it should be close to the lower bound given by the area cost of the multiplier, i.e. 5250 GE for $\mathbf{F}_{2^{131}}$.

	additions	squarings	m-squarings	multiplications	inversions
$\mathbf{F}_{2^{131}}$	1	1	1	131	1178
$\mathbf{F}_{2^{163}}$	1	1	1	163	1629

Table 9. Cycle counts for field operations in normal basis on ASIC

8.3 Cost of step on ECC2K-130

A step in the Pollard rho computation on ECC2K-130 consumes $1\mathbf{I}+2\mathbf{M}+131\mathbf{S}+1\mathbf{mS}$ using method 1. In polynomial basis, this takes 273 cycles on an ASIC while in normal basis it takes 1572 cycles. Using method 2, a step requires $1\mathbf{I}+2\mathbf{M}+1\mathbf{S}+2\mathbf{mS}$ and the additional conversion to normal basis if the computations are done in polynomial basis. In polynomial basis, the step is performed in 274 cycles on an ASIC while in normal basis it is done in 1443 cycles. A larger cycle count for the normal basis is due to the iterative approach of the design of the components in this basis as opposed to the parallel design used for the polynomial basis. These cycles counts do not consider the use of simultaneous inversion since the gain of this method should be assessed once the whole processor is assembled, following [BMdDQ06]. This technique should be employed as much as possible given the high cost of an inversion. Therefore, the cycle counts are likely to be significantly lower in practice. For instance, combining 10 inversions theoretically lowers the cycle count for the step in normal basis by about 50%.

Concerning the area, a lower bound can be determined by gathering the costs of the operators. Note that this bound does not include the cost of storing elements. In polynomial basis, the lower bound on the area of a processor based on the components described above is roughly 125000 GE when using method 1 and 140000 GE for method 2. It is mostly the cost of the inverter, as its multiplier can also be used to perform the two multiplications needed in each step. In normal basis, the same estimate leads to 6000 GE (mainly the cost of the multiplier). Based on these estimates, the processor relying on the normal basis appears to be more efficient from an area-time point of view as it is roughly 6 times slower but 20 times smaller. However, the use of several squarers in parallel should improve the area-time product of the processor relying on the polynomial basis.

8.4 Cost of step on ECC2-131

A step in the Pollard rho computation on ECC2-131 consumes $1\mathbf{I}+2\mathbf{M}+1\mathbf{S}$. In polynomial basis this takes 13 cycles on ASIC while in normal basis it takes

1441 cycles. Again, the iterative approach used in the normal basis causes a higher cycle count for the normal basis operators. The area costs are on the same order as on ECC2K-130. Therefore, the processor based on the polynomial basis appears to be more efficient here since it is 110 faster while being only 20 times larger.

8.5 Cost of step on ECC2-163

A step in the Pollard rho computation on ECC2-163 consumes $1\mathbf{I} + 2\mathbf{M} + 1\mathbf{S}$. In polynomial basis this takes 14 cycles on ASIC while in normal basis it takes 1956 cycles.

The processor based on the polynomial basis has a lower bound on the area cost around 160000 GE (mainly the cost of the inverter). It is 140 times faster than the one based on the normal basis. For this reason, it is expected to be more area-time efficient, as on ECC2-131.

8.6 Detailed cost of the full attack on ECC2K-130

With the cost of individual arithmetic blocks given above, we can now attempt a first order estimation of cost and performance of an ASIC that could be used for the ECC2K-130 challenge. We will consider using a standard 16mm2 die in the 90nm CMOS technology using a multi-project wafer (MPW) production in prototype quantities. The cost of producing and packaging 50-250 of such ASICs is estimated to be less than 60 000 EUR.

The performance of the subcomponents described in the preceding sections were all post-layout figures that include interconnection delays, clock distribution overhead and all required registers. We estimate that the overall clock rate will be around 1.5 GHz when all the components are combined into one system. Each chip would require a PLL for to distribute the internal clock. In this estimation we will again leave a generous margin in the timing and will assume that the system clock is 1.25 GHz.

When using the specific standard cell library the 16mm2, 90nm CMOS die has a net core area of 12mm2, which could support approximately 3.000.000 gates with generous overhead for power routing and placement. Reserving 1.000.000 gates for PLL, I/O interface and a shared point inspection unit, we will consider that only 2.000.000 gates will be available for point calculation units. A single chip could thus support 400 cores in parallel. Considering that each core requires 1572 clock cycles per iteration (section 7.3), such an ASIC would be able to calculate approximately 320 Miterations/s. The estimation of the complete attack able to break ECC2K-130 in one year would require approximately 69 000 Miterations/s. Even our overly pessimistic estimation shows that this performance can be achieved by a modest collection of around 220 dedicated ASICs.

9 AMD64 implementations

This section describes our software implementation for general-purpose CPUs supporting the amd64 (also known as "x86-64") instruction set. This implementation is tuned for Intel's popular Core 2 series of CPUs but also performs reasonably well on other recent CPUs, such as the AMD Phenom.

9.1 Bitslicing

New speed records for elliptic-curve cryptography on the Core 2 were recently announced in a Crypto 2009 paper [Ber09a]. The new speed records combine a fast complete addition law for binary Edwards curves, fast bitsliced multiplication for arithmetic in $\mathbf{F}_{2^n}$, and the Core 2's fast instructions for arithmetic on 128-bit vectors. Our amd64 implementation combines the bitsliced multiplication techniques from [Ber09a] with several additional techniques for bitsliced computation. (Binary Edwards curves do not appear to save time in this context; the implementation uses standard affine coordinates for Weierstrass curves.)

Bitslicing a data structure is a simple matter of transposition. Our implementation represents 128 elliptic-curve points $(x_0, y_0), (x_1, y_1), \ldots, (x_{127}, y_{127})$, with $x_i, y_i \in \mathbf{F}_{2^n}$, as $2n$ vectors $X_0, X_1, \ldots, X_{n-1}, Y_0, Y_1, \ldots, Y_{n-1}$, where the jth bits of X_i and Y_i are the ith bits of x_j and y_j respectively.

"Logical" vector operations act on bitsliced inputs as 128-way SIMD instructions. For example, a vector XOR carries out xors in parallel on 128 pairs of bits. Multiplication in $\mathbf{F}_{2^n}$ can be decomposed into bit operations, a series of bit XORs and bit ANDs; one can carry out 128 multiplications on 128 pairs of bitsliced inputs in parallel by performing the same series of vector XORs and vector ANDs. XOR and AND are not the only operations available, but other operations do not seem helpful for multiplication, which takes most of the time in this implementation.

Similar comments apply to higher-level computations, such as elliptic-curve arithmetic over $\mathbf{F}_{2^n}$. There is an obvious analogy between designing bitsliced software and designing ASICs, but one should not push the analogy too far: there are fundamental differences in gate costs, communication costs, etc.

Many common software-implementation techniques for $\mathbf{F}_{2^n}$ arithmetic, such as precomputed multiplication tables, perform quite badly when expressed as bit operations. However, bitslicing allows free shifts, masks, etc., and fast bitsliced algorithms for binary-field arithmetic outperform all known non-bitsliced algorithms.

9.2 Results

Tables 10 and 11 show measurements of the number of cycles per input used for various computations. These measurements are averages across billions of steps.

Elliptic-curve addition uses $1\mathbf{I} + 2\mathbf{M} + 1\mathbf{S}$, and computing the input to the addition uses $20\mathbf{S}$. (A non-bitsliced algorithm would use only $13\mathbf{S}$ on average, but

	addition	squaring	multiplication	inversion	normal	step
$\mathbf{F}_{2^{131}}$	3.5625	2.6406	85.32	1089.4	103.84	694
$\mathbf{F}_{2^{163}}$	4.0625	2.9141	140.35	1757.18	157.58	1059

Table 10. Cycle counts per input for bitsliced field operations in polynomial basis on a 3000MHz Core 2 Q6850 6fb

	addition	squaring	multiplication	inversion	normal	step
$\mathbf{F}_{2^{131}}$	4.4141	3.8516	101.40	1509.9	59.063	778
$\mathbf{F}_{2^{163}}$	4.9688	4.5859	170.98	2460.9	129.180	1249

Table 11. Cycle counts per input for bitsliced field operations in polynomial basis on a 2200MHz Phenom 9550 100f23

squarings are not the main operation in this implementation.) The implementation batches 48**I** into $1\mathbf{I} + 141\mathbf{M}$, and then computes $1\mathbf{I}$ as $8\mathbf{M} + 130\mathbf{S}$ (in the case of ECC2K-130), so on average each inversion costs $3.10417\mathbf{M} + 2.70833\mathbf{S}$. The total cost of field arithmetic in a step is therefore $5.10417\mathbf{M} + 23.70833\mathbf{S}$. The "step" cycle count shown above includes more than field arithmetic:

- About 63% of the time (on a Core 2) is spent on multiplications.
- About 15% of the time is spent on conversion to normal-basis representation. This computation uses the algorithm described in [Ber09b].
- About 9% of the time is spent on squarings.
- About 7% of the time is spent on additions.
- About 3% of the time is spent on weight calculation. This calculation combines standard full-adder circuits in an obvious way, adding (for example) 15 bits by first adding 7 bits, then adding 7 more bits, then adding the two sums to the remaining bit.
- The remaining 3% of the time is spent on miscellaneous overhead.

The 48 inversions are actually 48 bitsliced inversions of $48 \cdot 128$ field elements, each containing n bits. The implementation handles $48 \cdot 128$ points (x, y) in parallel. Each x-coordinate (in batches of 128) is converted to normal-basis representation, compressed to a Hamming weight, checked for being distinguished, and then further compressed to three bits that determine the point (x', y') that will be added to (x, y). The implementation computes each x' by repeated squaring, stores $x' + x$ along with (x, y), and inverts $x' + x$. The total active memory for all $x, y, x' + x, 1/(x' + x)$ is $4 \cdot 48 \cdot 128$ field elements, together occupying $3072n$ bytes: i.e., 402432 bytes for $n = 131$, or 500736 bytes for $n = 163$. Subsequent elliptic-curve operations use only a few extra field elements.

10 Cell implementations

Jointly developed by Sony, Toshiba and IBM, the Cell Broadband Engine (Cell) architecture [Hof05] is equipped with one dual-threaded, 64-bit in-order Power

Processing Element (PPE), which can offload work to the eight Synergistic Processing Elements (SPEs) [TCC+05]. Each SPE consists of a Synergistic Processing Unit (SPU), 256 kilobytes of private memory called Local Store (LS), a Memory Flow Controller, and a register file containing 128 registers of 128 bits each. The SPUs are the target of our Cell implementation and allow 128-bit wide single instruction, multiple data (SIMD) operations. The SPUs are asymmetric processors, having two pipelines (denoted by the odd and the even pipeline) which are designed to execute two disjoint sets of instructions. Hence, in the ideal case, two instructions can be dispatched per cycle.

All performance measurements for the Cell stated in this section are obtained by running on a single SPE on a PlayStation 3 (PS3) video game console on which the programmer has access to six SPEs.

Like the amd64 architecture, the SPU supports bit-logical operations on 128-bit registers. Hence, a bitsliced implementation — similar to the one presented in Section 9 — seems to be a good approach. However, for two reasons it is much harder to achieve good performance with the same techniques on the SPU:

The first reason is the restricted local storage size of only 256 KB. As bitsliced implementations work on 128 inputs in parallel, they need much more memory for intermediate values than a non-bitsliced implementation. Even if the code and intermediate results fit into 256 KB, the batch size for Montgomery inversions has to be smaller yielding a higher number of costly inversions per iteration. The second reason is that all instructions are executed in order; a fast implementation requires loop unrolling in several functions increasing the code size and limiting the available storage for batching even further.

We decided to implement both a bitsliced version and a non-bitsliced version to compare which approach gives better results for the SPE. Both implementations required hand-optimizing the code on the assembly level, the main focus is on the ECC2K-130 challenge.

10.1 Non-bitsliced implementation

We decided to represent 131-bit polynomials using two 128-bit vectors. Let $A, B \in \mathbf{F}_{2^{131}}$ in polynomial basis. In order to use 128-bit look-up tables and to get 16-bit aligned intermediate results the multiplication is broken into parts as follows

$$\begin{aligned}
A &= A_l + A_h \cdot z^{128} = \widetilde{A}_l + \widetilde{A}_h \cdot z^{121} \\
B &= B_l + B_h \cdot z^{128} = \widetilde{B}_l + \widetilde{B}_h \cdot z^{15} \\
C &= A \cdot B = \widetilde{A}_l \cdot B_l + \widetilde{A}_l \cdot B_h \cdot z^{128} + \widetilde{A}_h \cdot \widetilde{B}_l \cdot z^{121} + \widetilde{A}_h \cdot \widetilde{B}_h \cdot z^{136}
\end{aligned}$$

For the first two multiplications a four bit lookup table and for the third and fourth a two bit lookup table are used. The squaring is implemented by inserting a 0 bit between consecutive bits of the binary representation. In order to hide latencies two steps are interleaved aiming at filling both pipelines in order to reduce the total number of required cycles. The m-squarings are implemented using look-up tables for $3 \leq m \leq 10$ and take for any such value of m a constant

number of cycles. The inversion is implemented using a sequence of squarings, m-squarings and multiplications.

The optimal number of concurrent walks is as large as possible, i.e. such that the executable and all the required memory fit in the LS. In practice 256 walks are processed in parallel.

	addition	squaring	m-squaring	multiplication	inversion	normal	step
$\mathbf{F}_{2^{131}}$	1 - 2	44	98	161	8000	98	1293

Table 12. Cycle counts per input for non-bitsliced field operations in polynomial basis on one SPE of a 3192 MHz Cell Broadband Engine, rev. 5.1

Cost of step on ECC2K-130 A step in the Pollard rho computation on ECC2K-130 consumes $\frac{1}{256}\mathbf{I} + 5\mathbf{M} + 1\mathbf{S} + 2\mathbf{mS}$ plus the conversion from polynomial to normal basis when using method 2 as described in Section 4.2. The required number of cycles for one iteration on the curve ECC2K-130 are stated in Table 12. Addition is done by two XOR instructions, which go in the even pipeline, and requires at most two and at least one instruction if interleaved with two odd instructions. There are 118 miscellaneous cycles which include the additions, the calculation of the weight, the test if a point is distinguished and various overhead. The cycle counts stated in Table 12 are obtained by "counting" the required number of cycles of our assembly code with the help of the SPU timing tool: a static timing analysis tool available for the Cell.

10.2 Bitsliced implementation

The bitsliced implementation is based on the C++-code for the amd64 architecture. In a first step we ported the code to C, to reduce the size of the resulting binary. For the ECC2K-130 we then implemented bitsliced versions of multiplication, reduction, squaring, addition and conditional move (cmov) in assembly to accelerate the computations.

The maximal batch size that we can use for the ECC2K-130 challenge is 14. Timings for the implementation are given in Table 13, all timings include costs for function calls, they ignore costs for reading the input, which is negligible for long computations until a distinguished point has been found. The measurements are averages across billions of steps measured at runtime.

As for the amd64 implementation, elliptic-curve addition uses $1\mathbf{I} + 2\mathbf{M} + 1\mathbf{S}$, and computing the input to the addition uses $20\mathbf{S}$. The implementation batches $14\mathbf{I}$ into $1\mathbf{I} + 39\mathbf{M}$, and then computes $1\mathbf{I}$ as $8\mathbf{M} + 130\mathbf{S}$ so on average each inversion costs $3.3571\mathbf{M} + 9.2857\mathbf{S}$. The total cost of field arithmetic in a step is therefore $5.3571\mathbf{M} + 30.2857\mathbf{S}$. The "step" cycle count shown above includes more than field arithmetic:

	addition	cmov	squaring	multiplication	inversion	normal	step
$\mathbf{F}_{2^{131}}$	5.7422	6.3672	4.5625	131.2656	1870.6016	33.0391	1179.5078

Table 13. Cycle counts per input for bitsliced field operations in polynomial basis on one SPE of a 3192 MHz Cell Broadband Engine, rev. 5.1

- About 60% of the time is spent on multiplications.
- About 3% of the time is spent on conversion to normal-basis representation.
- About 12% of the time is spent on squarings.
- About 3% of the time is spent on additions.
- About 3% of the time is spent on conditional moves.
- About 11% of the time is spent on weight calculation.
- The remaining 8% of the time is spent on miscellaneous overhead.

For algorithmic details see also Section 9.

11 Complete implementation of the attack

Eventually all platforms described so far will be used to attack ECC2K-130. As a proof of concept and as infrastructure for our optimized implementations we built `ref-ntl`, a C++ reference implementation of an ECC2K discrete-logarithm attack using Shoup's NTL for field arithmetic. The implementation has several components:

- Descriptions of several different ECC2K challenges that the user can target. Similarly to the data in Section 2 each description consists of an irreducible polynomial F, curve parameters a, b, curve points P, Q in hexadecimal, the order ℓ of P, the root s of $T^2 + (-1)^a T + 2$ modulo ℓ that corresponds to the Frobenius endomorphism (so that $\sigma(P) = [s]P$), a choice of normal-basis generator, and a choice of weight defining distinguished points.
- A `setup` program that converts a series of 64-bit seeds $t_1, t_2, \ldots$ into a series of curve points $A(t_1)P \oplus Q, A(t_2)P \oplus Q, \ldots$. The function A uses AES to expand each seed t_j into a bit-string $(c_{j,127}, c_{j,126}, \ldots, c_{j,1}, c_{j,0})$ of length 128 and then interprets it as a Frobenius expansion to compute starting point $Q \oplus \sum_{i=0}^{127} c_{j,i}\sigma^i(P)$.
- An `iterate` program that given an elliptic curve point iterates the step function until a distinguished point is found and then reports it to a server. This computation is the real bottleneck in the implementation; the task of the optimized implementations in Sections 7–10 is to perform the same computation as quickly as possible on various platforms.
- A short script that normalizes each distinguished point and sorts the normalized distinguished points to find collisions.
- A verbose variant of the `iterate` program that, starting from two colliding seeds, recomputes the corresponding distinguished points while keeping track of the iteration steps. This recomputation takes negligible time and removes

the need for the optimized implementations to keep track of the iteration steps.
- Final programs `finish` and `verify` that express each of the colliding normalized distinguished points as linear combinations of P and Q and that print the discrete logarithm of Q base P.

As an end-to-end test of the implementation we solved a randomly generated challenge over $\mathbf{F}_{2^{41}}$, using about one second of computation on one core of a 2.4GHz Core 2 Quad. We checked the result using the Magma computer-algebra system. We then solved ECC2K-95, as described in Section 1.

References

[ACD+06] Roberto Avanzi, Henri Cohen, Christophe Doche, Gerhard Frey, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. CRC Press, 2006.

[Ber09a] Daniel J. Bernstein. Batch binary Edwards. In *Crypto 2009*, volume 5677 of *LNCS*, pages 317–336, 2009.

[Ber09b] Daniel J. Bernstein. Optimizing linear maps modulo 2, 2009. `http://cr.yp.to/papers.html#linearmod2`.

[BMdDQ06] Philippe Bulens, Guerric Meurice de Dormale, and Jean-Jacques Quisquater. Hardware for Collision Search on Elliptic Curve over GF(2^m). In *Proceedings of SHARCS'06*, 2006. `http://www.hyperelliptic.org/tanja/SHARCS/talks06/bulens.pdf`.

[BP81] Richard P. Brent and John M. Pollard. Factorization of the eighth Fermat number. *Mathematics of Computation*, 36:627–630, 1981.

[Cer97a] Certicom. Certicom ECC Challenge. `http://www.certicom.com/images/pdfs/cert_ecc_challenge.pdf`, 1997.

[Cer97b] Certicom. ECC Curves List. `http://www.certicom.com/index.php/curves-list`, 1997.

[GLV00] Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Improving the parallelized Pollard lambda search on anomalous binary curves. *Mathematics of Computation*, 69(232):1699–1705, 2000.

[Har] Robert Harley. Elliptic curve discrete logarithms project. `http://pauillac.inria.fr/~harley/ecdl/`.

[Har60] Bernard Harris. Probability distributions related to random mappings. *The Annals of Mathematical Statistics*, 31:1045–1062, 1960.

[Hof05] H. Peter Hofstee. Power efficient processor architecture and the Cell processor. In *HPCA 2005*, pages 258–262. IEEE Computer Society, 2005.

[IT88] Toshiya Itoh and Shigeo Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases. *Inf. Comput.*, 78(3):171–177, 1988.

[KPP+06] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler. Breaking ciphers with COPACOBANA-a cost-optimized parallel code breaker. *Lecture Notes in Computer Science*, 4249:101, 2006.

[MdDBQ07] Guerric Meurice de Dormale, Philippe Bulens, and Jean-Jacques Quisquater. Collision Search for Elliptic Curve Discrete Logarithm over GF(2^m) with FPGA. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES 2007)*, pages 378–393. Springer, 2007.

[Mon87] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48:243–264, 1987.

[MvOV96] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[NS06] M. Novotnỳ and J. Schmidt. Two Architectures of a General Digit-Serial Normal Basis Multiplier. In *Proceedings of 9th Euromicro Conference on Digital System Design*, pages 550–553, 2006.

[Pol78] John M. Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of Computation*, 32:918–924, 1978.

[Sun06] Berk Sunar. A Euclidean algorithm for normal bases. *Acta Applicandae Mathematicae*, 93:57–74, 2006.

[TCC+05] Osamu Takahashi, Russ Cook, Scott Cottier, Sang H. Dhong, Brian Flachs, Koji Hirairi, Atsushi Kawasumi, Hiroaki Murakami, Hiromi Noro, Hwa-Joon Oh, Shoji Onish, Juergen Pille, and Joel Silberman. The circuit design of the synergistic processor element of a Cell processor. In *ICCAD 2005*, pages 111–117. IEEE Computer Society, 2005.

[Tes01] Edlyn Teske. On random walks for Pollard's rho method. *Mathematics of Computation*, 70(234):809–825, 2001.

[vOW99] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.

[WZ98] Michael J. Wiener and Robert J. Zuccherato. Faster attacks on elliptic curve cryptosystems. In *Selected Areas in Cryptography*, volume 1556 of *LNCS*, pages 190–200, 1998.

Intel's New AES and Carry-Less Multiplication Instructions—Applications and Implications

Shay Gueron

University of Haifa and Intel Corporation, Israel

Intel is adding to its processors 6 new instructions (AESENC, AESENCLAST, AESDEC, AESDELAST, AESIMC, AESKEYGENASSIST) that facilitate secure and high performance AES encryption, decryption, and key expansion, and one new instruction (PCLMULQDQ) that performs carry-less multiplication. PCLMULQDQ can be used in several applications, including elliptic curve cryptography, CRC, and the Galois Counter Mode (GCM). The talk will provide details on the instructions, their various usage models, and how they enhance performance and security. In particular, we will explain why and how parallel modes of operation can gain significantly from the instructions.

FSBday:

Implementing Wagner's generalized birthday attack against the SHA-3* round-1 candidate FSB

Daniel J. Bernstein[1], Tanja Lange[2], Ruben Niederhagen[3], Christiane Peters[2], and Peter Schwabe[2] **

[1] Department of Computer Science
University of Illinois at Chicago, Chicago, IL 60607–7045, USA
djb@cr.yp.to
[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, Netherlands
tanja@hyperelliptic.org, c.p.peters@tue.nl, peter@cryptojedi.org
[3] Lehrstuhl für Betriebssysteme, RWTH Aachen University
Kopernikusstr. 16, 52056 Aachen, Germany
ruben@polycephaly.org

Abstract. This paper applies generalized birthday attacks to the FSB compression function, and shows how to adapt the attacks so that they run in far less memory. In particular, this paper presents details of a parallel implementation attacking FSB_{48}, a scaled-down version of FSB proposed by the FSB submitters. The implementation runs on a cluster of 8 PCs, each with only 8GB of RAM and 700GB of disk. This situation is very interesting for estimating the security of systems against distributed attacks using contributed off-the-shelf PCs.
Keywords: SHA-3, Birthday, FSB – Wagner, not much Memory

1 Introduction

The hash function FSB [2] uses a compression function based on error-correcting codes. This paper describes, analyzes, and optimizes a parallelized generalized birthday attack against the FSB compression function.

This paper focuses on a reduced-size version FSB_{48} which was suggested as a training case by the designers of FSB. The attack has not finished at the time of writing this document but we give performance figures for running the code for this attack. Our results allow us to estimate how expensive a similar attack would be for full-size FSB.

A straightforward implementation of Wagner's generalized birthday attack [12] would need 20 TB of storage. However, we are running the attack on 8

* SHA-2 will soon retire, see [10]

** This work was supported by the National Science Foundation under grant ITR-0716498, by the European Commission under Contract ICT-2007-216499 CACE, and by the European Commission under Contract ICT-2007-216646 ECRYPT II. Permanent ID of this document: ded1984108ff55330edb8631e7bc410c. Date: 2009.09.01.

nodes of the Coding and Cryptography Computer Cluster (CCCC) at Technische Universiteit Eindhoven which has a total hard disk space of only 5.5 TB. We detail how we deal with this restricted background storage, by applying and generalizing ideas described by Bernstein in [6] and compressing partial results. We also explain the algorithmic measures we took to make the attack run as fast as possible, carefully balancing our code to use available RAM, network throughput, hard disk throughput and computing power.

We are to the best of our knowledge the first to give a detailed description of a full implementation of a generalized birthday attack. We plan to put all code described in this paper into the public domain to maximize reusability of our results.

Hash-function design. This paper achieves new speed records for generalized birthday attacks, and in particular for generalized birthday attacks against the FSB *compression* function. However, generalized birthday attacks are still much more expensive than generic attacks against the FSB *hash* function. "Generic attacks" are attacks that work against any hash function with the same output length.

The FSB designers chose the size of the FSB compression function so that a particular lower bound on the cost of generalized birthday attacks would be safely above the cost of generic attacks. Our results should not be taken as any indication of a security problem in FSB; the actual cost of generalized birthday attacks is very far above the lower bound stated by the FSB designers. It appears that the FSB compression function was designed too conservatively, with an unnecessarily large output length.

FSB was one of the 64 hash functions submitted to NIST's SHA-3 competition, and one of the 51 hash functions selected for the first round. However, FSB was significantly slower than most submissions, and was not one of the 14 hash functions selected for the second round. It would be interesting to explore smaller and thus faster FSB variants that remain secure against generalized birthday attacks.

Organization of the paper. In Section 2 we give a short introduction to Wagner's generalized birthday attack and Bernstein's adaptation of this attack to storage-restricted environments. Section 3 describes the FSB hash function to the extent necessary to understand our attack methodology. In Section 4 we describe our attack strategy which has to match the restricted hard disk space of our computer cluster. Section 5 details the measures we applied to make the attack run as efficiently as possible dealing with the bottlenecks mentioned before. We evaluate the overall cost of our attack in Section 6, and give cost estimates for a similar attack against full-size FSB in Section 7.

Naming conventions. Throughout the paper we will denote list j on level i as $L_{i,j}$. For both, levels and lists we start counting at zero.

Logarithms denoted as lg are logarithms to the base 2.

Additions of list elements or constants used in the algorithm are additions modulo 2.

In units such as GB, TB, PB and EB we will always assume base 1024 instead of 1000. In particular we give 700 GB as the size of a hard disk advertised as 750 GB.

2 Wagner's Generalized Birthday Attack

The generalized birthday problem, given 2^{i-1} lists containing B-bit strings, is to find 2^{i-1} elements — exactly one in each list — whose xor equals 0.

The special case $i = 2$ is the classic birthday problem: given two lists containing B-bit strings, find two elements — exactly one in each list — whose xor equals 0. In other words, find an element of the first list that equals an element of the second list.

This section describes a solution to the generalized birthday problem due to Wagner [12]. Wagner also considered generalizations to operations other than xor, and to the case of k lists when k is not a power of 2.

2.1 The tree algorithm

Wagner's algorithm builds a binary tree as described in this subsection starting from the input lists $L_{0,0}, L_{0,1}, \ldots, L_{0,2^{i-1}-1}$ (see Figure 4.1). The speed and success probability of the algorithm are analyzed under the assumption that each list contains $2^{B/i}$ elements chosen uniformly at random.

On level 0 take the first two lists $L_{0,0}$ and $L_{0,1}$ and compare their list elements on their least significant B/i bits. Given that each list contains about $2^{B/i}$ elements we can expect $2^{B/i}$ pairs of elements which are equal on those least significant B/i bits. We take the xor of both elements on all their B bits and put the xor into a new list $L_{1,0}$. Similarly compare the other lists — always two at a time — and look for elements matching on their least significant B/i bits which are xored and put into new lists. This process of *merging* yields 2^{i-2} lists containing each about $2^{B/i}$ elements which are zero on their least significant B/i bits. This completes level 0.

On level 1 take the first two lists $L_{1,0}$ and $L_{1,1}$ which are the results of merging the lists $L_{0,0}$ and $L_{0,1}$ as well as $L_{0,2}$ and $L_{0,3}$ from level 0. Compare the elements of $L_{1,0}$ and $L_{1,1}$ on their least significant $2B/i$ bits. As a result of the xoring in the previous level, the last B/i bits are already known to be 0, so it suffices to compare the next B/i bits. Since each list on level 1 contains about $2^{B/i}$ elements we again can expect about $2^{B/i}$ elements matching on B/i bits. We build the xor of each pair of matching elements and put it into a new list $L_{2,0}$. Similarly compare the remaining lists on level 1.

Continue in the same way until level $i-2$. On each level j we consider the elements on their least significant $(j+1)B/i$ bits of which jB/i bits are known to be zero as a result of the previous merge. On level $i-2$ we get two lists containing about $2^{B/i}$ elements. The least significant $(i-2)B/i$ bits of each element in both lists are zero. Comparing the elements of both lists on their $2B/i$ remaining bits gives 1 expected match, i.e., one xor equal to zero. Since each element is the

xor of elements from the previous steps this final xor is the xor of 2^{i-1} elements from the original lists and thus a solution to the generalized birthday problem.

2.2 Wagner in memory-restricted environments

A 2007 paper [6] by Bernstein includes two techniques to mount Wagner's attack on computers which do not have enough memory to hold all list entries. Various special cases of the same techniques also appear in a 2005 paper [4] by Augot, Finiasz, and Sendrier and in a 2009 paper [9] by Minder and Sinclair.

Clamping through precomputation. Suppose that there is space for lists of size only 2^b with $b < B/i$. Bernstein suggests to generate $2^{b\cdot(B-ib)}$ entries and only consider those of which the least significant $B - ib$ bits are zero.

We generalize this idea as follows: The least significant $B - ib$ bits can have an arbitrary value, this *clamping value* does not even have to be the same on all lists as long as the *sum* of all clamping values is zero. This will be important if an attack does not produce a collision. We then can simply restart the attack with different clamping values.

Clamping through precomputation may be limited by the maximal number of entries we can generate per list. Furthermore, halving the available storage space increases the precomputation time by a factor of 2^i.

Note that clamping some bits through precomputation might be a good idea even if enough memory is available as we can reduce the amount of data in later steps and thus make those steps more efficient.

After the precomputation step we apply Wagner's tree algorithm to lists containing bit strings of length B' where B' equals B minus the number of clamped bits. For performance evaluation we will only consider lists on level 0 *after* clamping through precomputation and then use B instead of B' for the number of bits in these entries.

Repeating the attack. Another way to mount Wagner's attack in memory-restricted environments is to carry out the whole computation with smaller lists leaving some bits at the end "uncontrolled". We then can deal with the lower success probability by repeatedly running the attack with different clamping values.

In the context of clamping through precomputation we can simply vary the clamping values used during precomputation. If for some reason we cannot clamp any bits through precomputation we can apply the same idea of changing clamping values in an arbitrary merge step of the tree algorithm. Note that any solution to the generalized birthday problem can be found by some choice of clamping values.

Expected number of runs. Wagner's algorithm, without clamping through precomputation, produces an expected number of exactly one collision. However this does not mean that running the algorithm necessarily produces a collision.

In general, the expected number of runs of Wagner's attack is a function of the number of remaining bits in the entries of the two input lists of the last merge step and the number of elements in these lists.

Assume that b bits are clamped on each level and that lists have length 2^b. Then the probability to have at least one collision after running the attack once is

$$P_{\text{success}} = 1 - \left(\frac{2^{B-(i-2)b} - 1}{2^{B-(i-2)b}}\right)^{2^{2b}},$$

and the expected number of runs $E(R)$ is

$$E(R) = \frac{1}{P_{\text{success}}}. \tag{2.1}$$

For larger values of $B - ib$ the expected number of runs is about 2^{B-ib}. We model the total time for the attack as being linear in the amount of data on level 0, i.e.,

$$t_W \in \Theta\left(2^{i-1}2^{B-ib}2^b\right). \tag{2.2}$$

Here 2^{i-1} is the number of lists, 2^{B-ib} is approximately the number of runs, and 2^b is the number of entries per list. Observe that this formula will usually underestimate the real time of the attack by assuming that all computations on subsequent levels are together still linear in the time required for computations on level 0.

Using Pollard iteration. If because of memory restrictions the number of uncontrolled bits is high, it may be more efficient to use a variant of Wagner's attack that uses Pollard iteration [8, Chapter 3, exercises 6 and 7].

Assume that $L_0 = L_1$, $L_2 = L_3$, etc., and that combinations $x_0 + x_1$ with $x_0 = x_1$ are excluded. The output of the generalized birthday attack will then be a collision between two distinct elements of $L_0 + L_2 + \cdots$.

We can instead start with only 2^{i-2} lists $L_0, L_2, \ldots$ and apply the usual Wagner tree algorithm, with a nonzero clamping constant to enforce the condition that $x_0 \neq x_1$. The number of clamped bits before the last merge step is now $(i-3)b$. The last merge step produces 2^{2b} possible values, the smallest of which has an expected number of $2b$ leading zeros, leaving $B - (i-1)b$ uncontrolled.

Think of this computation as a function mapping clamping constants to the final $B-(i-1)b$ uncontrolled bits and apply Pollard iteration to find a collision between the output of two such computations; combination then yields a collision of 2^{i-1} vectors.

As Pollard iteration has square-root running time, the expected number of runs for this variant is $2^{B/2-(i-1)b/2}$, each taking time $2^{i-2}2^b$ (cmp. (2.2)), so the expected running time is

$$t_{PW} \in \Theta\left(2^{i-2}2^{B/2-(i-1)b/2+b}\right). \tag{2.3}$$

The Pollard variant of the attack becomes more efficient than plain Wagner with repeated runs if $B > (i+2)b$.

3 The FSB Hash Function

In this section we briefly describe the construction of the FSB hash function. Since we are going to attack the function we omit details which are necessary for implementing the function but do not influence the attack. The second part of this section gives a rough description of how to apply Wagner's generalized birthday attack to find collisions of the compression function of FSB.

3.1 Details of the FSB hash function

The Fast Syndrome Based hash function (FSB) was introduced by Augot, Finiasz and Sendrier in 2003. See [3], [4], and [2]. The security of FSB's compression function relies on the difficulty of the "Syndrome Decoding Problem" from coding theory.

The FSB hash function processes a message in three steps: First the message is converted by a so-called domain extender into suitable inputs for the compression function which digests the inputs in the second step. In the third and final step the Whirlpool hash function designed by Barreto and Rijmen [5] is applied to the output of the compression function in order to produce the desired length of output.

Our goal in this paper is to investigate the security of the compression function. We do not describe the domain extender, the conversion of the message to inputs for the compression function, or the last step involving Whirlpool.

The compression function. The main parameters of the compression function are called n, r and w. We consider n strings of length r which are chosen uniformly at random and can be written as an $r \times n$ binary matrix H. Note that the matrix H can be seen as the parity check matrix of a binary linear code. The FSB proposal [2] actually specifies a particular structure of H for efficiency; we do not consider attacks exploiting this structure.

An n-bit string of weight w is called *regular* if there is exactly a single 1 in each interval $[(i-1)\frac{n}{w}, i\frac{n}{w}-1]_{1\leq i\leq w}$. We will refer to such an interval as a *block*. The input to the compression function is a regular n-bit string of weight w.

The compression function works as follows. The matrix H is split into w blocks of n/w columns. Each non-zero entry of the input bit string indicates exactly one column in each block. The output of the compression function is an r-bit string which is produced by computing the xor of all the w columns of the matrix H indicated by the input string.

Preimages and collisions. A preimage of an output of length r of one round of the compression function is a regular n-bit string of weight w. A collision occurs if there are $2w$ columns of H — exactly two in each block — which add up to zero.

Finding preimages or collisions means solving two problems coming from coding theory: finding a preimage means solving the Regular Syndrome Decoding problem and finding collisions means solving the so-called 2-regular Null-Syndrome Decoding problem. Both problems were defined and proven to be NP-complete in [4].

Parameters. We follow the notation in [2] and write $\text{FSB}_{\texttt{length}}$ for the version of FSB which produces a hash value of length `length`. Note that the output of the compression function has r bits where r is considerably larger than `length`.

NIST demands hash lengths of 160, 224, 256, 384, and 512 bits, respectively. Therefore the SHA-3 proposal contains five versions of FSB: FSB_{160}, FSB_{224}, FSB_{256}, FSB_{384}, and FSB_{512}. We list the parameters for those versions in Table 7.1.

The proposal also contains FSB_{48}, which is a reduced-size version of FSB and the main attack target in this paper. The binary matrix H for FSB_{48} has dimension $192 \times 3 \cdot 2^{17}$; i.e., r equals 192 and n is $3 \cdot 2^{17}$. In each round a message chunk is converted into a regular $3 \cdot 2^{17}$-bit string of Hamming weight $w = 24$. The matrix H contains 24 blocks of length 2^{14}. Each 1 in the regular bit string indicates exactly one column in a block of the matrix H. The output of the compression function is the xor of those 24 columns.

3.2 Attacking the compression function of FSB_{48}

Coron and Joux pointed out in [7] that Wagner's generalized birthday attack can be used to find preimages and collisions in the compression function of FSB. The following paragraphs present a slightly streamlined version of the attack of [7] in the case of FSB_{48}.

Determining the number of lists for a Wagner attack on FSB_{48}. A collision for FSB_{48} is given by 48 columns of the matrix H which add up to zero; the collision has exactly two columns per block. Each block contains 2^{14} columns and each column is a 192-bit string.

We choose 16 lists to solve this particular 48-sum problem. Each list entry will be the xor of three columns coming from one and a half blocks. This ensures that we do not have any overlaps, i.e., more than two columns coming from one matrix block in the end. We assume that taking sums of the columns of H does not bias the distribution of 192-bit strings. Applying Wagner's attack in a straightforward way means that we need to have at least $2^{\lceil 192/5 \rceil}$ entries per list. By clamping away 39 bits in each step we expect to get at least one collision after one run of the tree algorithm.

Building lists. We build 16 lists containing 192-bit strings each being the xor of three distinct columns of the matrix H. We select each triple of three columns from one and a half blocks of H in the following way:

List $L_{0,0}$ contains the sums of columns i_0, j_0, k_0, where columns i_0 and j_0 come from the first block of 2^{14} columns, and column k_0 is picked from the following block with the restriction that it is taken from the first half of it. Since we cannot have overlapping elements we get about 2^{27} sums of columns i_0 and j_0 coming from the first block. These two columns are then added to all possible columns k_0 coming from the first 2^{13} elements of the second block of the matrix H. In total we get about 2^{40} elements for $L_{0,0}$.

We note that by splitting every second block in half we neglect several solutions of the 48-xor problem. For example, a solution involving two columns from

the first half of the second block cannot be found by this algorithm. We justify our choice by noting that fewer lists would nevertheless require more storage and a longer precomputation phase to build the lists.

The second list $L_{0,1}$ contains sums of columns i_1, j_1, k_1, where column i_1 is picked from the second half of the second block of H and j_1 and k_1 come from the third block of 2^{14} columns. This again yields about 2^{40} elements.

Similarly, we construct the lists $L_{0,2}$, $L_{0,3}$,..., $L_{0,15}$.

For each list we generate more than twice the amount needed for a straightforward attack as explained above. In order to reduce the amount of data for the following steps we note that about $2^{40}/4$ elements are likely to be zero on their least significant two bits. Clamping those two bits away should thus yield a list of 2^{38} bit strings. Note that since we know the least significant two bits of the list elements we can ignore them and regard the list elements as 190-bit strings. Now we expect that a straightforward application of Wagner's attack to 16 lists with about $2^{190/5}$ elements yields a collision after completing the tree algorithm.

Note on complexity in the FSB proposal. The SHA-3 proposal estimates the complexity of Wagner's attack as described above as $2^{r/i}r$ where 2^{i-1} is the number of lists used in the algorithm. This does not take memory into account, and in general is an underestimate of the work required by Wagner's algorithm; i.e., attacks of this type against FSB are more difficult than claimed by the FSB designers.

Note on information-set decoding. The FSB designers say in [2] that Wagner's attack is the fastest known attack for finding preimages, and for finding collisions for small FSB parameters, but that another attack — information-set decoding — is better than Wagner's attack for finding collisions for large FSB parameters.

In general, information-set decoding can be used to find an n-bit string of weight 48 indicating 48 columns of H which add up to zero. Information-set decoding will not take into account that we look for a *regular* n-bit string. The only known way to obtain a regular n-bit string is running the algorithm repeatedly until the output happens to be regular. Thus, the running times given in [2] provide certainly lower bounds for information-set decoding, but in practice they are not likely to hold.

4 Attack Strategy

In this section we will discuss the necessary measures we took to mount the attack on our cluster. We will start with an evaluation of available and required storage.

4.1 How large is a list entry?

The number of bytes required to store one list entry depends on how we represent the entry. We considered four different ways of representing an entry:

Value-only representation. The obvious way of representing a list entry is as a 192-bit string, the xor of columns of the matrix. Bits we already know to be zero of course do not have to be stored, so on each level of the tree the number of bits per entry decreases by the number of bits clamped on the previous level. Ultimately we are not interested in the *value* of the entry — we know already that in a successful attack it will be all-zero at the end — but in the column positions in the matrix that lead to this all-zero value. However, we will show in Section 4.3 that computations only involving the *value* can be useful if the attack has to be run multiple times due to storage restrictions.

Value-and-positions representation. If enough storage is available we can store positions in the matrix alongside the value. Observe that unlike storage requirements for *values* the number of bytes for *positions* increases with increasing levels, and becomes dominant for higher levels.

Compressed positions. Instead of storing full positions we can save storage by only storing, e.g., positions modulo 256. After the attack has successfully finished the full position information can be computed by checking which of the possible positions lead to the appropriate intermediate results on each level.

Dynamic recomputation. If we keep full positions we do not have to store the value at all. Every time we need the value (or parts of it) it can be dynamically recomputed from the positions. In each level the size of a single entry doubles (because the number of positions doubles), the expected number of entries per list remains the same but the number of lists halves, so the total amount of data is the same on each level when using dynamic recomputation. As discussed in Section 3 we have 2^{40} possibilities to choose columns to produce entries of a list, so we can encode the positions on level 0 in 40 bits (5 bytes).

Observe that we can switch between representations during computation if at some level another representation becomes more efficient: We can switch between value-and-position representation to compressed-positions representation and back. We can switch from one of the above to compressed positions and we can switch from any other representation to value-only representation.

4.2 What list size can we handle?

To estimate the storage requirements it is convenient to consider *dynamic recomputation* (storing positions only) because in this case the amount of required storage is constant over all levels and this representation has the smallest memory consumption on level 0.

As described in Section 3.2 we can start with 16 lists of size 2^{38}, each containing bit strings of length $r' = 190$. However, storing 16 lists with 2^{38} entries, each entry encoded in 5 bytes requires 20 TB of storage space.

The computer cluster used for the attack consists of 8 nodes with a storage space of 700 GB each. Hence, we have to adapt our attack to cope with total storage limited to 5.5 TB.

On the first level we have 16 lists and as we need at least 5 bytes per list entry we can handle at most $5.5 \cdot 2^{40}/2^4/5 = 1.1 \times 2^{36}$ entries per list. Some of the disk space is used for operating system and so a straight-forward implementation would use lists of size 2^{36}. First computing one half tree and switching to compressed-positions representation on level 2 would still not allow us to use lists of size 2^{37}.

We can generate at most 2^{40} entries per list so following [6] we could clamp 4 bits during list generation, giving us 2^{36} values for each of the 16 lists. These values have a length of 188 bits represented through 5 bytes holding the positions from the matrix. Clamping 36 bits in each of the 3 steps leaves two lists of length 2^{36} with 80 non-zero bits. According to (2.1) we thus expect to run the attack 256.5 times until we find a collision.

The only way of increasing the list size to 2^{37} and thus reduce the number of runs is to use value-only representation on higher levels.

4.3 The strategy

The main idea of our attack strategy is to distinguish between the task of finding clamping constants that yield a final collision and the task of actually computing the collision.

Finding appropriate clamping constants. This task does not require storing the positions, since we only need to know whether we find a collision with a particular set of clamping constants; we do not need to know which matrix positions give this collision.

Whenever storing the value needs less space we can thus *compress* entries by switching representation from positions to values. As a side effect this speeds up the computations because less data has to be loaded and stored.

Starting from lists $L_{0,0}, \ldots, L_{0,7}$, each containing 2^{37} entries we first compute list $L_{3,0}$ (see Figure 4.1) on 8 nodes. This list has entries with 78 remaining bits each. As we will describe in Section 5, these entries are presorted on hard disk according to 9 bits that do not have to be stored. Another 3 bits are determined by the node holding the data (see also Section 5) so only 66 bits or 9 bytes of each entry have to be stored, yielding a total storage requirement of 1152 GB versus 5120 GB necessary for storing entries in positions-only representation.

We then continue with the computation of list $L_{2,2}$, which has entries of 115 remaining bits. Again 9 of these bits do not have to be stored due to presorting, 3 are determined by the node, so only 103 bits or 13 bytes have to be stored, yielding a storage requirement of 1664 GB instead of 2560 GB for uncompressed entries.

After these lists have been stored persistently on disk, we proceed with the computation of list $L_{2,3}$, then $L_{3,1}$ and finally check whether $L_{4,0}$ contains at least one element. These computations require another 2560 GB.

Therefore total amount of storage sums up to 1152 GB + 1664 GB + 2560 GB = 5376 GB; obviously all data fits onto the hard disk of the 8 nodes.

If a computation with given clamping constants is not successful, we change clamping constants only for the computation of $L_{2,3}$. The lists $L_{3,0}$ and $L_{2,2}$ do not have to be computed again. All combinations of clamping values for lists $L_{0,12}$ to $L_{0,15}$ summing up to 0 are allowed. Therefore there is a large amount of valid clamp bit combinations.

With 37 bits clamped on every level and 3 clamped through precomputation we are left with 4 uncontrolled bits and therefore, according to (2.1), expect 16.5 runs of this algorithm.

Computing the matrix positions of the collision. In case of success we know which clamping constants we can use and we know which value in the lists $L_{3,0}$ and $L_{3,1}$ yields a final collision. Now we can recompute lists $L_{3,0}$ and $L_{3,1}$ without compression to obtain the positions. For this task we decided to store only positions and use dynamic recomputation. On level 0 and level 1 this is the most space-efficient approach and we do not expect a significant speedup from switching to compressed-positions representation on higher levels. In total one half-tree computation requires 5120 GB of storage, hence, they have to be performed one after the other on 8 nodes.

The (re-)computation of lists $L_{3,0}$ and $L_{3,2}$ is an additional time overhead over doing all computation on list positions in the first place. However, this cost is incurred only once, and is amply compensated for by the reduced data volume in previous steps. See Section 5.2.

5 Implementing the Attack

The computation platform for this particular implementation of Wagner's generalized birthday attack on FSB is an eight-node cluster of conventional desktop PCs. Each node has an Intel Core 2 Quad Q6600 CPU with a clock rate of 2.40 GHz and direct fully cached access to 8 GB of RAM. About 700 GB mass storage are provided by a Western Digital SATA hard disk with 20 GB reserved for system and user data. The nodes are connected via switched Gigabit Ethernet using Marvell PCI-E adapter cards.

We chose MPI as communication model for the implementation. This choice has several virtues:

- MPI provides an easy interface to start the application on all nodes and to initialize the communication paths.
- MPI offers synchronous message-based communication primitives.
- MPI is a broadly accepted standard for HPC applications and is provided on a multitude of different platforms.

We decided to use MPICH2 [1] which is an implementation of the MPI 2.0 standard from the University of Chicago. MPICH2 provides an Ethernet-based back end for the communication with remote nodes and a fast shared-memory-based back end for local data exchange.

We implemented two micro-benchmarks to measure hard disk and network throughput. The results of these benchmarks are shown in Figure 5.1. Note

Fig. 4.1. Structure of the attack: in each box the upper line denotes the list, the lower line gives the nodes holding fractions of this list

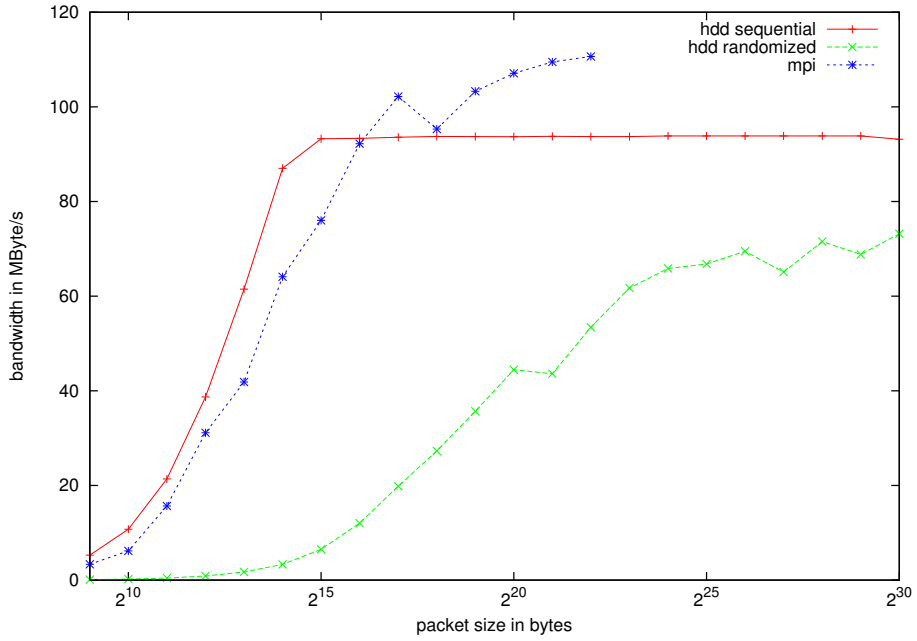


Fig. 5.1. Micro-benchmarks measuring hard disk and network throughput.

that we measure hard disk throughput directly on the device, circumventing the filesystem, to reach peak performance of the hard disk. We measured both sequential and randomized access to the disk.

The rest of this section explains how we parallelized and streamlined Wagner's attack to make the best of the available hardware.

5.1 Parallelization

Most of the time in the attack is spent on determining the right clamping constants. As described in Section 4 this involves computations of several partial trees, e.g., the computation of $L_{3,0}$ from lists $L_{0,0}, \ldots, L_{0,7}$ (half tree) or the computation of $L_{2,2}$ from lists $L_{0,8}, \ldots, L_{0,11}$ (quarter tree). Other computations do not start with lists of level 0, the computation of list $L_{3,1}$ for example is computed from the (previously computed and stored) lists $L_{2,2}$ and $L_{2,3}$.

Lists of level 0 are generated with the current clamping constants. On every level, each list is sorted and afterwards merged with its neighboring list giving the entries for the next level. The sorting and merging is repeated until the final list of the partial tree is computed.

Distributing data over nodes. This algorithm is parallelized by distributing fractions of lists over the nodes in a way that each node can perform sort and merge locally on two lists. On each level of the computation, each node contains fractions of two lists. The lists on level j are split between n nodes according to $\lg(n)$ bits of each value. For example when computing the left half-tree, on level 0, node 0 contains all entries of lists 0 and 1 ending with a zero bit (in the bits

not controlled by initial clamping), and node 1 contains all entries of lists 0 and 1 ending with a one bit.

Therefore, from the view of one node, on each level the fractions of both lists are loaded from hard disk, the entries are sorted and the two lists are merged. The newly generated list is split into its fractions and these fractions are sent over the network to their associated nodes. There the data is received and stored onto the hard disk. The continuous dataflow of this implementation is depicted in Figure 5.2.

Presorting into parts. To be able to perform the sort in memory, incoming data is presorted into one of 512 parts according to the 9 least significant bits of the current sort range. This leads to an expected part size for uncompressed entries of 640 MB (0.625 GB) which can be loaded into main memory at once to be sorted further. The benefit of presorting the entries before storing them is:

1. We can sort a whole fraction, that exceeds the size of the memory, by sorting its presorted parts independently.
2. Two adjacent parts of the two lists on one node (with the same presort-bits) can be merged directly after they are sorted.
3. We can save 9 bits when compressing entries to value-only representation.

Merge. The merge is implemented straightforwardly. If blocks of entries in both lists share the same value then all possible combinations are generated: specifically, if a b-bit string appears in the compared positions in c_1 entries in the first list and c_2 entries in the second list then all c_1c_2 xors appear in the output list.

5.2 Efficient implementation

Cluster computation imposes three main bottlenecks:

- the computational power and memory latency of the CPUs for computation-intensive applications
- limitations of network throughput and latency for communication-intensive applications
- hard disk throughput and latency for data-intensive applications

Wagner's algorithm imposes hard load on all of these components: a large amount of data needs to be sorted, merged and distributed over the nodes occupying as much storage as possible. Therefore, demand for optimization is primarily determined by the slowest component in terms of data throughput; latency generally can be hidden by pipelining and data prefetch.

Finding bottlenecks. Our benchmarks show that, for sufficiently large packets, the performance of the system is mainly bottlenecked by hard disk throughput (cmp. Figure 5.1). Since the throughput of MPI over Gigabit Ethernet is higher than the hard disk throughput for packet sizes larger than 2^{16} bytes and since

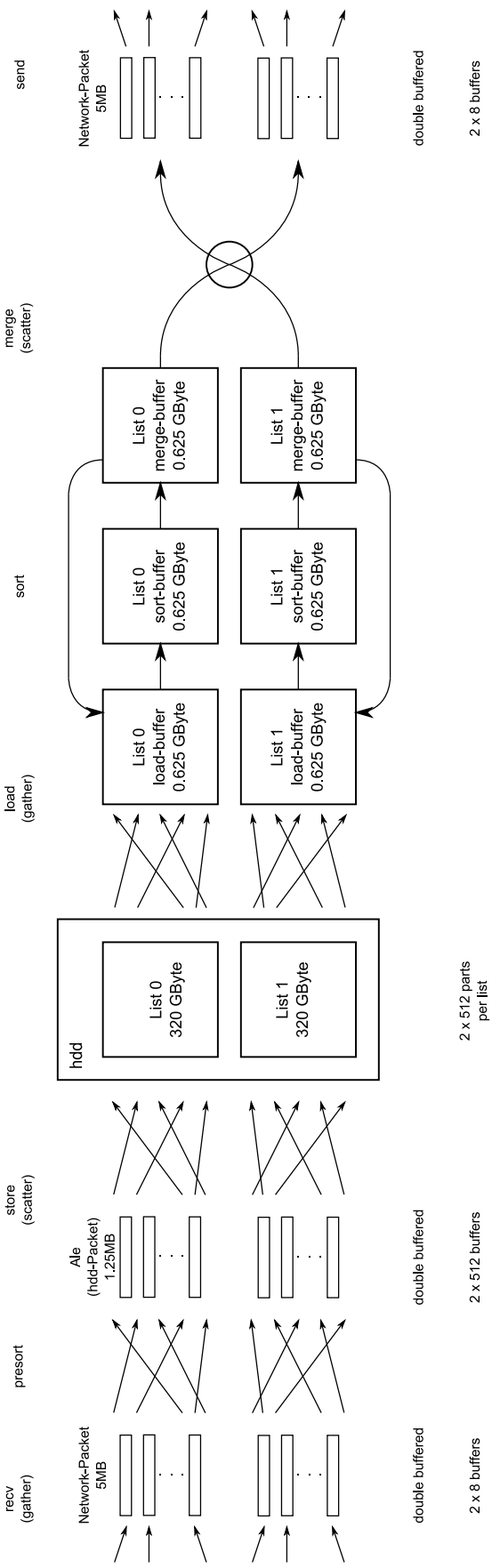


Fig. 5.2. Data flow during the computation within one half-tree

the same amount of data has to be sent that needs to be stored, no performance penalty is expected by the network for this size of packets.

Therefore, our first implementation goal was to design an interface to the hard disk that permits maximum hard disk throughput. The second goal was to optimize the implementation of sort and merge algorithms up to a level where the hard disks are kept busy at peak throughput.

Persistent data storage. Since we do not need any caching-, journaling- or even filing-capabilities of conventional filesystems, we implemented a throughput-optimized filesystem, which we call *AleSystem*. It provides fast and direct access to the hard disk and stores data in portions of *Ales*. Each cluster node has one large unformatted data partition `sda1`, which is directly opened by the AleSystem using native Linux file I/O. Caching is deactivated by using the open flag O_DIRECT: after data has been written, it is not read for a long time and does not benefit from caching. All administrative information is persistently stored as a file in the native Linux filesystem an mapped into the virtual address space of the process. On sequential access, the throughput of the AleSystem reaches about 90 MB/s which is roughly the maximum that the hard disk permits.

Tasks and Threads. Since our cluster nodes are driven by quad-core CPUs, the speed of the computation is primarily based on multi-threaded parallelization. On the one side, the receive-/presort-/store, on the other side, the load-/sort-/merge-/send-tasks are pipelined. At the current state of the implementation, we have several threads for sending/receiving data and for running the AleSystem. The core of the implementation is given by five threads which process the main computation. There are two threads which have the task to presort incoming data (one thread for each list). Furthermore, sorting is parallelized with two threads (one thread for each list) and for the merge task we have one more thread.

Memory layout. Given this task distribution, the size of necessary buffers can be defined. The micro-benchmarks show that bigger buffers generally lead to higher throughput. However, the sum of all buffer sizes is limited by the size of the available RAM. For the list parts we need 6 buffers, each 640MB, adding up to 3.75 GB. We need two times 2×8 network buffers for double-buffered send and receive, which results in 32 network buffers. To presort the entries double-buffered into 512 parts of two lists, we need 2048 ales. The size of network packets must be $2^x \cdot 5, x \geq 3$ bytes because uncompressed entries of the highest level occupy 40 bytes. The size of an ale additionally must be a multiple of 512 bytes due to hardware requirements for DMA access. This gives a size of $2^y \cdot 5, y \geq 9$ bytes for each ale. Therefore, we chose a size of $2^{20} \cdot 5$ bytes = 5 MB for the network packets summing up to 160 MB and a size of $2^{18} \cdot 5$ bytes = 1.25 MB for the ales giving a memory demand of 2.5 GB. Over all, our implementation requires about 6.5 GB of RAM leaving enough space for the operating system and additional data as stack and the administrative data for the AleSystem.

Efficiency and further optimizations. Using our rough splitting of tasks to threads, we reach an average CPU usage of about 60% up to 80% peak.

At the current optimization state, our average hard disk throughput is about 40 MB/s. The hard disk micro-benchmark (see Figure 5.1) shows, that an average throughput between 45 MB/s and 50 MB/s should be feasible for packet sizes of 1.25 MB. Since sorting is the most complex task, we will further parallelize sorting to be able to use 100% of the CPU if the hard disk permits higher data transfer. We expect that further parallelization of the sort task will increase CPU data throughput on sort up to about 50 MB/s. That should suffice for maximum hard disk throughput.

6 Results

The attack is currently running on 8 nodes of the Coding and Cryptography Computer Cluster (CCCC). This section gives timing results of the different steps of the computation and estimates the total time of the attack based on the expected number of runs as given in (2.1).

6.1 Running time

Step one. As described before the first major step is to compute a set of clamping values which leads to a collision. In this first step entries are stored by positions on level 0 and 1 and from level 2 on list entries consist of values.

Computation of list $L_{3,0}$ takes about 32h and list $L_{2,2}$ about 14h, summing up to 46h. These computations need to be done only once.

The time needed to compute list $L_{2,3}$ is about the same as for $L_{2,2}$ (14h), list $L_{3,1}$ takes about 4h and checking for a collision in lists $L_{3,0}$ and $L_{3,1}$ on level 4 about another 3.5h, summing up to about 21.5h. Those steps need to be done on average 16.5 times and and we thus expect them to take about 355h.

Step two. Finally, when we find a collision and compute the correct clamping values, we will recompute the collision with uncompressed lists to find the right columns leading to this collision. Computing lists $L_{3,0}$ and $L_{3,1}$ uncompressed (by storing positions) takes at most 33h each, summing up to 66h.

Overall, finding a collision for the FSB_{48} compression function using our algorithm and cluster takes on average 467h or about 19.5 days.

6.2 Time-storage tradeoffs

As described in Section 4, the main restriction on the attack strategy was the total amount of background storage.

If we had 10496 GB of storage at hand we could handle lists of size 2^{38}, again using the compression techniques described in Section 4. As described in Section 4 this would give us exactly one expected collision in the last merge step and thus reduce the number of required runs to find the right clamping constants from 16.5 to 1.58. With a total storage of 20 TB we could run a

straightforward Wagner attack without compression which would eliminate the need to recompute two half trees at the end.

Increasing the size of the background storage even further would eventually allow to store list entry values alongside the positions and thus eliminate the need for dynamic recomputation. However, the performance of the attack is bottlenecked by hard disk throughput rather than CPU time so we don't expect any improvement through this measure.

On clusters with even less background storage the computation time will (asymptotically) increase by a factor of 16 with each halving of the storage size. For example a cluster with 2688 GB of storage can only handle lists of size 2^{36}. The attack would then require 256.5 computations to find appropriate clamping constants.

Of course the time required for one half-tree computation depends on the amount of data. As long as the performance is bottlenecked mainly by hard disk (or network) throughput the running time is linearly dependent on the amount of data, i.e. a Wagner computation involving 2 half-tree computations with lists of size 2^{38} is about 4.5 times as fast as a Wagner computation involving 18 half-tree computations with lists of size 2^{37}.

7 Scalability Analysis

The attack described in this paper including the variants discussed in Section 6 are much more expensive in terms of time and especially memory than a brute-force attack against the 48-bit hash function FSB_{48}.

This section gives estimates of the power of Wagner's attack against the larger versions of FSB, demonstrating that the FSB design overestimated the power of the attack. Table 7.1 gives the parameters of all FSB hash functions.

A straightforward Wagner attack against FSB_{160} uses 16 lists of size 2^{127} containing elements with 632 bits. The entries of these lists are generated as xors of 10 columns from 5 blocks, yielding 2^{135} possibilities to generate the entries. Precomputation includes clamping of 8 bits. Each entry then requires 135 bits of storage so each list occupies more than 2^{131} bytes. For comparison, the largest currently available storage systems offer a few petabytes (2^{50} bytes) of storage.

To limit the amount of memory we can instead generate, e.g., 32 lists of size 2^{60}, where each list entry is the xor of 5 columns from 2.5 blocks, with 7 bits clamped during precomputation. Each list entry then requires 67 bits of storage.

Clamping 60 bits in each step leaves 273 bits uncontrolled so the Pollard variant of Wagner's algorithm (see Section 2.2) becomes more efficient than the plain attack. This attack generates 16 lists of size 2^{60}, containing entries which are the xor of 5 columns from 5 distinct blocks each. This gives us the possibility to clamp 10 bits through precomputation, leaving $B = 630$ bits for each entry on level 0.

The time required by this attack is approximately 2^{224} (see (2.3)). This is substantially faster than a brute-force collision attack on the compression func-

	n	w	r	Number of lists	Size of lists	Bits per entry	Total storage	Time
FSB_{48}	3×2^{17}	24	192	16	2^{38}	190	$5 \cdot 2^{42}$	$5 \cdot 2^{42}$
FSB_{160}	7×2^{18}	112	896	16	2^{127}	632	$17 \cdot 2^{131}$	$17 \cdot 2^{131}$
				16 (Pollard)	2^{60}	630	$9 \cdot 2^{64}$	$9 \cdot 2^{224}$
FSB_{224}	2^{21}	128	1024	16	2^{177}	884	$24 \cdot 2^{181}$	$24 \cdot 2^{181}$
				16 (Pollard)	2^{60}	858	$13 \cdot 2^{64}$	$13 \cdot 2^{343}$
FSB_{256}	23×2^{16}	184	1472	16	2^{202}	1010	$27 \cdot 2^{206}$	$27 \cdot 2^{206}$
				16 (Pollard)	2^{60}	972	$14 \cdot 2^{64}$	$14 \cdot 2^{386}$
				32 (Pollard)	2^{56}	1024	$18 \cdot 2^{60}$	$18 \cdot 2^{405}$
FSB_{384}	23×2^{16}	184	1472	16	2^{291}	1453	$39 \cdot 2^{295}$	$39 \cdot 2^{295}$
				32 (Pollard)	2^{60}	1467	$9 \cdot 2^{65}$	$18 \cdot 2^{618.5}$
FSB_{512}	31×2^{16}	248	1987	16	2^{393}	1962	$53 \cdot 2^{397}$	$53 \cdot 2^{397}$
				32 (Pollard)	2^{60}	1956	$12 \cdot 2^{65}$	$24 \cdot 2^{863}$

Table 7.1. Parameters of the FSB variants and estimates for the cost of generalized birthday attacks against the compression function. Storage is measured in bytes.

tion, but is clearly much slower than a brute-force collision attack on the hash function, and even slower than a brute-force *preimage* attack on the hash function.

Similar statements hold for the other full-size versions of FSB. Table 7.1 gives rough estimates for the time complexity of Wagner's attack without storage restriction and with storage restricted to a few hundred exabytes (2^{60} entries per list). These estimates only consider the number and size of lists being a power of 2 and the number of bits clamped in each level being the same. The estimates ignore the time complexity of precomputation. Time is computed according to (2.2) and (2.3) with the size of level-0 entries (in bytes) as a constant factor.

Although fine-tuning the attacks might give small speedups compared to the estimates, it is clear that the compression function of FSB is oversized, assuming that Wagner's algorithm in a somewhat memory-restricted environment is the most efficient attack strategy.

References

1. MPICH2 : High-performance and widely portable MPI. `http://www.mcs.anl.gov/research/projects/mpich2/` (accessed 2009-08-18).
2. Daniel Augot, Matthieu Finiasz, Philippe Gaborit, Stéphane Manuel, and Nicolas Sendrier. SHA-3 Proposal: FSB, 2009. `http://www-rocq.inria.fr/secret/CBCrypto/index.php?pg=fsb`.
3. Daniel Augot, Matthieu Finiasz, and Nicolas Sendrier. A fast provably secure cryptographic hash function, 2003. `http://eprint.iacr.org/2003/230`.
4. Daniel Augot, Matthieu Finiasz, and Nicolas Sendrier. A family of fast syndrome based cryptographic hash functions. In Ed Dawson and Serge Vaudenay, editors, *Mycrypt*, volume 3715 of *LNCS*, pages 64–83. Springer, 2005.

5. Paulo S. L. M. Barreto and Vincent Rijmen. The WHIRLPOOL Hashing Function. `http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html`.
6. Daniel J. Bernstein. Better price-performance ratios for generalized birthday attacks. In *Workshop Record of SHARCS'07: Special-purpose Hardware for Attacking Cryptographic Systems*, 2007. `http://cr.yp.to/papers.html#genbday`.
7. Jean-Sébastien Coron and Antoine Joux. Cryptanalysis of a provably secure cryptographic hash function, 2004. `http://eprint.iacr.org/2004/013`.
8. Donald E. Knuth. *The Art of Computer Programming. Vol. 2, Seminumerical Algorithms*. Addison-Wesley Publishing Co., Reading, Mass., third edition, 1997. Addison-Wesley Series in Computer Science and Information Processing.
9. Lorenz Minder and Alistair Sinclair. The extended k-tree algorithm. In Claire Mathieu, editor, *SODA*, pages 586–595. SIAM, 2009.
10. Michael Naehrig, Christiane Peters, and Peter Schwabe. SHA-2 will soon retire. To appear. `http://cryptojedi.org/users/peter/index.shtml#retire`.
11. David Wagner. A generalized birthday problem (extended abstract). In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002: 22nd Annual International Cryptology Conference*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–304. Springer, Berlin, 2002. See also newer version [12]. `http://www.cs.berkeley.edu/~daw/papers/genbday.html`.
12. David Wagner. A generalized birthday problem (extended abstract) (long version), 2002. See also older version [11]. `http://www.cs.berkeley.edu/~daw/papers/genbday.html`.

Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete?

Daniel J. Bernstein *

Department of Computer Science (MC 152)
The University of Illinois at Chicago
Chicago, IL 60607–7053
djb@cr.yp.to

Abstract. Current proposals for special-purpose factorization hardware will become obsolete if large quantum computers are built: the number-field sieve scales much more poorly than Shor's quantum algorithm for factorization. Will *all* special-purpose cryptanalytic hardware become obsolete in a post-quantum world?

A quantum algorithm by Brassard, Høyer, and Tapp has frequently been claimed to reduce the cost of b-bit hash collisions from $2^{b/2}$ to $2^{b/3}$. This paper analyzes the Brassard–Høyer–Tapp algorithm and shows that it has fundamentally worse price-performance ratio than the classical van Oorschot–Wiener hash-collision circuits, even under optimistic assumptions regarding the speed of quantum computers.

Keywords. hash functions, collision-search algorithms, table lookups, parallelization, rho, post-quantum cryptanalysis

1 Introduction

The SHARCS (Special-Purpose Hardware for Attacking Cryptographic Systems) workshops have showcased a wide variety of hardware designs for factorization, brute-force search, and hash collisions. These hardware designs often achieve surprisingly good price-performance ratios and are among the top threats against currently deployed cryptosystems, such as RSA-1024.

Would any of this work be useful for a post-quantum attacker—an attacker equipped with a large quantum computer? The power of today's cryptanalytic hardware is of tremendous current interest, but will the

* Permanent ID of this document: 971550562a76ba87a7b2da14f71ca923. Date of this document: 2009.08.23. This work was supported by the National Science Foundation under grant ITR–0716498.

same hardware designs remain competitive in a world full of quantum computers, assuming that those computers are in fact built?

One might guess that the answer to both of these questions is no: that large quantum computers will become the tool of choice for all large cryptanalytic tasks. Should SHARCS prepare for a transition to a post-quantum SHARCS?

Case study: Factorization. Today's public efforts to build factorization hardware are focused on the number-field sieve. The number-field sieve is conjectured to factor b-bit RSA moduli in time $2^{b^{1/3+o(1)}}$; older algorithms take time $2^{b^{1/2+o(1)}}$ and do not appear to be competitive with the number-field sieve once b is sufficiently large. Detailed analyses show that "sufficiently large" includes a wide range of b's of real-world cryptographic interest, notably $b = 1024$.

The standard advertising for quantum computers is that they can factor much more efficiently than the number-field sieve. Specifically, Shor in [15] and [16] introduced an algorithm to factor a b-bit integer in $b^{\Theta(1)}$ operations on a quantum computer having $b^{\Theta(1)}$ qubits. For a detailed analysis of the number of qubit operations inherent in Shor's algorithm see, e.g., [19].

Simulating this quantum computer on traditional hardware would make it exponentially slower. The goal of quantum-computer engineering is to directly build qubits as physical devices that can efficiently and reliably carry out quantum operations. Note that, thanks to "quantum error correction," perfect reliability is not required; for example, [4, Section 5.3.3.3] shows that an essentially perfect qubit can be simulated by an essentially constant number of 99.99%-reliable qubits.

Assume that this goal is achieved, and that a quantum computer can be built for $b^{\Theta(1)}$ Euros to factor a b-bit integer in $b^{\Theta(1)}$ seconds. This quantum computer will be much more scalable than number-field-sieve hardware, and therefore much more cost-effective than number-field-sieve hardware for large b—including b's of cryptographic interest if the exponents $\Theta(1)$ are reasonably small.

Case study: Preimage search. Similar comments apply to hardware for brute-force search, i.e., hardware to compute preimages.

Consider a function H that can be computed by a straight-line sequence of h bit operations. Assume for simplicity that there is a unique b-bit string x satisfying $H(x) = 0$. Grover in [8] and [9] presented a quantum algorithm to find this x with high probability in approximately $2^{b/2}h$ operations on $\Theta(h)$ qubits. A real-world quantum computer with

similar performance would scale much more effectively than traditional hardware using $2^b h$ operations, and would therefore be much more cost-effective than traditional hardware for large b—again possibly including b's of cryptographic interest. Grover's speedup from $2^b h$ to $2^{b/2} h$ is not as dramatic as Shor's speedup from $2^{b^{1/3+o(1)}}$ to $b^{\Theta(1)}$, but it is still a clear speedup when b is large.

More generally, assume that there are exactly p preimages of 0 under H. Traditional hardware finds a preimage with high probability using approximately $(2^b/p)h$ operations. Boyer, Brassard, Høyer, and Tapp in [5] presented a minor extension of Grover's algorithm to find a preimage with high probability using approximately $(2^{b/2}/p^{1/2})h$ quantum operations on $\Theta(h)$ qubits. It is not necessary for p to be known in advance.

If quantum search is run for only $\epsilon(2^{b/2}/p^{1/2})h$ operations then it has approximately an ϵ^2 chance of success.

Case study: Collision search. The point of this paper is that all known quantum algorithms to find collisions in hash functions are *less* cost-effective than traditional cryptanalytic hardware, even under optimistic assumptions regarding the speed of quantum computers. Quantum computers win for sufficiently large factorizations, and for sufficiently large preimage searches, but they do not win for collision searches.

This conclusion does not depend on the engineering difficulty of building quantum computers; it will remain true even in a world full of quantum computers. This conclusion also does not depend on real-world limits on interesting input sizes. Within the space of known quantum collision algorithms, the most cost-effective algorithms are tantamount to non-quantum algorithms, and it is clear that non-quantum algorithms should be implemented with standard bits rather than with qubits.

In particular, this paper shows that the quantum collision method introduced in [6] by Brassard, Høyer, and Tapp is fundamentally less cost-effective than the collision-search circuits that had been introduced years earlier by van Oorschot and Wiener in [17]. There is a popular myth that the Brassard–Høyer–Tapp algorithm reduces the cost of b-bit hash collisions from $2^{b/2}$ to $2^{b/3}$; this myth rests on a nonsensical notion of cost and is debunked in this paper.

Figures 1.1 and 1.2 summarize the asymptotic speeds of the attack machines considered in this paper. The horizontal axis is machine size, from 2^0 to $2^{b/2}$. The vertical axis is (typical) time to find a collision, from 2^0 to 2^b. Figure 1.1 assumes a realistic two-dimensional communication mesh; Figure 1.2 makes the naive assumption that communication is free.

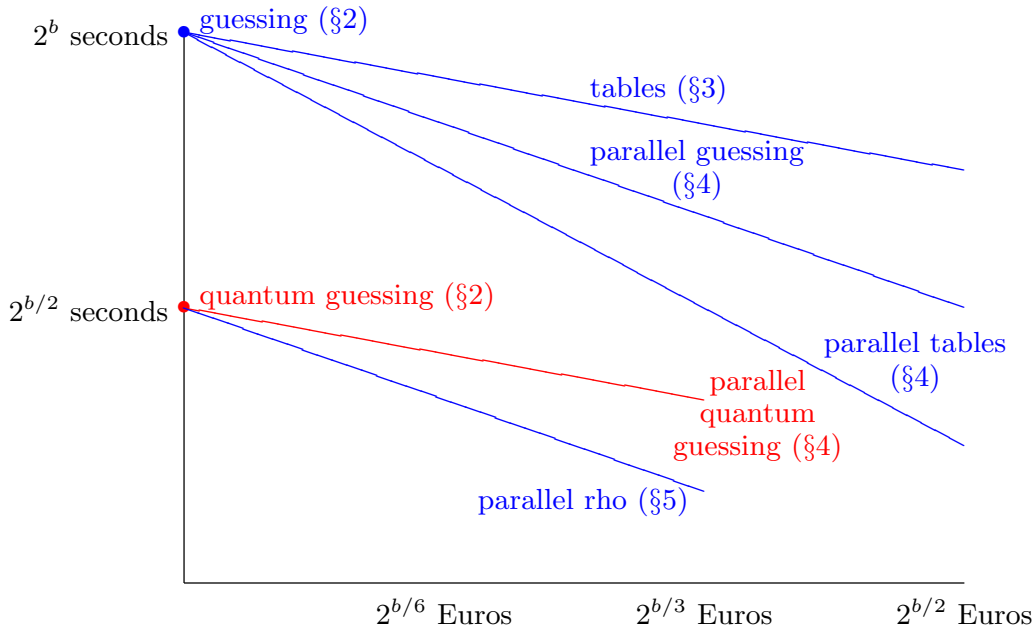


Fig. 1.1. Asymptotic collision-search time assuming realistic communication costs. Parallel rho is 1994 van Oorschot–Wiener [17]. Parallel quantum guessing is 2003 Grover–Rudolph [10].

2 Guessing a collision

A collision in a function H is, by definition, a pair (x, y) such that $x \neq y$ and $H(x) = H(y)$. The simplest way to find a collision is to simply guess a pair (x, y) in the domain of H and see whether it is a collision.

Assume, for concreteness, that H maps $(b + c)$-bit strings to b-bit strings, where $c \geq 1$. Assume that x and y are uniform random $(b+c)$-bit strings. What is the chance that (x, y) is a collision in H? The answer depends on the distribution of output values of H but is guaranteed to be at least $1/2^b - 1/2^{b+c}$. The proof is a standard calculation: say the 2^b output values of H have $p_0, p_1, \ldots, p_{2^b-1}$ preimages respectively, where $p_0 + p_1 + \cdots + p_{2^b-1} = 2^{b+c}$; then the number of collisions (x, y) is

$$\begin{aligned}
&p_0(p_0 - 1) + p_1(p_1 - 1) + \cdots + p_{2^b-1}(p_{2^b-1} - 1) \\
&\quad = p_0^2 + p_1^2 + \cdots + p_{2^b-1}^2 - (p_0 + p_1 + \cdots + p_{2^b-1}) \\
&\quad \geq \frac{(p_0 + p_1 + \cdots + p_{2^b-1})^2}{2^b} - (p_0 + p_1 + \cdots + p_{2^b-1}) \\
&\quad = 2^{b+2c} - 2^{b+c}
\end{aligned}$$

by Cauchy's inequality.

A sequence of N independent guesses succeeds with probability at least $1 - (1 - (1/2^b - 1/2^{b+c}))^N$ and involves at worst $2N$ computations

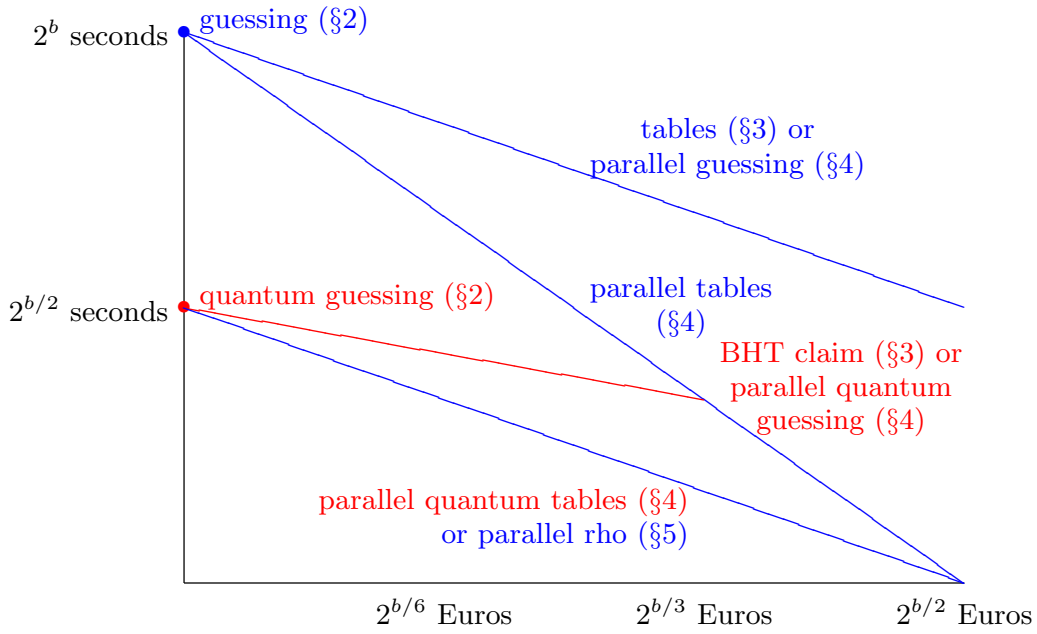


Fig. 1.2. Asymptotic collision-search time assuming free communication. Parallel rho is 1994 van Oorschot–Wiener [17]. "BHT claim" is 1998 Brassard–Høyer–Tapp [6]. Parallel quantum guessing is 2003 Grover–Rudolph [10].

of H. In particular, a sequence of $\lceil 1/(1/2^b - 1/2^{b+c}) \rceil \approx 2^b$ independent guesses succeeds with probability more than $1 - \exp(-1) \approx 0.63$ and involves (at worst) $\approx 2^{b+1}$ computations of H. This attack can be implemented on a very small circuit, typically dominated by the size of a circuit to compute H.

The impact of quantum computers. A collision in H is exactly a preimage of 0 under the $(2b+2c)$-bit-to-1-bit function F defined as follows: $F(x,y) = 0$ if $H(x) = H(y)$ and $x \neq y$; $F(x,y) = 1$ if $H(x) \neq H(y)$ or $x = y$. One can find a preimage by quantum search instead of by guessing. Quantum search uses approximately $2^{b/2}h$ quantum operations on $\Theta(h)$ qubits, where h is the cost of evaluating the function. (To understand the appearance of $2^{b/2}$ here, recall that quantum search finds one out of p b-bit preimages in time approximately $2^{b/2}/p^{1/2}$; now replace b by $2b+2c$, and replace p by $2^{b+2c} - 2^{b+c}$.)

One could summarize this change by claiming that quantum computers reduce collision-search time from 2^b to $2^{b/2}$, saving a factor of $2^{b/2}$. There are two reasons that the actual speedup factor is much smaller. The first reason is that, even in the most optimistic visions of quantum computing, qubits will be larger and slower than bits. The second reason is that there are many other ways to reduce the time far below 2^b, and in fact far below $2^{b/2}$, without quantum computing. There are also

faster quantum collision-search algorithms, but—as shown in subsequent sections of this paper—the non-quantum algorithms are the most cost-effective algorithms known.

3 Table lookups

There is a classic way to use large tables to reduce the number of H evaluations:

- Generate many inputs $x_1, x_2, \ldots, x_M$.
- Compute $H(x_1), H(x_2), \ldots, H(x_M)$, and lexicographically sort the M pairs $(H(x_1), x_1), (H(x_2), x_2), \ldots, (H(x_M), x_M)$.
- Generate many more inputs $y_1, y_2, \ldots, y_N$. After generating y_j, compute $H(y_j)$ and look it up in the sorted list, hoping to find a collision.

This attack has the same effect as searching all MN pairs (x_i, y_j) for collisions $H(x_i) = H(y_j)$. In particular, the attack has a high probability of success if $M \approx N \approx 2^{b/2}$. What makes the attack interesting is that it is faster than considering each pair (x_i, y_j) separately—although it also requires a large attack machine with $M(2b + c)$ bits of memory.

In a naive model of communication, random access to a huge array takes constant time; looking up $H(y_i)$ in the sorted list takes approximately $\lg M$ memory accesses; and sorting in the first place takes approximately $M \lg M$ memory accesses. The table-lookup attack thus takes $M+N$ evaluations of H and additional time approximately $(M+N) \lg M$ for memory access. For example, if $M \approx N \approx 2^{b/2}$, then the attack takes $2^{b/2+1}$ evaluations of H and additional time approximately $2^{b/2}b$ for memory access.

In a realistic two-dimensional model of communication, random access to an M-element array takes time $M^{1/2}$. The table-lookup attack thus takes $M + N$ evaluations of H and additional time approximately $(M + N)M^{1/2} \lg M$ for memory access. For example, if $M \approx N \approx 2^{b/2}$, then the attack takes $2^{b/2+1}$ evaluations of H and additional time approximately $2^{3b/4}b$ for memory access. Memory access is the dominant cost here for typical choices of H.

To summarize, a size-M machine finds collisions in time roughly $2^b/M$ in a naive model of communication, or time roughly $2^b/M^{1/2}$ in a realistic model of communication. If this machine is run for only ϵ times as long then it has approximately an ϵ chance of success.

The impact of quantum computers. Fix $x_1, x_2, \ldots, x_M$. Consider the b-bit-to-1-bit function F defined as follows: $F(y) = 0$ if there is a collision

among $(x_1, y), (x_2, y), \ldots, (x_M, y)$; otherwise $F(y) = 1$. The above attack guesses a preimage of 0 under F.

Brassard, Høyer, and Tapp in [6] propose instead finding a preimage of F by quantum search. They claim in [6, Section 3] that this quantum attack takes "expected time $O((k + \sqrt{N/rk})(T + \log k))$" where "$N$" is the number of hash-function inputs (i.e., 2^{b+c}), "N/r" is the number of hash-function outputs (i.e., 2^b), "k" is the table size (i.e., M), and "T" is the cost of evaluating the hash function (i.e., h). In other words, they state that quantum search finds a preimage of F in expected time $O((M + 2^{b/2}/M^{1/2})(h + \log M))$.

There are several reasons to question the Brassard–Høyer–Tapp claim. Quantum search uses $2^{b/2}/M^{1/2}$ evaluations of F, not merely $2^{b/2}/M^{1/2}$ evaluations of H. Computing $F(y)$ requires not only computing $H(y)$ but also comparing $H(y)$ to $H(x_1), H(x_2), \ldots, H(x_M)$. There are two obstacles to performing these comparisons efficiently when M is large:

- Realistic two-dimensional models of quantum computation, just like realistic models of non-quantum computation, need time $M^{1/2}$ for random access to a table of size M. This $M^{1/2}$ loss is as large as the $M^{1/2}$ speedup claimed by Brassard, Høyer, and Tapp.
- A straight-line circuit to compare $H(y)$ to $H(x_1), H(x_2), \ldots, H(x_M)$ uses $\Theta(Mb)$ bit operations, so a quantum circuit has to use $\Theta(Mb)$ qubit operations. Sorting the table $H(x_1), H(x_2), \ldots, H(x_M)$ does not reduce the size of a *straight-line* comparison circuit, so it does not reduce the number of quantum operations. The underlying problem is that, inside the quantum search, the input to the comparison is a quantum superposition of b-bit strings, so the output depends on all Mb bits in the precomputed table.

There are much simpler quantum collision-search algorithms that reach the speed that Brassard, Høyer, and Tapp claim for their algorithm; see the next section of this paper. Unfortunately, as discussed later, this speed is still not competitive with non-quantum collision hardware.

4 Parallelization

There is a much simpler way to build a machine of size M that finds collisions in time $2^b/M$. The machine consists of M small independent collision-guessing units (each unit being one of the circuits described in Section 2), all running in parallel. This machine does as much work in time T as a single collision-guessing machine would do in time MT. In

particular, it has high probability of finding a collision in time $2^b/M$—not just in a naive model of communication, but in a realistic model of communication. This machine, unlike the table-lookup machine described in the previous section, does not have trouble with communication as M grows.

A more sophisticated size-M machine sorts $H(x_1), H(x_2), \ldots, H(x_M)$ and $H(y_1), H(y_2), \ldots, H(y_M)$ in time $\Theta(bM^{1/2})$ using a two-dimensional mesh-sorting algorithm; see, e.g., [14] and [13]. Computing the H values takes time $\Theta(h)$; the sorting dominates if M is large. This machine has probability approximately $M^2/2^b$ of finding a collision, assuming $M \leq 2^{b/2}$. Repeating the same procedure $2^b/M^2$ times takes time only $\Theta(2^b b/M^{3/2})$ and has high probability of finding a collision.

To summarize, this size-M machine finds collisions in time roughly $2^b/M^{3/2}$ in a realistic model of communication. For example, a machine of size $2^{b/3}$ finds collisions in time roughly $2^{b/2}$. If this machine is run for ϵ times as long then it has approximately an ϵ chance of success.

The impact of quantum computers. Consider a size-M quantum computer that consists of M small independent collision-searching units, all running in parallel. After approximately $2^{b/2}h\epsilon$ quantum operations, each collision-searching unit has approximately an ϵ^2 chance of success, so the entire machine has approximately an $M\epsilon^2$ chance of success. In particular, the machine has a high probability of success after approximately $2^{b/2}h/M^{1/2}$ quantum operations.

For example, a quantum computer of size $2^{b/3}$ can find collisions in time approximately $2^{b/3}$, as claimed in [6]. The fact that mindless parallelism would achieve the same performance as [6] was pointed out by Grover and Rudolph in [10].

One can also try to build the quantum analogue of the more sophisticated size-M machine discussed above. Consider the function F that, given $2Mb$ bits $(x_1, x_2, \ldots, x_M, y_1, y_2, \ldots, y_M)$, outputs 0 if and only if some (x_i, y_j) is a collision in H. Quantum search finds a preimage of F using approximately $2^{b/2}/M$ quantum evaluations of F, saving a factor of $M^{1/2}$ compared to the previous algorithm. A standard two-dimensional mesh-sorting algorithm to compute F can be converted into a two-dimensional mesh-sorting quantum algorithm taking time $\Theta(bM^{1/2})$ on a machine of size M.

This more sophisticated machine might be slightly better than the mindlessly parallel quantum machine if H is expensive, but its overall time is still on the scale of $2^{b/2}/M^{1/2}$. The benefit of considering M inputs together is that M operations produce M^2 collision opportunities, a factor

M better than mindless parallelism—but this speeds up quantum search by only $M^{1/2}$, while communication costs also grow by a factor $M^{1/2}$.

The same idea would be an improvement over [6] and [10] in a three-dimensional model of parallel quantum computation, or in a naive parallel model without communication delays. The function F can be evaluated by a straight-line sequence of essentially bM bit operations (by standard sorting algorithms), and if communication were free then a machine of size M could carry out all of those bit operations in essentially constant time. For example, a quantum computer of size $2^{b/3}$ would be able to find collisions in time approximately $2^{b/6}$ in a naive model.

5 The rho method

Let me review. The best size-M non-quantum machine described so far takes time roughly $2^b/M^{3/2}$ to find a collision: for example, $2^{7b/10}$ if $M = 2^{b/5}$. Quantum search reduces $2^b/M^{3/2}$ to $2^{b/2}/M^{1/2}$: for example, $2^{4b/10}$ if $M = 2^{b/5}$. All of these results are for a realistic model of communication; a naive model would save a factor of $M^{1/2}$.

"But what about the rho method?" the cryptographers are screaming. "What kind of idiot would build a machine of size $2^{b/5}$ to find collisions in time $2^{4b/10}$, when everybody knows how to build a machine of size only $2^{b/10}$ to find collisions just as quickly?"

Recall that the rho method *iterates* the function H. Choose a $(b+c)$-bit string x_0, compute the b-bit string $H(x_0)$, apply an injective padding function π to produce a $(b+c)$-bit string $x_1 = \pi(H(x_0))$, compute $H(x_1)$, compute $x_2 = \pi(H(x_1))$, etc. After approximately $2^{b/2}$ steps one can reasonably expect to find a "distinguished point": a string x_i whose first $b/2$ bits are all 0. (In practice very simple functions π such as "append c zero bits" seem to work for every function H of cryptographic interest, although theorems obviously require more randomness in π.)

Now consider another such sequence $y_0, y_1, \ldots$, again iterated until a distinguished point. There are approximately 2^b pairs (x_i, y_j) before those distinguished points, so one can reasonably expect that those pairs include a collision. Furthermore, if those pairs *do* include a collision, then the distinguished points will be identical; the sequence lengths will then reveal the difference $i - j$, and an easy recomputation of the sequences will find the collision.

More generally, consider a machine with M parallel iterating units, and redefine a "distinguished point" as a string x_i whose first $b/2 - \lceil \lg M \rceil$ bits are all 0. In time approximately $2^{b/2}/M$ this machine will have consid-

ered $\Theta(2^{b/2})$ inputs to H and will have found $\Theta(M)$ distinguished points. The inputs have a good chance of including a collision, and that collision is easily found from a match in the distinguished points. Sorting the distinguished points takes time only $\Theta(M^{1/2})$; this is not a bottleneck for $M \leq 2^{b/3}$.

To summarize, this size-M machine finds collisions in time roughly $2^{b/2}/M$. For example, a machine of size 1 finds collisions in time roughly $2^{b/2}$; a machine of size $2^{b/6}$ finds collisions in time roughly $2^{b/3}$; and a machine of size $2^{b/3}$ finds collisions in time roughly $2^{b/6}$. All of these results hold in a realistic model of communication.

The special case $M = 1$ was introduced by Pollard in [12] in 1975. The general case, finding collisions in time $2^{b/2}/M$, was introduced by van Oorschot and Wiener in [17] in 1994.

The impact of quantum computers. All of the quantum-collision algorithms in the literature are steps backwards from the non-quantum algorithm of [17].

The best time claimed—by Brassard, Høyer, and Tapp in [6], and by Grover and Rudolph in [10]—is $2^{b/2}/M^{1/2}$ on a size-M quantum computer. This is no better than running M parallel copies of Pollard's 1975 method, and is much worse than the van Oorschot–Wiener method.

The previous section of this paper explains how to achieve time $2^{b/2}/M$ on a size-M quantum computer, but only in a naive model allowing free communication. The design has to evaluate H as many times as the van Oorschot–Wiener method, and has to evaluate it on qubits rather than on bits. The lack of iteration in this design might be pleasing for purists who insist on proofs of performance, but this feature is of no practical interest.

Of course, one can also achieve quantum time $2^{b/2}/M$ by viewing the van Oorschot–Wiener algorithm as a quantum algorithm. However, replacing bits with qubits certainly does not save time! There are several obvious ways to combine quantum search with the rho method, but I have not found any such combinations that *improve* performance, and I conjecture that—in a suitable generic model—no such improvements are possible. Quantum search allows N operations to search N^2 possibilities, but the rho method already has the same efficiency.

Many authors have claimed that quantum computers will have an impact on the complexity of hash collisions, reducing time $2^{b/2}$ to time $2^{b/3}$. In fact, time $2^{b/3}$ had already been achieved by non-quantum machines of size just $2^{b/6}$, and smaller time $2^{b/4}$ had already been achieved by non-quantum machines of size $2^{b/4}$. Anyone afraid of quantum hash-

collision algorithms already has much more to fear from non-quantum hash-collision algorithms.

References

1. — (no editor), *Proceedings of the 18th annual ACM symposium on theory of computing*, Association for Computing Machinery, New York, 1986. ISBN 0–89791–193–8. See [14].
2. — (no editor), *2nd ACM conference on computer and communication security, Fairfax, Virginia, November 1994*, Association for Computing Machinery, 1994. See [17].
3. — (no editor), *Proceedings of the twenty-eighth annual ACM symposium on the theory of computing, held in Philadelphia, PA, May 22–24, 1996*, Association for Computing Machinery, 1996. ISBN 0-89791-785-5. MR 97g:68005. See [8].
4. Panos Aliferis, *Level reduction and the quantum threshold theorem* (2007). URL: `http://arxiv.org/abs/quant-ph/0703230`. Citations in this document: §1.
5. Michel Boyer, Gilles Brassard, Peter Høyer, Alain Tapp, *Tight bounds on quantum searching* (1996). URL: `http://arxiv.org/abs/quant-ph/9605034v1`. Citations in this document: §1.
6. Gilles Brassard, Peter Høyer, Alain Tapp, *Quantum cryptanalysis of hash and claw-free functions*, in [11] (1998), 163–169. MR 99g:94013. Citations in this document: §1, §1.2, §1.2, §3, §3, §4, §4, §4, §5.
7. Shafi Goldwasser (editor), *35th annual IEEE symposium on the foundations of computer science. Proceedings of the IEEE symposium held in Santa Fe, NM, November 20–22, 1994*, IEEE, 1994. ISBN 0-8186-6580-7. MR 98h:68008. See [15].
8. Lov K. Grover, *A fast quantum mechanical algorithm for database search*, in [3] (1996), 212–219. MR 1427516. Citations in this document: §1.
9. Lov K. Grover, *Quantum mechanics helps in searching for a needle in a haystack*, Physical Review Letters **79** (1997), 325–328. Citations in this document: §1.
10. Lov K. Grover, Terry Rudolph, *How significant are the known collision and element distinctness quantum algorithms?*, Quantum Information & Computation **4** (2003), 201–206. MR 2005c:81037. URL: `http://arxiv.org/abs/quant-ph/0309123`. Citations in this document: §1.1, §1.1, §1.2, §1.2, §4, §4, §5.
11. Claudio L. Lucchesi, Arnaldo V. Moura (editors), *LATIN'98: theoretical informatics. Proceedings of the 3rd Latin American symposium held in Campinas, April 20–24, 1998*, Lecture Notes in Computer Science, 1380, Springer, 1998. ISBN ISBN 3-540-64275-7. MR 99d:68007. See [6].
12. John M. Pollard, *A Monte Carlo method for factorization*, BIT **15** (1975), 331–334. ISSN 0006–3835. MR 52:13611. URL: `http://cr.yp.to/bib/entries.html#1975/pollard`. Citations in this document: §5.
13. Manfred Schimmler, *Fast sorting on the instruction systolic array*, report 8709, Christian-Albrechts-Universität Kiel, 1987. Citations in this document: §4.
14. Claus P. Schnorr, Adi Shamir, *An optimal sorting algorithm for mesh-connected computers*, in [1] (1986), 255–261. Citations in this document: §4.
15. Peter W. Shor, *Algorithms for quantum computation: discrete logarithms and factoring.*, in [7] (1994), 124–134; see also newer version [16]. MR 1489242. Citations in this document: §1.

16. Peter W. Shor, *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer*, SIAM Journal on Computing **26** (1997), 1484–1509; see also older version [15]. MR MR 98i:11108. Citations in this document: §1.
17. Paul C. van Oorschot, Michael Wiener, *Parallel collision search with application to hash functions and discrete logarithms*, in [2] (1994), 210–218; see also newer version [18]. Citations in this document: §1, §1.1, §1.1, §1.2, §1.2, §5, §5.
18. Paul C. van Oorschot, Michael Wiener, *Parallel collision search with cryptanalytic applications*, Journal of Cryptology **12** (1999), 1–28; see also older version [17]. ISSN 0933–2790. URL: `http://members.rogers.com/paulv/papers/pubs.html`.
19. Christof Zalka, *Fast versions of Shor's quantum factoring algorithm* (1998). URL: `http://arxiv.org/abs/quant-ph/9806084`. Citations in this document: §1.

Shortest Lattice Vector Enumeration on Graphics Cards *

Jens Hermans **[1], Michael Schneider[2], Johannes Buchmann[2], Frederik Vercauteren ***[1], and Bart Preneel[1]

[1] Katholieke Universiteit Leuven
{Jens.Hermans,Frederik.Vercauteren,Bart.Preneel}@esat.kuleuven.be

[2] Technische Universität Darmstadt
{mischnei,buchmann}@cdc.informatik.tu-darmstadt.de

Abstract. In this paper we make a first feasibility analysis for implementing lattice reduction algorithms on GPU using CUDA, a programming framework for NVIDIA graphics cards. The enumeration phase of the BKZ lattice reduction algorithm is chosen as a good candidate for massive parallelization on GPU. Given the nature of the problem we gain large speedups compared to previous CPU implementations. Our implementation saves more than 50% of the time in high lattice dimensions. Among other impacts, this result influences the security of lattice based cryptosystems.

Keywords: Lattice reduction, ENUM, parallelization, graphics cards, CUDA

1 Introduction

Lattice-based cryptosystems are considered to be secure against quantum computer attacks. Therefore those systems are promising alternatives to factoring or discrete logarithm based systems. For those systems already exist quantum computer algorithms that solve the problems efficiently. This will turn out as a problem as soon as large scale quantum computers will be built.

The security of lattice-based schemes is based on the hardness of special lattice problems. Lattice basis reduction helps to determine the actual hardness of those problems in practice. During the last ten years there have been no notable improvements to practical lattice reduction. So people start thinking about using special hardware to speed up the existing algorithms.

* The work described in this report has in part been supported by the Commission of the European Communities through the ICT program under contract ICT-2007-216676. The information in this document is provided as is, and no warranty is given or implied that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

** Research assistant, sponsored by the Fund for Scientific Research - Flanders (FWO).

*** Postdoctoral Fellow of the Fund for Scientific Research - Flanders (FWO).

In 2008, Bernstein et al. use parallelization techniques on graphic cards to solve integer factorization using elliptic curves [BCC+09]. Using NVidia's CUDA parallelization framework, they gained a speed-up of up to 6 compared to computation on a four core CPU. Former applications of GPU parallelization in cryptography were mainly to secret key cryptography, see [CIKL05] for example.

Our Contribution. In this paper we present a parallel version of the enumeration algorithm that searches short vectors in a lattice. We are using the CUDA framework of NVIDIA for implementing an algorithm on graphics cards. Firstly we explain the ideas of how to parallelize enumeration on GPU. Secondly we present some first experimental results. Using the GPU, we reduce the time required for enumeration of a lattice in dimensions bigger than 50 to less than 50%. This result influences the security of lattice based cryptosystems, that is mostly based on the hardness of finding short vectors in a lattice.

The original algorithm for exhaustive search in lattices was presented by Fincke and Pohst [FP83] and Kannan [Kan83]. The algorithm used in practice today is the variant of Schnorr and Euchner [SE91]. In [PS08] Pujol and Stehlé analyze the stability of the enumeration when using floating point arithmetic. A parallel version of the enumeration is not known to date.

2 Preliminaries

A lattice is a discrete subgroup of $\mathbb{R}^d$. It can be represented by a basis matrix $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$. We call $L(\mathbf{B}) = \{\sum_{i=1}^{n} x_i \mathbf{b}_i, x_i \in \mathbb{Z}\}$ the lattice spanned by the basis vectors $\mathbf{b}_i \in \mathbb{R}^d$. The dimension n of a lattice is the number of linear independent vectors in the lattice, i.e. the number of basis vectors. When $n = d$ the lattice is called full dimensional.

The basis of a lattice is not unique. Every unimodular transformation $\mathbf{M}$, i.e. transformation with $\det \mathbf{M} = \pm 1$, turns a basis matrix $\mathbf{B}$ into a second basis $\mathbf{MB}$ of the same lattice.

The determinant of a lattice is defined as $\det(L(\mathbf{B})) = \sqrt{\det(\mathbf{B}^T\mathbf{B})}$. For full dimensional lattices we have $\det(L(\mathbf{B})) = |\det(\mathbf{B})|$. The determinant of a lattice is invariant of the choice of the lattice basis, which follows from the multiplicative property of the determinant and the fact that basis transformations have determinant 1.

The length of the shortest vector of a lattice $L(\mathbf{B})$ is denoted $\lambda_1(L(\mathbf{B}))$ or in short λ_1 if the lattice is uniquely determined.

The Gram-Schmidt orthogonalization computes an orthogonal projection of a basis. It is an efficient algorithm that outputs $\mathbf{B}^* = [\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*]$ and $\mu_{i,j}$ such that $\mathbf{B} = \mathbf{B}^* \cdot [\mu_{i,j}]$, where $[\mu_{i,j}]$ is an upper triangular matrix consisting of the Gram-Schmidt coefficients $\mu_{i,j}$ for $1 \leq j \leq i \leq n$. The orthogonalized matrix $\mathbf{B}^*$ is not necessarily a basis of the lattice.

2.1 Lattice Basis Reduction

Problems. Some lattice bases are more useful than others. The goal of lattice basis reduction (or in short lattice reduction) is to find a basis consisting of short and almost orthogonal lattice vectors. More exactly, we can define some (hard) problems on lattices. The most important one is the *shortest vector problem* (SVP), which is looking for a vector $\mathbf{v} \in L \setminus \{\mathbf{0}\}$ with $\|\mathbf{v}\| = \lambda_1(L(\mathbf{B}))$. In most cases, the Euclidean norm $\|\cdot\|_2$ is considered. As the SVP is $\mathcal{NP}$-hard (at least under randomized reductions) people consider the approximate version γ-SVP, that is looking for a vector $\mathbf{v} \in L \setminus \{\mathbf{0}\}$ with $\|\mathbf{v}\| \leq \gamma \cdot \lambda_1(L(\mathbf{B}))$.

Other important problems like the *closest vector problem* (CVP) that searches for a nearest lattice vector to a given point in space, its approximation variant γ-CVP, or the *shortest basis problem* (SBP) are listed and described in detail in [ECR].

Algorithms. One of the main contributions to lattice reduction was the work of Lenstra, Lenstra, and Lovász in 1982 [LLL82], where they introduced the LLL algorithm, which was the first polynomial time algorithm to solve the approximate shortest vector problem in higher dimensions. Another famous contribution is the BKZ block algorithm of Schnorr and Euchner [SE91]. In practice, this is the algorithm that gives the best solution to lattice reduction so far. Their paper [SE91] also introduces the enumeration algorithm (ENUM), which practically is the fastest algorithm to solve the exact shortest vector problem using complete enumeration of all lattice vectors. It is used as a black box in the BKZ algorithm. The enumeration algorithm organizes linear combinations of the basis vectors in a search tree and performs a depth first search above the tree. In the same paper, Schnorr and Euchner explain the idea of deep inserting vectors into a basis during LLL reduction. This approach results in shorter vectors at the expense of the algorithms runtime.

Other promising algorithm variants were presented by Schnorr [Sch03], Nguyen and Stehlé [NS05], and Gama and Nguyen [GN08a]. The variant of [NS05] is implemented in the `fpLLL` library of [Ste], which is the fastest public implementation of ENUM algorithms. In [GN08b] Gama and Nguyen compare the NTL implementation [Sho] of floating point LLL, the deep insertion variant of LLL and the BKZ algorithm. It is the first comprehensive comparison of lattice basis reduction algorithms and helps understanding their practical behavior. Koy introduced the notion of a primal-dual reduction in [Koy04]. Schnorr [Sch03] and Ludwig [BL06] deal with random sampling reduction. Both are a bit different concepts of lattice reduction, where primal-dual reduction uses the dual of a lattice for reducing and random sampling combines LLL-like algorithms with an exhaustive point search in a set of lattice vectors that is likely to contain short vectors.

The parallelization of lattice reduction was considered in [Vil92,HT93,RV92]. These papers present parallel versions for n and n^2 processors, where n is the lattice dimension. In [Jou93] the parallel LLL of Villard [Vil92] is combined with the floating point ideas of [SE91]. In [Wet98] the authors present a blockwise

generalization of Villards algorithm. Backes and Wetzel worked out a parallel variant of the LLL algorithm for multi-core CPU architectures [BW09]. All previous parallel algorithms handle the LLL algorithm, but to our knowledge there exists no parallel version of the enumeration algorithm. Furthermore, for GPU parallelization there is no work done.

Applications. Lattice reduction has applications in cryptography as well as in cryptanalysis. The foundation of some promising cryptographic primitives is based on the hardness of lattice problems. Lattice reduction helps determining the practical hardness of those problems and is a basis for real world application of those hash functions, signatures, and encryption schemes. Well known examples are the SWIFFT hash functions of Lyubashevsky et al. [LMPR08], the signature scheme of Gentry, Peikert and Vaikuntanathan [GPV08], or the encryption schemes of [AD97,Pei09,SSTX09]. The NTRU [HPS98,otCC09] and GGH [GGH97] schemes do not provide a security proof, but the most promising attacks are also lattice based ones.

In cryptanalysis, there are further applications of lattice basis reduction. Not only lattice-based systems can be broken using this technique. There are attacks on RSA and similar systems, using lattice reduction to find roots of certain polynomials [CNS99,DN00]. Low density knapsack cryptosystems where successfully attacked with lattice reduction [LO85]. Other applications of lattice basis reduction are factoring numbers and computing discrete logarithms using diophantine approximations [Sch91]. In Operations Research, or generally speaking, discrete optimization, lattice reduction can be used to solve linear integer programs [Len83].

2.2 Programming Graphics Cards

Graphical Processing Units (GPUs) is hardware that is specifically designed to perform a massive number of specific graphical operations in parallel. The introduction of platforms like CUDA by NVidia [Nvi07a] or CTM by ATI [AMD06], that make it easier to run custom programs instead of limited graphical operations on a GPU, has been the major breakthrough for the GPU as a general computing platform. The introduction of integer and bit arithmetic also broadened the scope to cryptographic applications.

Applications. Many general mathematical packages are available for GPU, like the BLAS library [NVI07b] that supports basic linear algebra operations.

An obvious application in the area of cryptography is brute force searching using multiple parallel threads on the GPU. There are also implementations of AES [CIKL05] [Man07] [HW07] and RSA [MPS07] [SG08], [Fle07] available. These implementations can also be used (partially) in cryptanalysis. Integer factorization on elliptic curves has been implemented in [BCC$^+$09]. However, to date, no applications based on lattices are available for GPU.

Programming Model. For the work in this paper the CUDA platform will be used. The GPUs from the Tesla range, which support CUDA, are composed of several multiprocessors, each containing a small number of scalar processors. For the programmer this underlying hardware model is hidden by the concept of SIMT-programming: Single Instruction, Multiple Thread. The basic idea is that the code for a single thread is written, which is then uploaded to the device and executed in parallel by multiple threads.

The threads are organized in multidimensional arrays, called blocks. All blocks are again put in a multidimensional array, called the *grid*. When executing a program (a grid), threads are scheduled in groups of 32 threads, called *warps*. Within a warp threads should not diverge, as otherwise the execution of the warp is serialized.

Memory Model. The Tesla GPUs provide multiple levels of memory: registers, shared memory, global memory, texture and constant memory. Registers and shared memory are on chip and close to the multiprocessor and can be accessed with low latency. The number of registers and shared memory is limited, since the number available for one multiprocessor must be shared among all threads in a single block.

Global memory is off-chip and is not cached. As such, access to global memory can slow down the computations drastically, so several strategies for speeding up memory access should be considered (besides the general strategy of avoiding global memory access). By coalescing memory access, e.g. loading the same memory address or a consecutive block of memory from multiple threads, the delay is reduced, since a coalesced memory access has the same cost as a single random memory access. By launching a large number of blocks the latency introduced by memory loading can also be hidden, since other blocks can be scheduled in the meantime.

The constant and texture memory are cached and can be used for specific types of data or special access patterns.

Instruction Set. Modern GPUs provide the full range of (32 and) 64 bit floating point, integer and bit operations. Addition and multiplication are fast, other operations can, depending on the type, be much slower. There is no point in using other than 32 or 64 bit numbers, since smaller types are always cast to larger types. Most GPUs have a specialized FMAD instruction, which performs a floating point multiplication followed by an addition at the cost of only a single operation. This instruction can be used during the BKZ enumeration.

One problem that occurs on GPU's is the fact that today GPU's are not able to deal with higher precision than 64 bit floating point numbers. For lattice reduction, sometimes higher bit sizes are required to guarantee the correct termination of the algorithms. For an n-dimensional lattice, using the floating point LLL algorithm of [LLL82], one requires a precision of $\mathcal{O}(n \log B)$ bits, where B is an upper bound for the d-dimensional vectors [NS05]. For the L^2 algorithm of [NS05], the required bit size is $\mathcal{O}(n \log_2 3)$, which is independent of the entry size. For more details on the floating point LLL analysis see [NS05] and [NS06].

In [PS08] the authors state that for enumeration algorithms double precision is suitable up to dimension 90, which is beyond the dimensions that are practical today. Therefore enumeration should be possible on actual graphics cards, whereas LLL-like algorithms will gain some problems and require some multi-precision framework.

3 Parallel Enumeration on GPU

In this section we present our algorithm for shortest vector enumeration in lattices. Firstly we briefly explain the ENUM algorithm of Schnorr and Euchner [SE91]. Secondly we explain the ideas of GPU parallelization of the algorithm before presenting our new algorithm.

The enumeration algorithms used are variants of those in [Kan83] and [FP83]. Schnorr and Euchner [SE91] improve the enumeration technique. Their algorithm is the fastest one today and also the one used in the `NTL` [Sho] and `fpLLL` [Ste] library. Therefore we have chosen this algorithm as basis for our parallel algorithm.

The ENUM algorithm enumerates over all linear combinations $[x_1, \ldots, x_n] \in \mathbb{Z}^n$ which generate a vector $\mathbf{v} = \sum_{i=1}^{n} x_i \mathbf{b}_i$. Those linear combinations are organized in a tree structure. Leafs of the tree contain full linear combinations, whereas inner nodes contain partly filled vectors. The search for the tree leaf that determines the shortest lattice vector is performed in a depth first search order. The most important part of the enumeration is cutting off parts of the tree, i.e. the strategy which subtrees are explored and which ones cannot lead to a shorter vector. Usually, as initial bound for the length of the shortest vector, one uses the norm of the first basis vector. For a more detailed description we refer to [PS08].

3.1 Multi-Thread Enumeration

Roughly speaking, the parallel enumeration works as follows. The search tree of combinations that is explored in the enumeration algorithm can be split at a high level, distributing subtrees among several threads. Each thread then runs an enumeration algorithm, keeping the first coefficients fixed. These threads can run independently of the others, which limits communication needed between threads. The top level enumeration is performed on CPU and outputs start vectors for the single GPU threads.

When the number of postponed subtrees is higher than the number of threads that we can start in parallel, then we copy the start vectors to the GPU and let it enumerate the subtrees. After all threads have finished enumerating their subtrees we proceed in the same manner: caching start vectors on CPU and start enumeration on GPU. Figure 1 illustrates this approach. The variable α defines the region where the initial enumeration is performed. The subtrees where GPU threads work are also depicted in Figure 1.

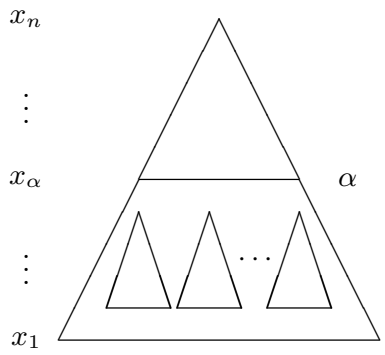


Fig. 1. Illustration of the algorithm flow. The top part is enumerated on CPU, the lower subtrees are explored in parallel on GPU.

If a GPU subtree enumeration finds a new optimal vector, it writes back the coordinates and norm of this vector to the main memory. The other GPU threads will directly receive the new norm, which will allow them to cut away more parts of the subtree.

Early Termination. The computation power of the GPU is used best when as many threads as possible are working at the same time. Therefore the number of enumeration steps that can be performed on GPU is bounded by a value **S** (for each subtree). When **S** is exceeded by one of the GPU subtree enumerations, this subtree enumeration stops computing and writes back it current state to the main memory. This state can be used later on, to restart the enumeration at exactly the same position it stopped. The early termination after **S** enumeration steps is needed because the subtree enumerations have different lengths. The early termination ensures that the GPU is always running a high number of enumerations in parallel and isn't stalled by a small number of long-running enumerations. This small number long-running threads are the cause of thread divergence, which causes performance degradation.

Because of early termination some of the subtree enumerations are not finished after a single launch of the GPU enumeration. This is the main reason why the entire algorithm is iterated several times. At each iteration the GPU launches a mix of enumerations: new subtrees from the top enumeration and subtrees that were not finished in one of the previous GPU launches (because **S** was exceeded).

3.2 The Iterated Parallel ENUM Algorithm

Algorithm 1 shows the high-level layout of the GPU enumeration algorithm. Details concerning the updating of the bound A, as well as the write-back of newly discovered optimal vectors have been omitted. The actual enumeration is also not shown: it is part of several subroutines which are called from the main algorithm.

Algorithm 1: High-level GPU ENUM Algorithm

Input: $\mathbf{b}_i$, A, α, n

1 Compute the Gram-Schmidt decomposition of $\mathbf{b}_i$
2 **while** *true* **do**
3 $\quad S = \{(\mathbf{x}_i, \mathbf{\Delta x}_i, \mathbf{\Delta^2 x}_i, l_i = \alpha, s_i = 0)\}_i \leftarrow$ Top enum: generate at most NUMSTARTPOINTS $- \#T$ vectors
4 $\quad R = \{(\bar{\mathbf{x}}_i, \mathbf{\Delta x}_i, \mathbf{\Delta^2 x}_i, l_i, s_i)\}_i \leftarrow$ GPU enumeration, starting from $S \cup T$
5 $\quad T \leftarrow \{R_i : s_i \geq \mathbf{S}\}$
6 $\quad$ **if** $\#T <$ CPUTHRESHOLD **then**
7 $\quad\quad$ Enumerate the starting points in T on the CPU.
8 $\quad\quad$ Stop
9 $\quad$ **end**
10 **end**

Output: $(x_1, \ldots, x_n)$ with $\left\|\sum_{i=1}^{n} x_i \mathbf{b}_i\right\| = \lambda_1(L)$

The whole process of launching a grid of GPU threads is iterated several times (line 2), until the whole search tree has been enumerated either on GPU or CPU.

In line 3, the top of the search tree is enumerated, to generate a set S of starting vectors $\mathbf{x}_i$ for which enumeration should be started at level α. More detailed, the top enumeration in the region between α and n outputs distinct vectors

$$\mathbf{x}_i = [0, \ldots, 0, x_\alpha, \ldots, x_n] \qquad \text{for } i = 1 \ldots \text{NUMSTARTPOINTS} - \#T\,.$$

The top enumeration will stop automatically if a sufficient amount of vectors from the top of the tree have been enumerated. The rest of the top of the tree is enumerated in the following iterations of the algorithm.

Line 4 performs the actual GPU enumeration. In each iteration, a set of starting vectors and starting levels $\{\mathbf{x}_i, l_i\}$ is uploaded to the GPU. These starting vectors can be either vectors generated by the top enumeration in the region between α and n (in which case $l_i = \alpha$) or the vectors (and levels) written back by the GPU when exceeding $\mathbf{S}$, so that the enumeration will continue. In total NUMSTARTPOINTS vectors (a mix of new and old vectors) are uploaded at each iteration. For each starting vector $\mathbf{x}_i$ (with associated starting level l_i) the GPU outputs a vector

$$\bar{\mathbf{x}}_i = [\bar{x}_1, \ldots, \bar{x}_{\alpha-1}, x_\alpha, \ldots, x_n] \qquad \text{for } i = 1 \ldots \text{NUMSTARTPOINTS}\,,$$

(which describes the current position in the search tree), the current level l_i, the number of enumeration steps s_i performed and also part of the internal state of the enumeration. This state $\{\bar{\mathbf{x}}_i, \mathbf{\Delta x}_i, \mathbf{\Delta^2 x}_i, l_i\}$ can be used to continue the enumeration later on. The vectors $\mathbf{\Delta x}_i$ and $\mathbf{\Delta^2 x}_i$ are used in the enumeration to generate the zig-zag pattern and are part of the internal state of the enumeration [SE91]. This state is added to the output to be able to efficiently restart the enumeration at the point it was terminated.

Line 5 will select the resulting vectors from the GPU enumeration that were terminated because of reaching the bound **S**. These will be added to the set T of *leftover* vectors, which will be relaunched in the next iteration of the algorithm. If the set of leftover vectors is too small to get an efficient GPU enumeration, the CPU takes over and finishes of the last part of the enumeration. This final part only takes limited time.

GPU Threads. The number of starting points NUMSTARTPOINTS is higher than the number of threads (NUMTHREADS) that are launched on the GPU. Each thread will pick a new starting vector from the stack, after it finished enumerating his starting point. This is done until all NUMSTARTPOINTS starting vectors are enumerated or stopped after reaching **S** enum steps. The reason for this is closely connected to the early termination feature discussed before. Since the subtree enumerations have different lengths, a thread should be able to continue working even if the subtree enumeration was small. In our experiments, NUMSTARTPOINTS was around 20-30 times higher than NUMTHREADS, which means that on average every GPU thread enumerated 20-30 subtrees in each iteration.

4 Experimental Results

In this section we present some preliminary results of our CUDA implementation of our algorithm. For comparison we used the highly optimized ENUM algorithm of the `fpLLL` library in version 3.0.11 of Stehlé and Pujol from [Ste][3]. The CUDA program was compiled using `nvcc`, for the CPU programs we used `g++` with compiler flag `-O2`. The tests were run on a Intel Core2 Extreme CPU X9650 (using one single core) running at 3 GHz, and a NVIDIA GTX 280 graphics card. We run up to 100000 threads in parallel on the GPU.

Our input lattices are LLL reduced with $\delta = 0.99$. We chose random lattices following the construction principle of [GM03] with bit size of the entries of $10 \cdot n$. Both algorithms output the same coefficient vectors and therefore a lattice vector with shortest possible length. Table 1 and Figure 2 illustrate the experimental

n	45	48	50	52	54
`fpLLL`	18.3s	139s	277s	2483s	6960s
`CUDA`	20.2s	92s	133s	959s	2599s
	110%	66%	48%	39%	37%

Table 1. Average time needed for enumeration of lattices in each dimension n.

results. They compare our CUDA implementation with the ENUM of `fpLLL` (run `fpLLL` with parameter *-a svp*). The figure shows the runtimes of both algorithms when applied to five different lattices of each dimension. One can notice that in

[3] Timings of the NTL version of ENUM can be found in [GN08b].

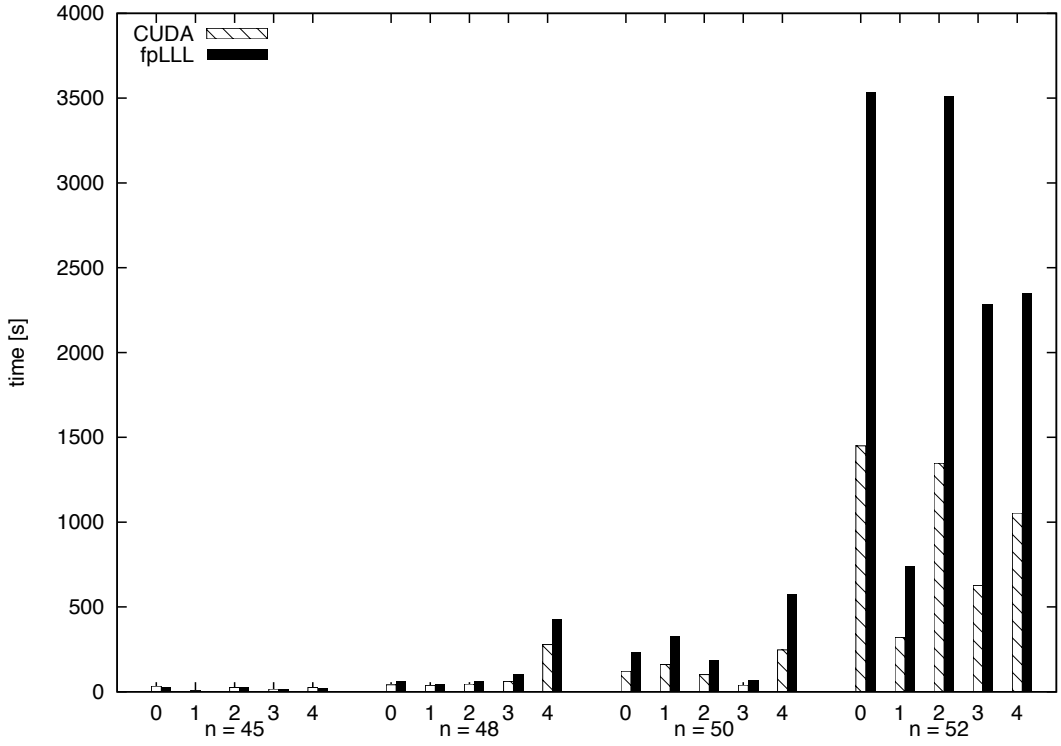


Fig. 2. Timings for enumeration. The graph shows the time needed for enumerating five different random lattices in each dimension n.

dimension above 48, our CUDA implementation always outperforms the `fpLLL` implementation. In lower dimensions, the initialization of the GPU requires more time than the enumeration itself, therefore the complete algorithm lasts longer than on CPU. Table 1 shows the average value over all five lattices in each dimension. Again one notices that in dimension 45, the GPU algorithm is a bit slower. It demonstrates its strength in dimensions above 48, where the time goes down to 48% in dimension 50 and down to 37% in dimension 54. Therefore we can state that the GPU algorithm gains big speedups in dimensions bigger than 50, which are the interesting ones in practice.

On the GPU a throughput up to 100 million enumeration steps per second is achieved, while similar experiments on CPU only yielded 25 million steps per second.

Further Work. Further improvements are possible using multiple CPU cores. Our implementation only uses one CPU core for the top enumeration, whereas additional cores are kept aside for the computations. When GPU starts enumeration it would be possible to start threads on different CPU cores as well. We expect a speedup of two compared to our actual implementation using this idea.

A second opportunity for further speedups is the tweaking of the several parameters (like $\mathbf{S}$, α, NUMTHREADS, NUMSTARTPOINTS). Exhaustive testing will improve the sets of parameters for GPU enumeration for each lattice dimension.

It is possible to start enumeration using a shorter starting value than the first basis vectors norm. The Gaussian heuristic can be used to predict the norm of the shortest basis vector λ_1. This can lead to enormous speed ups in the algorithm. We did not include this improvement into our algorithm so far to get comparable results to `fpLLL`.

Acknowledgments

We thank the anonymous referees for their valuable comments. We thank Özgür Dagdelen for creating some of the initial ideas of parallelizing lattice enumeration and Benjamin Milde for the nice discussions and help with the implementation.

Finally we would like to thank EcryptII[4] and CASED[5] for providing the funding for the visits during which this work was prepared.

References

[AD97] Miklós Ajtai and Cynthia Dwork. A public-key cryptosystem with worst-case/average-case equivalence. In *Proceedings of the Annual Symposium on the Theory of Computing (STOC) 1997*, pages 284–293, 1997.

[AMD06] Advanced Micro Devices. ATI CTM Guide. Technical report, 2006.

[BCC+09] Daniel J. Bernstein, Tien-Ren Chen, Chen-Mou Cheng, Tanja Lange, and Bo-Yin Yang. ECM on graphics cards. In *Advances in Cryptology — Eurocrypt 2009*, Lecture Notes in Computer Science, pages 483–501, 2009.

[BL06] Johannes Buchmann and Christoph Ludwig. Practical lattice basis sampling reduction. In F. Hess, S. Pauli, and M. Pohst, editors, *Proceedings of ANTS VII*, volume 4076 of *Lecture Notes in Computer Science*, pages 222–237. Springer-Verlag, 2006.

[BW09] Werner Backes and Susanne Wetzel. Parallel lattice basis reduction using a multi-threaded Schnorr-Euchner LLL algorithm. In *15th International European Conference on Parallel and Distributed Computing (Euro-Par)*, 2009.

[CIKL05] Debra L. Cook, John Ioannidis, Angelos D. Keromytis, and Jake Luck. Cryptographics: Secret key cryptography using graphics cards. In *CT-RSA*, pages 334–350, 2005.

[CNS99] Christophe Coupé, Phong Q. Nguyen, and Jacques Stern. The effectiveness of lattice attacks against low exponent RSA. In *Public-Key Cryptography (PKC)*, volume 1560 of *Lecture Notes in Computer Science*, pages 204–218. Springer-Verlag, 1999.

[DN00] Glenn Durfee and Phong Q. Nguyen. Cryptanalysis of the RSA schemes with short secret exponent from Asiacrypt '99. In *Advances in Cryptology — Asiacrypt 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 14–29. Springer-Verlag, 2000.

[ECR] ECRYPT II. Wiki - Hard problems in cryptography - lattices. http://www.ecrypt.eu.org/wiki/index.php/Lattices.

[4] http://www.ecrypt.eu.org/

[5] http://www.cased.de

[Fle07] S. Fleissner. GPU-Accelerated Montgomery Exponentiation. *Lecture Notes in Computer Science*, 4487:213, 2007.

[FP83] U. Fincke and Michael Pohst. A procedure for determining algebraic integers of given norm. In *European Computer Algebra Conference*, volume 162 of *Lecture Notes in Computer Science*, pages 194–202. Springer-Verlag, 1983.

[GGH97] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Public-key cryptosystems from lattice reduction problems. In *Advances in Cryptology — Crypto 1997*, Lecture Notes in Computer Science, pages 112–131. Springer-Verlag, 1997.

[GM03] Daniel Goldstein and Andrew Mayer. On the equidistribution of hecke points. *Forum Mathematicum 2003, 15:2*, pages 165–189, 2003.

[GN08a] Nicolas Gama and Phong Q. Nguyen. Finding short lattice vectors within mordell's inequality. In *Proceedings of the Annual Symposium on the Theory of Computing (STOC) 2008*, pages 207–216. ACM Press, 2008.

[GN08b] Nicolas Gama and Phong Q. Nguyen. Predicting lattice reduction. In *Advances in Cryptology — Eurocrypt 2008*, pages 31–51, 2008.

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *Proceedings of the Annual Symposium on the Theory of Computing (STOC) 2008*, pages 197–206. ACM Press, 2008.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In *Algorithmic Number Theory Symposium — ANTS 1998*, pages 267–288, 1998.

[HT93] Christian Heckler and Lothar Thiele. A parallel lattice basis reduction for mesh-connected processor arrays and parallel complexity. In *SPDP*, pages 400–407, 1993.

[HW07] O. Harrison and J. Waldron. AES Encryption Implementation and Analysis on Commodity Graphics Processing Units. *Lecture Notes in Computer Science*, 4727:209, 2007.

[Jou93] Antoine Joux. A fast parallel lattice reduction algorithm. In *Proceedings of the Second Gauss Symposium*, pages 1–15, 1993.

[Kan83] Ravi Kannan. Improved algorithms for integer programming and related lattice problems. In *Proceedings of the Annual Symposium on the Theory of Computing (STOC)1983*, pages 193–206. ACM Press, 1983.

[Koy04] Henrik Koy. Primale-duale Segment-Reduktion. http://www.mi.informatik.uni-frankfurt.de/research/papers.html, 2004.

[Len83] H.W.jun. Lenstra. Integer programming with a fixed number of variables. *Math. Oper. Res.*, 8:538–548, 1983.

[LLL82] Arjen Lenstra, Hendrik Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.

[LMPR08] Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. Swifft: A modest proposal for fft hashing. In *Fast Software Encryption (FSE) 2008*, Lecture Notes in Computer Science, pages 54–72. Springer-Verlag, 2008.

[LO85] J. C. Lagarias and Andrew M. Odlyzko. Solving low-density subset sum problems. *J. ACM*, 32(1):229–246, 1985.

[Man07] S.A. Manavski. Cuda Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography. 2007.

[MPS07] A. Moss, D. Page, and N.P. Smart. Toward Acceleration of RSA Using 3D Graphics Hardware. *Lecture Notes in Computer Science*, 4887:364, 2007.

[NS05] Phong Q. Nguyen and Damien Stehlé. Floating-point LLL revisited. In *Advances in Cryptology — Eurocrypt 2005*, pages 215–233, 2005.

[NS06] Phong Q. Nguyen and Damien Stehlé. LLL on the average. In *Algorithmic Number Theory Symposium — ANTS 2006*, pages 238–256, 2006.

[Nvi07a] Nvidia. Compute Unified Device Architecture Programming Guide. Technical report, 2007.

[NVI07b] NVIDIA. CUBLAS Library, 2007.

[otCC09] 1363 Working Group of the C/MM Committee. IEEE P1363.1 Standard Specification for Public-Key Cryptographic Techniques Based on Hard Problems over Lattices, 2009. Available at `http://grouper.ieee.org/groups/1363/`.

[Pei09] Chris Peikert. Public-key cryptosystems from the worst-case shortest vector problem: extended abstract. In *Proceedings of the Annual Symposium on the Theory of Computing (STOC) 2009*, pages 333–342, 2009.

[PS08] Xavier Pujol and Damien Stehlé. Rigorous and efficient short lattice vectors enumeration. In *Advances in Cryptology — Asiacrypt 2008*, pages 390–405, 2008.

[RV92] J. L. Roch and G. Villard. Parallel gcd and lattice basis reduction. In L. Bougé, M. Cosnard, Y. Robert, and D. Trystram, editors, *Proceedings of the Second Joint International Conference on Vector and Parallel Processing. Parallel Processing: VAPP V, CONPAR'92 (Lyon, France, September 1992)*, volume 634 of *Lecture Notes in Computer Science*, pages 557–564. Springer-Verlag, 1992.

[Sch91] Claus-Peter Schnorr. Factoring integers and computing discrete logarithms via diophantine approximations. In *Advances in Cryptology — Eurocrypt 1991*, pages 281–293, 1991.

[Sch03] Claus-Peter Schnorr. Lattice reduction by random sampling and birthday methods. In *STACS 2003: 20th Annual Symposium on Theoretical Aspects of Computer Science*, volume 2607 of *Lecture Notes in Computer Science*, pages 146–156. Springer-Verlag, 2003.

[SE91] Claus-Peter Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. In *FCT '91: Proceedings of the 8th International Symposium on Fundamentals of Computation Theory*, pages 68–85. Springer-Verlag, 1991.

[SG08] R. Szerwinski and T. Guneysu. Exploiting the Power of GPUs for Asymmetric Cryptography. *Lecture Notes in Computer Science*, 5154:79–99, 2008.

[Sho] Victor Shoup. Number theory library (NTL) for C++. http://www.shoup.net/ntl/.

[SSTX09] Damien Stehl, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa. Efficient public key encryption based on ideal lattices. Cryptology ePrint Archive, Report 2009/285, 2009. http://eprint.iacr.org/.

[Ste] Damien Stehlé. Damien Stehlé's homepage at école normale supérieure de Lyon. http://perso.ens-lyon.fr/damien.stehle/english.html.

[Vil92] Gilles Villard. Parallel lattice basis reduction. In *ISSAC*, pages 269–277, 1992.

[Wet98] Susanne Wetzel. An efficient parallel block-reduction algorithm. In J. P. Buhler, editor, *Proceedings of the 3rd International Symposium on Algorithmic Number Theory, ANTS'98 (Portland, Oregon, June 21-25, 1998)*, volume 1423 of *Lecture Notes in Computer Science*, pages 323–337. Springer-Verlag, 1998.

The Billion-Mulmod-Per-Second PC

Daniel J. Bernstein[1], Hsueh-Chung Chen[2], Ming-Shing Chen[3], Chen-Mou Cheng[2], Chun-Hung Hsiao[3], Tanja Lange[4], Zong-Cing Lin[2], Bo-Yin Yang[3]

[1] University of Illinois at Chicago
[2] National Taiwan University
[3] Academia Sinica
[4] Technische Universiteit Eindhoven

Abstract. This paper sets new speed records for ECM, the elliptic-curve method of factorization, on several different hardware platforms: GPUs (specifically the NVIDIA GTX), x86 CPUs with SSE2 (specifically the Intel Core 2 and the AMD Phenom), and the Cell (specifically the PlayStation 3 and the PowerXCell 8i). In particular, this paper explains how to carry out more than one billion 192-bit modular multiplications per second on a $2000 personal computer.

1 Introduction

The paper "ECM on Graphics Cards" at Eurocrypt 2009 [6] reported a new implementation of ECM performing 41.88 million 280-bit mulmods per second on an NVIDIA GTX 295 GPU. Here "mulmods" are modular multiplications, and ECM is the elliptic-curve method of factorization, a critical subroutine inside NFS, the number-field sieve. See [6, Section 1] for discussion of the cryptanalytic importance of ECM and NFS.

For comparison, the standard GMP-ECM software package, running simultaneously on all four cores of an Intel Core 2 Quad Q9550 CPU, performs only 14.85 million mulmods per second with the same 280-bit modulus length. The same paper recommended building a $2226 computer with two GTX 295 GPUs and a Core 2 Quad Q6600 CPU, performing in total an astonishing 96.79 million 280-bit mulmods per second.

In this paper we show that GPUs are capable of much higher performance. For example, with 210-bit moduli, the same GTX 295 can carry out 481 million mulmods per second. This example uses a somewhat smaller modulus size than [6], but this change explains only a small part of the tenfold increase in speed.

This paper also sets new ECM speed records on several different CPUs: for example, with 192-bit moduli, a Cell-based IBM BladeCenter QS22 can carry out 334 million mulmods per second; an AMD Phenom II 940 can carry out 202 million mulmods per second (20% more on the same CPU than the ECM software being used in the ongoing RSA-768 factorization project); an Intel Core 2 Quad Q9550 can carry out 114 million mulmods per second; and a low-cost PlayStation 3 can carry out 102 million mulmods per second with slightly larger, 195-bit moduli. Our software is tuned in many platform-specific ways but in every case benefits from systematically exploiting the available parallelism.

We find that GPUs are faster than CPUs, but that the best price-performance ratio is achieved by computers that run CPUs and GPUs simultaneously, as in [6]. Specifically, a computer with one Phenom II 940 CPU and two GTX 295 GPUs costs only about $2000 and handles 1.1 billion 192-bit mulmods per second with our ECM software, several times faster than the best result of [6].

Our GPU software goes beyond the software of [6] not only in speed but also in generality: most importantly, within the range of modulus sizes that are of interest inside NFS-over-ECM, we handle several different sizes, while [6] handled only 280 bits. We expect the same techniques to be useful for other computations bottlenecked by modular multiplication. The traditional example (typically with 1024-bit moduli, larger than in this paper) is RSA, while a much more modern example (typically with similar modulus sizes to this paper) is evaluation of pairings to verify short signatures. Note that for efficiency one must feed many simultaneous computations to the hardware; this is not possible for a laptop carrying out an occasional cryptographic computation, but it is feasible for a busy server bottlenecked by cryptography, and it is very easily achieved inside cryptanalytic computations such as NFS-over-ECM.

Readers not familiar with ECM can find all relevant background in [32], [7], and [6].

2 Today's Computing Hardwares

2.1 X86 and Streaming SIMD Extensions

The 64-bit x86 instruction set (x86-64) is supported by all AMD CPUs since the K8 (Opteron and Athlon 64), some versions of the Intel Pentium 4, and all Intel Core 2 and i7 CPUs. In x86-64 there are sixteen 64-bit general-purpose integer registers (GPRs). Modern x86-64 processors decode the variable-length CISC instructions into RISC-like micro-operations for possibly out-of-order dispatching in 3 unified pipelines (Intel Core) or 3 integer plus 3 floating-point pipelines (AMD).

The GNU Multi-Precision (GMP) library uses the biggest multiplication available, the `MUL` instruction (unsigned $64\times64 = 128$-bit) to compute multi-precision integers in a straightforward manner, using 64-bit limbs with native `ADC` (add-with-carry) instructions.

AMD K8 and K10 CPUs sport an impressive integer multiplier that can in theory dispatch a $64 \times 64 = 128$-bit `MUL` once every two cycles with a latency of 4–5 cycles. In practice other bottlenecks — principally the forced use of registers `(RDX,RAX)` for the product and `RAX` for the multiplicand — makes it challenging to average one $64 \times 64 = 128$-bit `MUL` every 3 cycles.

Intel CPUs can only dispatch one $64 \times 64 = 128$-bit `MUL` every 4 cycles, with a latency of 7 cycles. Furthermore, `MUL` uses resources (32-bit multipliers) that conflict with other instructions. Therefore it becomes imperative to consider other approaches to big integer arithmetic than the 64-bit `MUL` instructions, such as the x86 vector instructions below.

SSE2 Instructions All Intel CPUs since the Pentium 4 and all AMD CPUs since the K8 supports the SSE2 (Streaming SIMD Extensions 2) instruction set, where SIMD in turn stands for Single Instruction Multiple Data (performing the same action on many operands). SSE2 instruction set operates on 16 architectural 128-bit registers, called `xmm` [0–15], as packed 8-, 16-, 32- or 64-bit `int`s. The instructions are arcane and highly non-orthogonal:

Load/Store: Between `xmm` and memory or lowest `xmm` unit zero-extended and GPR.
Reorganize Data: Multi-way 16- and 32-bit move called Shuffles (8-bit available in SSSE3 only), and Packing, Unpacking, or Conversion on vector data of different densities.
Logical: AND, OR, NOT, XOR; Shift (packed 16-, 32- or 64-bits) Left, Right Logical and Right Arithmetic; Shift entire `xmm` register right/left byte-wise only.

Arithmetic: Add/subtract on 8-, 16-, 32- and 64-bit integers (including "saturating" versions); multiply of 16-bits (high and low word returns, signed and unsigned, and *fused multiply-multiply-adds*) and 32-bits unsigned; max/min (signed 16-bit, unsigned 8-bit); unsigned averages (8-/16-bit); sum-of-differences for 8-bits. A regular set of arithmetic instructions are available on IEEE-754 single and double floats.

Experiments demonstrate that, on AMD CPUs, integer multiplication uses separate computational resources from the vector instructions. On Intel CPUs the resources conflict.

Both Intel Core and AMD K10 architectures can pipeline most vector instructions with a theoretical throughput of one instruction per cycle. One attractive possibility is vectorized floating-point arithmetic, specifically the `MULPD` (multiply 2 packed doubles — 53 bits of mantissa) instruction, with a radix of 2^{24}. Another attractive possibility is vectorized integer arithmetic, specifically the `PMULUDQ` (packed multiply unsigned doubleword to quadword) instruction, which can do two $32 \times 32 = 64$-bit products every cycle. Of course without intrinsic carry flags for SSE additions, for big integer arithmetic we still need to hand-carry *unsigned* limbs. Still, we are able to go as high as radix 2^{30}, which usually saves a limb, and carrying integral limbs uses shifts and bitmasks, which is no worse than the add-subtract in float limbs.

For completeness, we tested and optimized single-thread assembly code using `MUL`, `PMULUDQ`, and `MULPD` as the principal way to multiply for every x86-64 CPU we have. A simple model predicts, and experiments confirm, that `PMULUDQ` is fastest for Intel and using the 64-bit `MUL` is best for AMD.

Other vector instruction sets on x86 (SSE3, SSSE3, and SSE4) have no further instructions that help big integer arithmetic throughput. Note that the signed $32 \times 32 = 64$ multiply (`PMULDQ`) in SSSE3 cannot be used due to the lack of an arithmetic 64-bit right shift. We expect this to change only when AMD's SSE5 (with fused multiply-adds) come to market.

2.2 Graphics Cards from NVIDIA and CUDA

Today's graphics cards contain powerful graphics processing units (GPUs) to handle the increasing complexity and screen resolution in video games. GPUs have now developed into a powerful, highly parallel computing platform that finds more and more interest outside graphics-processing applications. In cryptography, there have been many attempts of exploiting the computational power of GPUs [10,11,26,31]. In particular, Bernstein *et al.* have explored the possibility of using graphics cards to speed up ECM computation [6]. The interested reader is referred to their paper for a more detailed description of the GPU computing platform and various NVIDIA graphics cards; here we only provide a brief summary of GPU programming. More importantly, we will compare our results with theirs in Section 4. Some of the information here is repeated from [6] to keep this paper self-contained.

Like Bernstein *et al.*, we use NVIDIA's GPUs because they provide a programmer-friendly parallel programming environment called CUDA. A GPU program (`*.cu`) is written in CUDA and compiled into intermediate virtual machine code (`*.ptx`). The NVIDIA driver then converts that into real machine code (`*.cubin`) and loads it to run with appropriate data.

A typical NVIDIA GPU contains several to several tens of streaming multiprocessors (MPs), as depicted in Figure 1. Each MP contains a scheduling and dispatching unit that

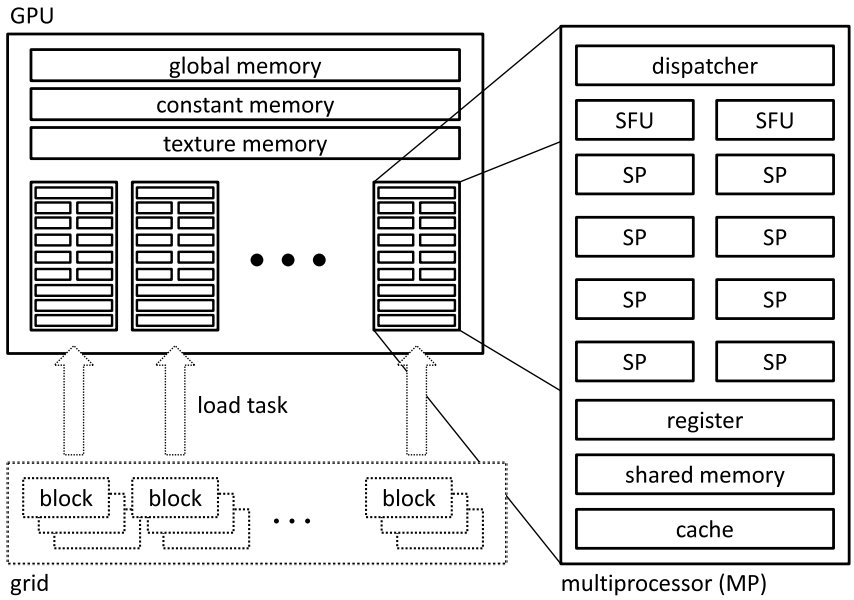


Fig. 1. NVIDIA's GPU Block Diagram

can handle many lightweight threads. The actual computation is done by eight streaming processors (SPs) and two super function units (SFUs) on each MP, and a GPU typically contains tens to hundreds of these SPs, which NVIDIA advertises as "cores."

Like in any other architecture with a memory hierarchy, the closer any memory is to the processor core, the faster it will be. So there are, as shown in Figure 1, a big register file, fast on-die shared memory, fast local read-only caches to device memory on the card, and uncached thread-local and global memories. Note that the NVIDIA platform only provides read-only caching. Programmers are on their own to manage the caching of read-write memories.

Uncached memories have relatively low throughput and long latency. For example, a GeForce 8800 GTX has 128 SPs running at 1.35 GHz; its controllers have a total throughput of only 86.4 GB/s to device memory. This translates to *one* 32-bit floating-point number per cycle per MP, not to mention the latency of 400–600 cycles.

These characteristics mean that GPU programming generally involves controlling a large number of *threads*. The benefits of using many threads are twofold. First, we will be able to exploit *thread-level parallelism* in the application via mapping these threads to the hundreds of SPs in GPU and having them run in parallel. Secondly, having multiple threads time-share a physical SP enables latency hiding, a well-known strategy in the computer architecture community. Only with enough threads can we eliminate wasted clock cycles (caused by dependent pipeline stalls) and fully utilize all computational units available on GPU. In particular, NVIDIA reports the theoretical maximal GFLOPS of their GPUs as if the user can always run enough threads filling up the 20+-stage pipelines of all SPs.

In the CUDA programming model, the threads of an application are organized in a two-tier hierarchy. At top level they form a *thread grid*, which just means that they run one single program called the *kernel*. A grid of threads are divided into *thread blocks*. Threads in

the same block can cooperate, while different blocks of threads run independently. A block of threads must be executed by one single MP, which makes intra-block cooperation such as thread synchronization much easier. Thread blocks also make scaling easier: GPUs with more MPs can process more blocks in parallel without changing the program or the kernel configuration.

Even though the CUDA programming model is about the concept of threads, it is essential for the programmer to understand that the minimal scheduling entity is actually a *warp*. In CUDA, the number of threads in a warp depends on microarchitecture implementation; for all current GPUs it is 32. Each instruction is in fact decoded only once a warp. Hence, *each thread in a warp must run the same program (SPMD or same program multiple data)*, controlled by built-in variables (e.g., `ThreadID`). Thus we need at least $8 \times 24/32 = 6$ warps per MP since a float instruction has a latency of 20–24 cycles.

The GPU threads are lightweight hardware threads, which incur very fast context switches with little overhead. To support this, the physical registers are divided among all active threads. For example, on 8800 GTX there are 8192 registers per MP. If we were to use 256 threads, then each thread could use 32 registers — very few for complicated algorithms. *The register pressure can be even greater, as sometimes the conversion of the virtual machine code leaves something to be desired.* GPUs from the GT2xx family have twice as many registers, making things much easier.

To summarize, the massive parallelism in NVIDIA's GPU architecture makes GPU programming very different from sequential programming on traditional CPUs. In general, GPUs are most suitable for executing the data-parallel part of an algorithm. To get the most out of the theoretical arithmetic throughput, one must maximize the ratio of arithmetic operations to memory accesses.

2.3 Cell Processor

The Cell processor was jointly developed by Sony, Toshiba, and IBM in 2005. A Cell processor is constructed from one Power Processing Element (PPE) and eight Synergistic Processor Elements (SPEs) with a high-bandwidth Element Interconnection Bus (EIB), as shown in Figure 2. The processor cores work at clock rates up to 4 GHz, while the EIB works at half of

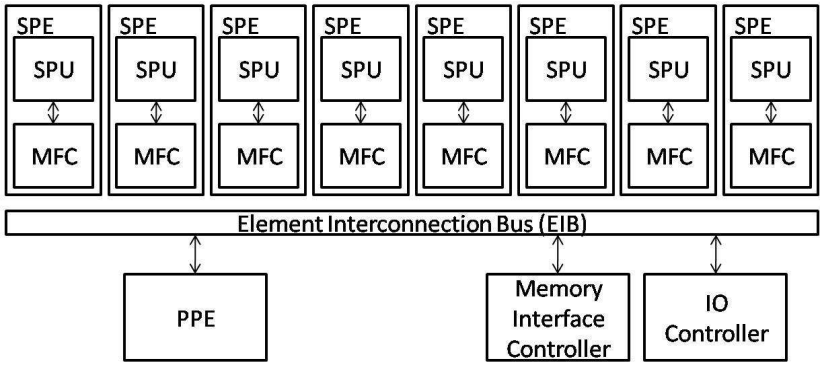


Fig. 2. The Cell Processor Block Diagram

the system clock rate. EIB does not only connects PPE and SPE but also memory and I/O controllers. EIB is composed of four rings, each offering a bandwidth of 16 bytes per cycle, and supports multiple simultaneous transfers per ring. The peak bandwidth of the entire EIB is 96 bytes per cycle. The PPE is more of a conventional PowerPC processor, which supports two-level on-chip caches with multi-threading capability and vector multimedia extensions. The PPE's main task is usually to run the operating system.

Each SPE is composed of a synergistic processor unit (SPU) and a memory flow controller (MFC). The SPU, acting like a RISC processor, is used mainly for computation. Each SPU has a SIMD unit that is capable of vector processing of integer and floating-point numbers of various lengths. The SIMD unit is an important feature for high-performance computing and will be described in more detail later in this section.

The SPU contains a 256-kilobyte local store for instructions and data needed for executing a program on it. Instructions and data must be explicitly moved to the local store by sending direct memory access (DMA) commands to the MFC. The MFC acts like a DMA engine and handles communication between the local SPE core and other cores, main memory, and the I/O controller. The use of DMA allows for efficient use of memory bandwidth and enables overlapping of computation and communication.

Figure 3 shows the block diagram of an SPE. Each SPU has a large 128-entry, 128-bit-

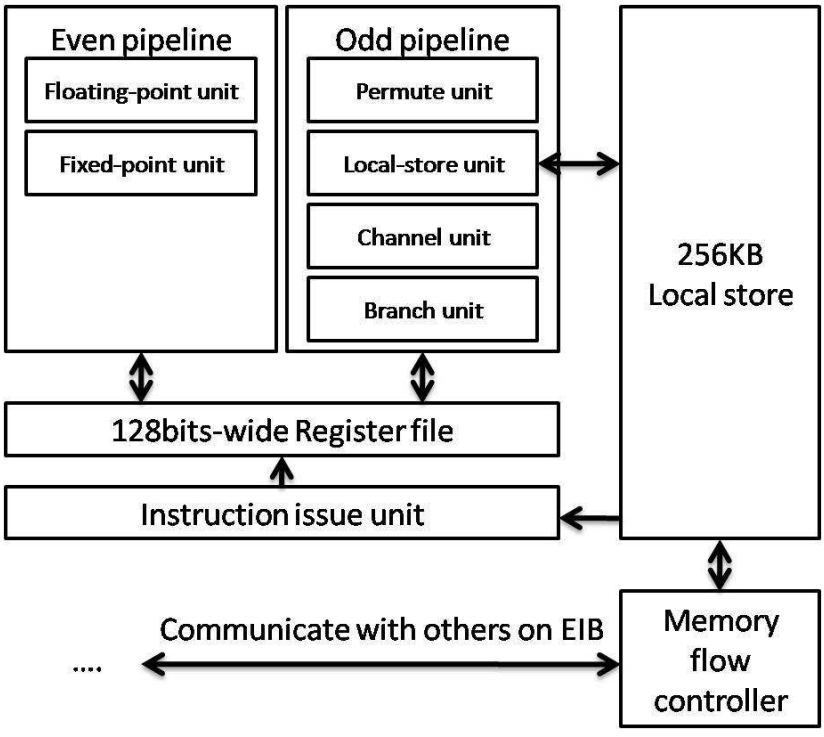


Fig. 3. SPE Block Diagram

wide register file. The flat architecture (all operand types stored in a single register file) of the register file allows for instruction latency hiding without speculation [1]. The SPU

has two in-order issued pipelines, called the even and odd pipelines, which can issue and complete up to two instructions per cycle. The two pipelines handle different instruction types, as shown in Figure 3. Roughly, value computation will use the even pipeline while access to local store (including address calculation) will use the odd pipeline. The arithmetic logical units (ALUs) in the SPU are also designed to support 128-bit-wide SIMD operations, which can process up to eight short integer operations, four single-precision floating-point operations, or two double-precision floating-point operations concurrently every cycle.

In 2008, IBM introduced a new variant of the Cell processors, called the PowerXCell 8i. Compared with the previous Cell processors, PowerXCell 8i supports fully-pipelined double-precision floating-point operations. The double-precision peak throughput of a PowerXCell 8i processor is 102 GFLOPS using 8 SPEs, as opposed to 14 GFLOPS with the previous Cell processor. The Roadrunner, currently #1 on the list of Top 500 supercomputers [2], consists of 12960 PowerXCell 8i processors and offers a peak performance of more than 1700000 GFLOPS.

There has been a small amount of previous work in optimizing cryptographic algorithms on the Cell processor. Costigan and Scott published their experience porting SSL to the Cell processor [12]. They use multi-precision math library provided by IBM Cell's SDK on the SPE to implement the kernel operations in SSL, whereas we implement our own multi-precision arithmetic without using IBM's library. Recently, Costigan and Schwabe reported a fast implementation of elliptic curve cryptography based on Curve25519 for the Cell [13]. This curve is defined modulo $2^{255} - 19$ to allow extremely fast reduction. We handle general moduli as required for ECM.

3 Implementation

3.1 Elliptic-curve Arithmetic

We use the windowed double-and-add algorithm to compute the scalar multiples of a point on elliptic curves [6, 7], in which a scalar multiplication is transformed into a sequence of point doublings and additions. To avoid the expensive division operations, we use the projective coordinates to represent the point $Q = (X : Y : Z)$, whereas the starting point P is stored in its affine coordinates (x, y). We choose the standard Edwards coordinates and use the mixed addition law on Edwards curves [21]. The addition law is given by

$$\begin{aligned} X_3 &= Z_1(X_1Y_2 - Y_1X_2)(X_1Y_1 + Z_1^2X_2Y_2) \\ Y_3 &= Z_1(X_1X_2 + Y_1Y_2)(X_1Y_1 - Z_1^2X_2Y_2) \\ Z_3 &= Z_1^2(X_1X_2 + Y_1Y_2)(X_1Y_2 - Y_1X_2), \end{aligned}$$

whereas the doubling law is given by

$$\begin{aligned} X_3 &= ((X_1 + Y_1)^2 - (X_1^2 + Y_1^2))((X_1^2 + Y_1^2) - 2Z_1^2) \\ Y_3 &= (X_1^2 + Y_1^2)(X_1^2 - Y_1^2) \\ Z_3 &= (X_1^2 + Y_1^2)((X_1^2 + Y_1^2) - 2Z_1^2). \end{aligned}$$

Note that the extended Edwards coordinates presented by Hisil et al. [20] save 1 multiplication per addition but require extra storage and scheduling. On several platforms storage is a concern so we did not apply those formulas.

In the windowed double-and-add algorithm, the number of doublings will be equal to the number of bits in the scalar k, while the number of additions will depend on the window size. With a larger window size, a smaller number of additions will be executed during the computation of the scalar multiplication. However, this speed-up comes at a price of extra storage space, i.e., the pre-computed points need to be stored in memory. On modern x86 CPUs, this is not a problem since the on-die caches are usually large enough to hold many pre-computed points. On the Cell processor and GPU, the on-die fast memories are more limited, and we need to store the pre-computed points in off-chip device memories, accessing which involves a high latency. The Cell processor has an architectural design that helps relieve this problem. Namely, with the help of MFC, we are able to store pre-computed points in off-chip main memory rather than in on-die local store. Since the computation time of a point doubling on an elliptic curve is much longer than the transmission time for fetching the next pre-computed point to be used in the subsequent addition (if any), we can overlap computation and communication via the well-known double-buffer mechanism. As a result, our ECM implementation is able to support virtually an arbitrarily large window size. A similar latency-hiding strategy also works on NVIDIA GPU, except that we need to explicitly move the data, as opposed to the use of a DMA memory controller.

3.2 Modular Arithmetic

The kernel operation of ECM is multi-precision integer modular arithmetic, in which an integer is represented using L limbs in radix 2^r with each limb ranging from -2^{r-1} to 2^{r-1} and stored in a single-precision variable. The single-precision arithmetic can be carried out by either fixed-point or floating-point arithmetic, depending on which arithmetic delivers higher throughput on the chosen hardware platform. Such a representation allows us to represent any integer between $-R/2$ and $R/2$, where $R = 2^{Lr}$.

On the x86 CPU, we take advantage of the wide arithmetic pipelines and use 64-bit integer arithmetic aided by XMM arithmetic. On NVIDIA GPU, we use 24-bit integer arithmetic, which in a single cycle can produce the lower 32 bits of the product of two 24-bit integers. We also use full 32-bit addition and subtraction, whose throughput is one every cycle. On the Cell processor, we use 16-bit integer arithmetic, which in a single cycle can produce a 32-bit product. The PowerXCell 8i processor has a better, fully-pipelined double-precision floating-point arithmetic implementation, which we take advantage of and implement the modular arithmetic with.

Stage-1 ECM requires additions, subtractions, and multiplications modulo N, where N is the number to be factored. We use Montgomery's method to avoid trial divisions in computing modular operations [25]. In this method, addition and subtraction modulo N are straightforward, as we can simply add and subtract, respectively, the two operands limb by limb, followed by a carry reduction to bring each limb to its normal range of -2^{r-1} to 2^{r-1}. We note that it is fairly safe to skip a small number of carry reduction steps after an addition or a subtraction because we have some headroom in the storage of the limbs if we do not need to multiply the result immediately.

The modular multiplication is more complicated, as it involves a multiplication step followed by a reduction step. Textbook description says that there are more advanced algorithms, e.g., the Karatsuba multiplication, that would outperform the plain schoolbook multiplication when the number of limbs is in the range that we are interested in. However, as the latter makes better use of the native fused multiply-and-add (MADD) instruction on the Cell processor and GPU, it turns out to be faster in this context and hence becomes the choice of multiplication method in our implementation. On the x86 CPU, since the number of limbs is small, we go to schoolbook directly.

The following step is the modular reduction. Suppose that the two original operands have L limbs with radix 2^r in multiplication step. Multiplication produces a partial product C with $2L$ limbs such that $C = \sum_{i=0}^{2L-1} c_i 2^{ri}$. In the multi-precision case of Montgomery multiplication, we will eliminate the lower half of C by adding a specific multiple of the modulus N sequentially, i.e., making $c'_0 = 0$, $c'_1 = 0, \ldots, c'_{L-1} = 0$ so that after this elimination step, the upper half $c'_{2L-1}2^{r(L-1)} + c'_{2L-2}2^{r(L-2)} + \ldots + c'_L$ will be the result of modular multiplication.

As we have mentioned in Section 3.1, some of the operands of the modular arithmetic operations are stored in off-chip device memories on the Cell processor and GPU. To load these operands incurs a long latency, which we hide via the well-known double-buffer strategy. To support this strategy, each modular arithmetic operation is further broken down into three sub-operations: load, execute, and store. This is similar to the design philosophy of the Reduced Instruction Set Computer (RISC), in which memory latency is exposed to the compiler designers and assembly-language programmers so that they can schedule the instructions properly to hide memory latency via instruction-level parallelism.

One reason why we are able to achieve such a tremendous speed-up over previous results is that we have a different thread organization. Recall that in [6], one modular multiplication is carried out by a group of 28 threads. This same amount of work can be done by fewer threads; in fact, there is a natural way to divide the work equally to be done by k threads as long as k divides the number of limbs n. It is easy to verify that in such a work decomposition, the total number of registers required for a fixed amount of computation roughly remains constant. If one uses less threads to compute a single multiplication, then each thread will use more resources, putting more pressure on, e.g., the fast on-die shared memory. The other hand, each thread will do more work, and hence we will have a higher compute-to-memory-access ratio. In [6], the authors used a design that is on one extreme of the spectrum, namely, n threads to compute an n-limb multiplication. In this paper, we try the other end, namely, one single thread to compute one n-limb multiplication. We are able to achieve a much improved compute-to-memory-access ratio, as well as eliminate inefficiencies due to synchronization overhead and such, resulting in a much improved performance.

3.3 Software Pipelining, Loop Unrolling, and Hyperthreading

ECM is embarrassingly parallelizable and can exploit SIMD by running many curves in parallel. This is always achievable in practice, since trying ECM on a single curve with a large parameter B_1 is not as effective as using the same amount of time to try many curves with a smaller B_1. This is also necessary for GPUs, since fewer threads would not run faster — we would see almost the same speed with many pipeline stalls. The reason of course is that compared to modern processors, the SPs in GPUs do very high latency operations. However, this phenomenon is not limited to GPUs.

We know that modern CPUs often have out-of-order execution and their dispatchers look very hard for ILP (instruction-level parallelism). But sometimes there is just not enough ILP, and all the pipelines would be mostly full of bubbles. In our preliminary implementations we observe some of these situations, especially in the reduction step of Montgomery modular multiplication. For example, on an SPU of the PowerXCell 8i processor, it takes about 900 cycles to complete two Montgomery multiplications simultaneously, using two-way SIMD on double-precision floats, of which 500 are actually wasted due to data-dependent stalls.

An analogous situation happened with an extreme case among modern CPUs, namely the Pentium 4, which has ALU running at a clock twice as fast as the rest of the chip, but with pipelines with 30+ stages. Even with out-of-order execution, it is extraordinarily difficult to fill a pipeline that is even deeper than the GPUs today. The difficulty to locate and use ILP is compounded because there are only six general-purpose registers.

Part of Intel's solution is to make the CPU run two hardware threads. The two threads each have their own set of architectural registers, switching whenever there is a stall. Intel calls this form of symmetric multithreading *hyperthreading*. While it of course can never get close to the 2× speedup from having another core, hyperthreading can achieve fairly impressive gains for certain classes of code.

If there are enough spare registers, both architectural and actual, then we can apply the following strategy to utilize these unused resources as well as the computational power wasted by the pipeline stalls. *We can run more "threads" of computation simultaneously* by interleaving instructions from several flows of independent computations into one single physical thread of instructions. By measuring the percentage of time spent in useful computation, we conclude that such a strategy of combining software pipelining and loop unrolling (SPLU) does work well on Cells.

On x86-64 CPUs, SPLU cannot work as above since they have too few GPRs architecturally. However, a different kind of SPLU is possible in the following sense: Modern x86-64 CPUs have multiple independent pipelines that execute multiple instructions in parallel. Their combined throughput is additive if there is no contention to shared resources such as arithmetic circuitry or external memory. When mixed properly, a sequence of instructions containing a stream of 64-bit integer multiplications (using `MUL` and GPRs) and another stream of 32-bit SIMD integer multiplications (using `PMULUDQ` and XMM registers) can theoretically achieve a throughput close to those combined from two threads executed separately on the latest AMD Phenom IIs. This we call *heterogenous software multi-threading*.

In practice, we are able to speed up our AMD code by 20%+. This agrees with conventional wisdom that the two types of multiplications share no circuitry on an AMD CPU.

Unfortunately this is not the case with Intel CPUs, and the throughput of the combined instruction stream is much lower than the sum of the throughputs had we executed two individual streams. *However, our heterogeneous software hyperthreaded ECM code used for the C2+ still gained more than native hyperthreading when we ran it on the Ci7.*

4 Experimental Results

We measure the performance of our implementations of stage-1 ECM for $B_1 = 8192$ on various hardware platforms and present the experimental results in this section. We have three different families of hardware platforms: GPU, x86 CPU, and Cell. For GPU, we

perform our experiments on NVIDIA GTX 295. For x86 CPU, we have AMD Phenom II 940 at 3 GHz (K10+) and Intel Core 2 Quad Q9550 at 2.83 GHz (C2+). For Cell, we have Sony PlayStation 3 (PS3) and IBM BladeCenter QS22, and only the latter supports high-throughput double-precision floating-point arithmetic.

The performance of our latest implementations of stage-1 ECM for these hardware platforms is summarized in Figure 4 and Table 1. We note that in Table 1, since different

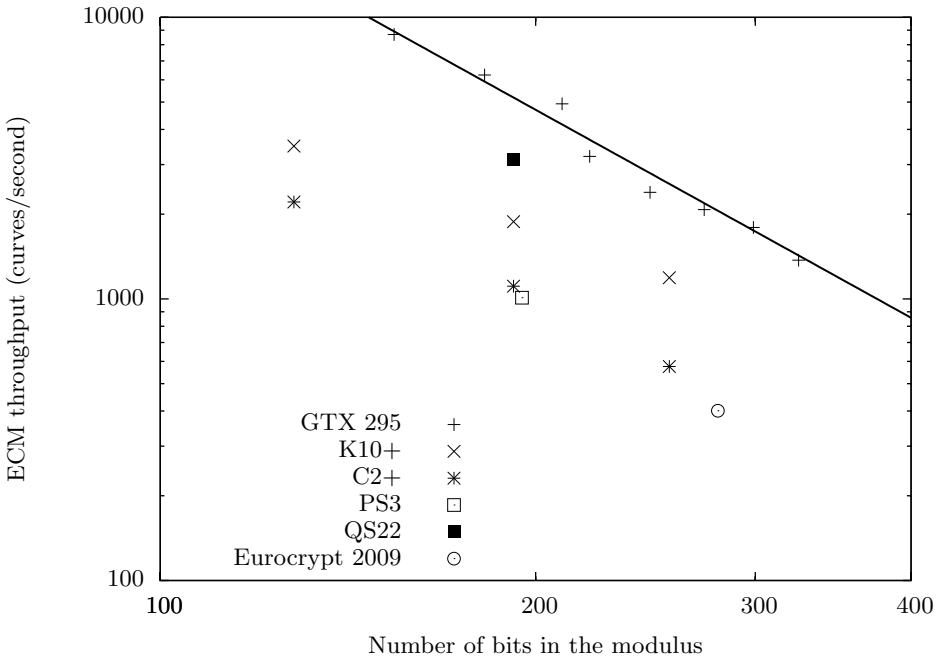


Fig. 4. Performance comparison of stage-1 ECM on various hardware platforms

implementations may use different bit lengths, we scale the results to 192 bits in order to compare their performance. As a rule of thumb, since the bottleneck of the computation is the (schoolbook) multi-precision integer multiplication, whose complexity grows quadratically as the number of bits in the multiplicand, we use the square of the length of the moduli in bits to scale the results. For example, the result of a 280-bit ECM would be scaled by $280^2/192^2$, or roughly a factor of two. Such a scaling ignores factors such as pressure on on-die memories, which can be significant for GPU implementations. As we can see from Figure 4, there are two dips on the GPU curve when we go above 210 bits and 299 bits. These are precisely when we have to reduce the number of thread blocks because we do not have enough fast memories to support as many thread blocks. As a result, the exponent of the linear regression line for GPU result on logarithmic scale is -2.46, showing that the performance actually drops faster than quadratically as we increase the number of bits in the modulus.

It is clear that from Figure 4, the GPUs have the best performance across all modulus lengths. The runner-up is AMD's K10+, whose price-performance ratio is also very compet-

Table 1. Performance results of stage-1 ECM on selected hardware platforms.

	GTX 295	K10+	C2+	QS22	PS3	GTX 295 [6]
#cores	480	4	4	16	6	480
clock (MHz)	1242	3000	2830	3200	3200	1242
price (USD)	500	190	270	$$$	413	500
TDP (watts)	295	125	95	200	<100	295
GFLOPS	1192	6+24	3+23	204	154	1192
#threads	46080	48+16	48+16	160	6	
#bits in moduli	210	192	192	192	195	280
#limbs	15	3+7	3+7	8	15	28
window size (bits)	u6	u6	u6	s5	s5	s4
mulmods (10^6/sec)	481	202	114	334	102	42
curves (1/sec)	4928	1881	1110	3120	1010	401
curves (1/sec, scaled)	5895	1881	1110	3120	1042	853

itive. Since the CPU results are obtained via SPLU, we list two numbers for GFLOPS and #threads, one for 64-bit integer (left) and the other for SIMD units (right). We note that such GFLOPS rating can be misleading since different platforms have different arithmetic pipeline widths, and the wider the pipeline is, the more useful a "FLOP" is, which is evident from the numbers of GPU vs. CPU as well as QS22 vs. PS3.

It is important to note that this gain in computational power does not come at a price of power consumption. The billion-mulmod PC that we recommend can be run on a 850W power supply, whereas we measured from the outlet a K10+ (Phenom II 940) running our code and got 170W. Since our box is more than five times as fast as the K10+, the billion-mulmod-per-second PC is more efficient per watt.

We show the effect of heterogeneous software hyperthreading on x86 CPUs in Figure 5, using 192-bit mulmods on AMD K10+ as an example. In Figure 5, each point represents a way to mix the XMM and integer instructions. We see that some ways of mixing actually result in worse performance than time sharing between XMM and integer units, although most combinations yield some improvements.

The Cell processor is also quite competitive in terms of price-performance ratio, as its price in Table 1 is that of a whole machine, unlike the other platforms for which only the component prices are listed. This is largely due to the fact that Sony is currently subsidizing its PS3 sales, making PS3 an attractive platform for ECM.

References

1. — (no editor), *SDK 3.1 Programming Tutorial*, IBM, 2008. Cited in §2.3.
2. TOP500 Supercomputing Sites, *32nd Top500 list* (2008). URL: `http://www.top500.org/lists/2008/11/`. Cited in §2.3.
3. — (no editor), *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Intel Corp., 2007. URL: `http://www.intel.com/design/processor/manuals/248966.pdf`.
4. — (no editor), *13th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2005), 17–20 April 2005, Napa, CA, USA*, IEEE Computer Society, 2005. ISBN 0-7695-2445-1. See [30].
5. A. O. L. Atkin, Francois Morain, *Finding suitable curves for the elliptic curve method of factorization*, Mathematics of Computation **60** (1993), 399–405. ISSN 0025-5718. MR 93k:11115. URL: `http://www.lix.polytechnique.fr/~morain/Articles/articles.english.html`.

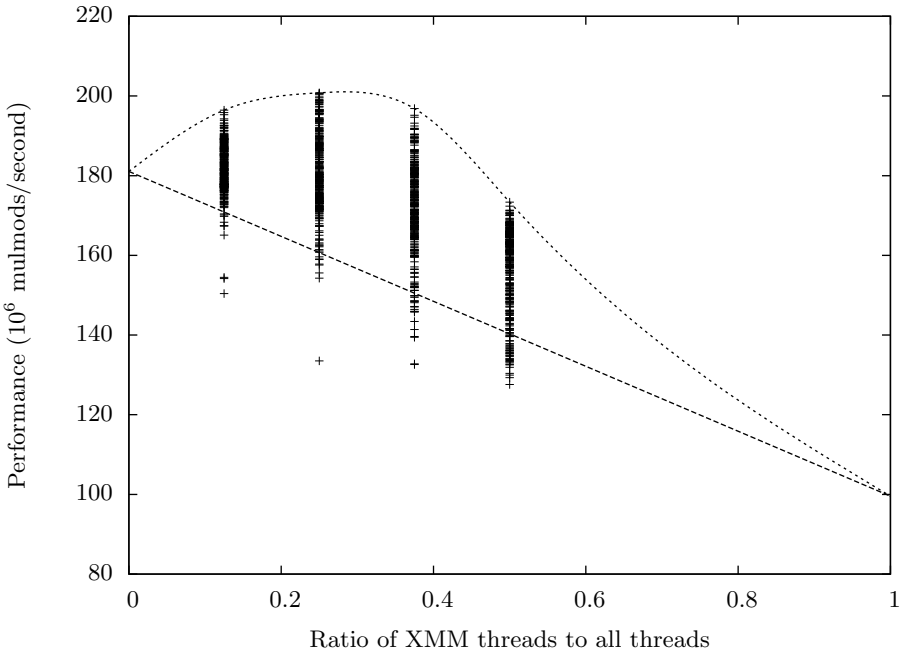


Fig. 5. Effect of Heterogeneous SSMT for 192-bit mulmods on K10+

6. Daniel J. Bernstein, Tien-Ren Chen, Chen-Mou Cheng, Tanja Lange, Bo-Yin Yang, *ECM on Graphics Cards*, in Eurocrypt [22] (2009), 483–501. URL: `http://eprint.iacr.org/2008/480/`. Cited in §1, §1, §1, §1, §1, §1, §1, §1, §2.2, §2.2, §3.2, §3.2, §1.
7. Daniel J. Bernstein, Peter Birkner, Tanja Lange, Christiane Peters, *ECM using Edwards curves* (2008). URL: `http://eprint.iacr.org/2008/016`. Cited in §1.
8. Daniel J. Bernstein, Tanja Lange, *Explicit-formulas database* (2008). URL: `http://hyperelliptic.org/EFD`.
9. Daniel J. Bernstein, Tanja Lange, *Faster addition and doubling on elliptic curves*, in Asiacrypt 2007 [23] (2007), 29–50. URL: `http://cr.yp.to/papers.html#newelliptic`.
10. Debra L. Cook, John Ioannidis, Angelos D. Keromytis, Jake Luck, *CryptoGraphics: Secret Key Cryptography Using Graphics Cards*, in CT-RSA 2005 [24] (2005), 334–350.
11. Debra L. Cook, Angelos D. Keromytis, *CryptoGraphics: Exploiting Graphics Cards For Security*, Advances in Information Security, 20, Springer, 2006. ISBN 978-0-387-29015-7.
12. Neil Costigan, Michael Scott, *Accelerating SSL using the vector processors in IBM's Cell Broadband Engine for Sony's Playstation 3* (2007). URL: `http://eprint.iacr.org/2007/061/`. Cited in §2.3.
13. Neil Costigan, Peter Schwabe, *Fast elliptic-curve cryptography on the Cell Broadband Engine* (2009). URL: `http://eprint.iacr.org/2009/016/`. Cited in §2.3.
14. Harold M. Edwards, *A normal form for elliptic curves*, Bulletin of the American Mathematical Society **44** (2007), 393–422. URL: `http://www.ams.org/bull/2007-44-03/S0273-0979-07-01153-6/home.html`.
15. Jens Franke, Thorsten Kleinjung, Christof Paar, Jan Pelzl, Christine Priplata, Colin Stahlke, *SHARK: A Realizable Special Hardware Sieving Device for Factoring 1024-Bit Integers*, in CHES 2005 [29] (2005), 119–130.
16. Kris Gaj, Soonhak Kwon, Patrick Baier, Paul Kohlbrenner, Hoang Le, Mohammed Khaleeluddin, Ramakrishna Bachimanchi, *Implementing the Elliptic Curve Method of Factoring in Reconfigurable Hardware*, in CHES 2006 [18] (2006), 119–133.
17. Steven D. Galbraith (editor), *Cryptography and Coding, 11th IMA International Conference, Cirencester, UK, December 18–20, 2007*, Lecture Notes in Computer Science, 4887, Springer, 2007. ISBN 978-3-540-77271-2. See [26].

18. Louis Goubin, Mitsuru Matsui (editors), *Cryptographic Hardware and Embedded Systems — CHES 2006, 8th International Workshop, Yokohama, Japan, October 10–13, 2006*, Lecture Notes in Computer Science, 4249, Springer, 2006. ISBN 3-540-46559-6. See [16].
19. Florian Hess, Sebastian Pauli, Michael E. Pohst (editors), *Algorithmic Number Theory, 7th International Symposium, ANTS-VII, Berlin, Germany, July 23–28, 2006*, Lecture Notes in Computer Science, 4076, Springer, Berlin, 2006. ISBN 3-540-36075-1. See [32].
20. Huseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, Ed Dawson, *Twisted Edwards Curves Revisited*, in Asiacrypt 2008 [28] (2008), 326–242. URL: `http://eprint.iacr.org/2008/552`. Cited in §3.1.
21. Huseyin Hisil, Kenneth Wong, Gary Carter, Ed Dawson, *Faster group operations on elliptic curves* (2007). URL: `http://eprint.iacr.org/2007/441`. Cited in §3.1.
22. Antoine Joux (editor), *Advances in Cryptology - Eurocrypt 2009 (28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009*, Lecture Notes in Computer Science, 5479, Springer, 200. See [6].
23. Kaoru Kurosawa (editor), *Advances in cryptology — ASIACRYPT 2007, 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2–6, 2007*, Lecture Notes in Computer Science, 4833, Springer, 2007. See [9].
24. Alfred J. Menezes (editor), *Topics in Cryptology — CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14–18, 2005*, Lecture Notes in Computer Science, 3376, Springer, 2005. ISBN 3-540-24399-2. See [10].
25. Peter L. Montgomery, *Modular multiplication without trial division*, Mathematics of Computation **44** (1985), 519–521. URL: `http://www.jstor.org/pss/2007970`. Cited in §3.2.
26. Andrew Moss, Dan Page, Nigel P. Smart, *Toward Acceleration of RSA Using 3D Graphics Hardware*, in Cryptography and Coding 2007 [17] (2007), 364–383.
27. Elisabeth Oswald, Pankaj Rohatgi (editors), *Cryptographic Hardware and Embedded Systems — CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10–13, 2008*, Lecture Notes in Computer Science, 5154, Springer, 2008. ISBN 978-3-540-85052-6. See [31].
nference on the Theory and Application of Cryptology and Information Security, Melbourne, Australia, December 7-11, 2008.
28. Josef Pieprzyk (editor), *Advances in Cryptology - ASIACRYPT 2008, 14th International Co*, Lecture Notes in Computer Science, 5350, Springer, 2008. See [20].
29. Josyula R. Rao, Berk Sunar (editors), *Cryptographic Hardware and Embedded Systems — CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 – September 1, 2005*, Lecture Notes in Computer Science, 3659, Springer, 2005. ISBN 3-540-28474-5. See [15].
30. Martin Šimka, Jan Pelzl, Thorsten Kleinjung, Jens Franke, Christine Priplata, Colin Stahlke, Miloš Drutarovský, Viktor Fischer, *Hardware Factorization Based on Elliptic Curve Method*, in FCCM 2005 [4] (2005), 107–116.
31. Robert Szerwinski, Tim Güneysu, *Exploiting the Power of GPUs for Asymmetric Cryptography*, in CHES 2008 [27] (2008), 79–99.
32. Paul Zimmermann, Bruce Dodson, *20 Years of ECM*, in ANTS 2006 [19] (2006), 525–542. Cited in §1.

Virtex-6 and Spartan-6, plus a Look into the Future

Peter Alfke

Xilinx, USA

Recently, Xilinx introduced two new FPGA families, Virtex-6 and Spartan-6, closely related in architecture, but each optimized for different markets and applications: Virtex-6 for high performance, features and capacity; Spartan-6 for low cost and low power consumption. Both families take advantage of 40/45 nm technology, and both are derived from the successful Virtex-5 architecture. I will give an overview of the salient features and capabilities of both families. Then I will give a peek into the future, explaining the impact of rapidly rising development costs for all future technology nodes. That limits ASICs and ASSPs to serve only high-volume opportunities, but offers unique advantages for FPGAs. But we need to overcome certain technical obstacles and streamline the user's design process.

Efficient FPGA Implementations of High-Dimensional Cube Testers on the Stream Cipher Grain-128

Jean-Philippe Aumasson[1,*], Itai Dinur[2], Luca Henzen[3], Willi Meier[1,†], and Adi Shamir[2]

[1] FHNW, Windisch, Switzerland
[2] Weizmann Institute, Rehovot, Israel
[3] ETH Zurich, Switzerland

Abstract. Cube testers are a generic class of methods for building distinguishers, based on cube attacks and on algebraic property-testers. In this paper, we report on an efficient FPGA implementation of cube testers on the stream cipher Grain-128. Our best result (a distinguisher on Grain-128 reduced to 237 rounds, out of 256) was achieved after a computation involving 2^{54} clockings of Grain-128, with a 256×32 parallelization. An extrapolation of our results with standard methods suggests the possibility of a distinguishing attack on the full Grain-128 in time 2^{83}, which is well below the 2^{128} complexity of exhaustive search. We also describe the method used for finding good cubes (a simple evolutionary algorithm), and report preliminary results on Grain-v1 obtained with a bitsliced C implementation. For instance, running a 30-dimensional cube tester on Grain-128 takes 10 seconds with our FPGA machine, against about 45 minutes with our bitsliced C implementation, and more than a day with a straightforward C implementation.

1 Introduction

The stream cipher Grain-128 was proposed by Hell, Johansson, Maximov, and Meier [16] as a variant of Grain-v1 [17,18], to accept keys of up to 128 bits, instead of up to 80 bits. Grain-v1 has been selected in the eSTREAM portfolio[4] of promising stream ciphers for hardware, and Grain-128 was expected to retain the merits of Grain-v1.

Grain-128 takes as input a 128-bit key and a 96-bit IV, and it produces a keystream after 256 rounds of initialization. Each round corresponds to clocking two feedback registers (a linear one, and a nonlinear one). Several attacks on Grain-128 were reported: [22] claims to detect nonrandomness on up to 313 rounds, but these results were not documented, and not confirmed by [9], which used similar methods to find a distinguisher on 192 rounds. Shortcut key-recovery attacks on 180 rounds were presented in [10], while [5] exploited a sliding property to speed up exhaustive search by a factor two. More recently, [21] presented related-key attacks on the full Grain-128. However, the relevance of related-key attacks is disputed, and no attack significantly faster than bruteforce is known for Grain-128 in the standard attack model.

The generic class of methods known as *cube testers* [1], based on cube attacks [8] and on algebraic property-testers, aims to detect non-randomness in cryptographic algorithms, via multiple queries with chosen values for the IV bits (more generally, referred to as public variables). Both cube attacks and cube testers sum the output of a cryptographic function over a *subset* of its inputs. Over GF(2), this summation can be viewed as high-order differentiation with respect to the summed variables. This property was used in [20] to suggest a general measurement for the strength of cryptographic functions of low algebraic degree. Similar summation methods, along with more concrete cryptanalytic ideas, were later used to attack several stream ciphers. For example, Englund, Johansson, and Turan [9] presented a framework to detect non-randomness in stream ciphers, and in [23] Vielhaber developed a key-recovery attack (called AIDA) on reduced versions of Trivium [6]—another cipher in the eSTREAM portfolio. More recently, generalizations of these attacks were proposed: Cube attacks generalize AIDA as a key-recovery attack on a large variety of cryptographic

*Supported by the Swiss National Science Foundation, project no. 113329.
†Supported by GEBERT RÜF STIFTUNG, project no. GRS-069/07.
[4] See `http://www.ecrypt.eu.org/stream`.

schemes. Cube testers use similar techniques to those used in [9], but are more general. Cube testers were previously applied to Trivium [1], and seem relevant to attack Grain-128, since it also builds on low-degree and sparse algebraic equations.

This paper presents an FPGA implementation of cube testers on Grain-128. We ran 256 instances of Grain-128 in parallel, each instance being itself parallelized by a factor 32. Our heaviest experiment involved the computation of 2^{54} clockings of the Grain-128 mechanism, and detected nonrandomness on up to 237 rounds (out of 256). As an aside, we describe some of the other tools we used: a bitsliced C implementation of cube testers on Grain-128, and a simple evolutionary algorithm for searching "good cubes".

Compared to previous works, our attacks are more robust and general. For example, [5] exploits a sliding property that can easily be avoided, as [5, §3.4] explains: "to eliminate the self-similarity of the initialization constant. If the last 16 bits of the LFSR would for example have been initialized with (0,...,0.1), then this would already have significantly reduced the probability of the sliding property."

2 Brief Description of Grain-128

The mechanism of Grain-128 consists of a 128-bit LFSR, a 128-bit NFSR (both over GF(2)), and a Boolean function h. The feedback polynomial of the NFSR has algebraic degree two, and h has degree three (see Fig. 1).

Given a 128-bit key and a 96-bit IV, one initializes Grain-128 by filling the NFSR with the key, and the LFSR with the IV padded with 1 bits. The mechanism is then clocked 256 times without producing output, and feeding the output of h back into both registers. Details can be found in [16].

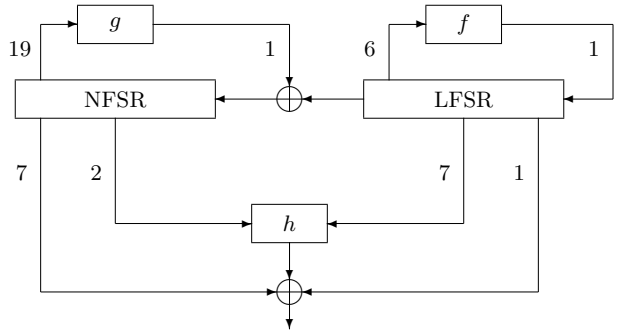


Fig. 1. Schematic view of Grain-128's keystream generation mechanism (numbers designate arities). During initialization, the output bit is fed back into both registers, i.e., added to the output of f and g.

3 Cube Testers

In this section, we briefly explain the principles behind cube testers, and describe the type of cube testers used for attacking Grain-128. More details can be found in [1], and in the article introducing (key-recovery) cube attacks [8].

An important observation regarding cube testers is that for any function $f : \{0,1\}^n \mapsto \{0,1\}$, the sum (XOR) of all entries in the truth table

$$\sum_{x \in \{0,1\}^n} f(x)$$

equals the coefficient of the highest degree monomial $x_1 \cdots x_n$ in the algebraic normal form (ANF) of f. This observation has been used by Englund, Johansson, and Turan [9] for building distinguishers.

For a stream cipher, one may consider as f the function mapping the key and the IV bits to the first bit of keystream. Obviously, evaluating f for each possible key/IV and xoring the values obtained yields the coefficient of the highest degree monomial in the implicit algebraic description of the cipher.

Instead, cube attacks work by summing $f(x)$ over a *subset* of its inputs. For example, if $n = 4$ and

$$f(x) = f(x_1, x_2, x_3, x_4) = x_1 + x_1x_2x_3 + x_1x_2x_4 + x_3 \ ,$$

then summing over the four possible values of (x_1, x_2) yields

$$\sum_{(x_1,x_2)\in\{0,1\}^2} f(x_1, x_2, x_3, x_4) = 4x_1 + 4x_3 + (x_3 + x_4) \equiv x_3 + x_4 \ ,$$

where $(x_3 + x_4)$ is the factor of x_1x_2 in f:

$$f(x_1, x_2, x_3, x_4) = x_1 + x_1x_2(x_3 + x_4) + x_3 \ .$$

Indeed, when x_3 and x_4 are fixed, then the maximum degree monomial becomes x_1x_2 and its coefficient equals the value $(x_3 + x_4)$. In the terminology of cube attacks, the polynomial $(x_3 + x_4)$ is called the *superpoly* of the *cube* x_1x_2. Cube attacks work by detecting linear superpolys, and then explicitly reconstructing them via probabilistic linearity tests [3].

Now, assume that we have a function $f(k_0, \ldots, k_{127}, v_0, \ldots, v_{95})$ that, given a key k and an IV v, returns the first keystream bit produced by Grain-128. For a fixed key $k_0, \ldots, k_{127}$, the sum

$$\sum_{(v_0,\ldots,v_{95})\in\{0,1\}^{96}} f(k_0, \ldots, k_{127}, v_0, v_{95})$$

gives the evaluation of the superpoly of the cube $v_0v_1 \cdots v_{95}$. More generally, one can fix some IV bits, and evaluate the superpoly of the cube formed by the other IV bits (then called the *cube variables*, or CV). Ideally, for a random key, this superpoly should be a uniformly distributed random polynomial. However, when the cipher is constructed with components of low degree, and sparse algebraically, this polynomial is likely to have some property which is efficiently detectable. More details about cube attacks and cube testers can be found in [1, 8].

In our tests below, we measure the *balance* of the superpoly, over 64 instances with distinct random keys.

4 Software Implementation

Since we need to run many independent instances of Grain-128 that operate on bits (rather than bytes or words), a *bitsliced* implementation in software is a natural choice. This technique was originally presented by Biham [2], and can speed up the preprocessing phase of cube attacks (and cube testers) as suggested by Crowley in [7].

To test small cubes, and to perform the search described in §6, we used a bitsliced implementation of Grain-128 that runs 64 instances of Grain-128 in parallel, each with (potentially) different keys and different IV's. We stored the internal states of the 64 instances in two arrays of 128 words of 64 bits, where each bit slice corresponds to an instance of Grain-128, and the i-th word of each array contains the i-th bit in the LFSR (or NFSR) of each instance.

Our bitsliced implementation provides a considerable speedup, compared to the reference implementation of Grain-128. For example, on a PC with an Intel Core 2 Duo processor, evaluating the superpoly of a cube of dimension 30 for 64 distinct instances of Grain-128 with a bitsliced implementation takes approximately 45 minutes, against more than a day with the designers' C implementation. Appendix A gives our C code.

5 Hardware Implementation

Field-programmable gate arrays (FPGA's) are reconfigurable hardware devices widely used in the implementation of cryptographic systems for high-speed or area-constrained applications. The possibility to reprogram the designed core makes FPGA's an attractive evaluation platform to test the hardware performances of selected algorithms. During the eSTREAM competition, many of the candidate stream ciphers were implemented and evaluated, on various FPGA's [4, 11, 13]. Especially for Profile 1 (HW), the FPGA performance in terms of speed, area, and flexibility was a crucial criterion to identify the most efficient candidates.

To attack Grain-128, we used a Xilinx Virtex-5 LX330 FPGA to run the first reported implementation of cube testers in hardware. This FPGA offers a large number of embedded programmable logic blocks, memories and clock managers, and is an excellent platform for large scale parallel computations.

Note that FPGA's have already been used for cryptanalytic purposes, most remarkably with COPACOBANA [15, 19], a machine with 120 FPGA's that can be programmed for exhaustive search of small keys, or for parallel computation of discrete logarithms.

5.1 Implementation of Grain-128

The Grain ciphers (Grain-128 and Grain-v1) are particularly suitable for resource-limited hardware environments. Low-area implementations of Grain-v1 are indeed able to fill just a few slices in various types of FPGA's [14]. Using only shift registers combined with XOR and AND gates, the simplicity of the Grain's construction could also be easily translated into high-speed architectures. Throughput and circuit's efficiency (area/speed ratio) are indeed the two main characteristics that have been used as guidelines to design our Grain-128 module for the Virtex-5 chip. The relatively small degree of optimization for Grain allows the choice of different datapath widths, resulting in the possibility of a speedup by a factor 32 (see [16]).

We selected a 32-bit datapath to get the fastest and most efficient design in terms of area and speed. Fig. 2 depicts our module, where both sides of the diagram contain four 32-bit register blocks. During the setup cycle, the key and the IV are stored inside these memory blocks. In normal functioning, they behave like shift register units, i.e., at each clock cycle the 32-bit vectors stored in the lower blocks are sent to the upper blocks. For the two lowest register blocks (indices between 96 and 127), the input vectors are generated by specific functions, according to the algorithm definition. The g' module executes the same computations of the function g plus the addition of the smallest index coming from the LFSR, while the output bits are entirely computed inside the h' module. Table 1 summarizes the overall structure of our 32×Grain-128 architecture.

Table 1. Performance results of our Grain-128 implementation.

	Frequency [MHz]	Throughput [Mbps]	Size [Slices]	Available area [Slices]
Grain-128 module	200	6,400	180	51,840

5.2 Implementation of Cube Testers

Besides the intrinsic speed improvement from software to hardware implementations of Grain-128, the main benefit resides in the possibility to parallelize the computations of the IV queries necessary for the cube tester. With 2^m instances of Grain-128 in parallel, running a cube tester with a $(n + m)$-dimensional cube will be as fast as with an n-dimensional cube on a single instance.

In addition to the array of Grain-128 modules, we designed three other components: the first provides the pseudorandom key and the 2^n IV's for each instance, the second collects and sums the outputs, and the last component is a controller unit. Fig. 3 illustrates the architecture of our cube tester implementation fitted in

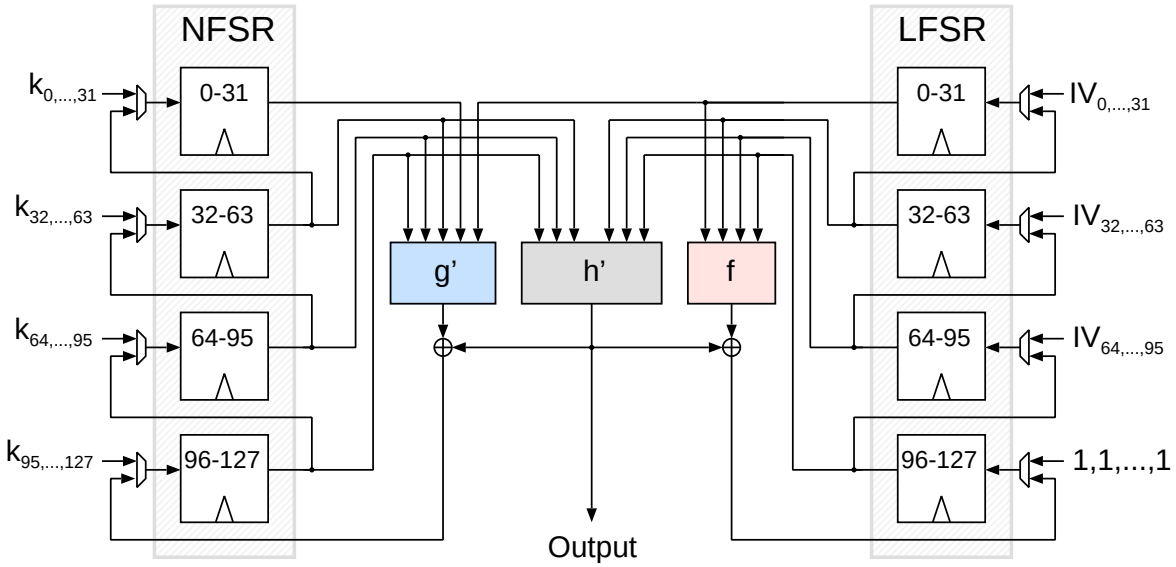


Fig. 2. Overview of our Grain-128 architecture. At the beginning of the simulation, the key and the IV are directly stored in the NFSR and LFSR register blocks. All connections are 32-bit wide.

a Virtex-5 chip. No special macro blocks has been used, we just tried to exploit all the available space to fit the largest Grain-128 array. Below we describe the mode of operation of each component:

- **Simulation controller**: This unit manages the IO interface to control the cube tester core. Through the signal `s_inst`, a new instance is started. After the complete evaluation of the cube over the Grain-128 array, the `u_inst` signal is asserted and later a new instantiation with a different key is started. This operation mode works differently from the software implementation, where the 256 instances run in parallel.
- **Input generator**: After each run of the cipher array, the $(n\text{-}m)$-bit partial IV is incremented by one. This vector is then combined with different m-bit offset vectors to generate the 2^m IVs. The key distribution is also managed here. A single key is given to the parallel Grain-128 modules and is updated only when the partial IV is equal to zero.
- **Output collector**: The outcoming 32-bit vectors from the parallel Grain-128 modules are xored, and the result is xored again with the intermediate results of the previous runs. The updated intermediate results are then stored until the `u_inst` signal is asserted. This causes a reset of the 32-bit intermediate vector and an update of an internal counter.

The m-bit binary representations of the numbers in $0, \ldots, 2^m - 1$ are stored in offset vectors. These vectors are given to the Grain-128 modules as the last cube bits inside the IV. The correct allocation of the CV bits inside the IV is performed by the CV routers. These blocks take the partial IV and the offset vectors to form a 96-bit IV, where the remaining bits are set to zero. When the cube is updated, the offset bits are reallocated, varying the composition of the IV's.

In the input generator, the key is also provided by a LSFR with (primitive) feedback polynomial $x^{128} + x^{29} + x^{27} + x^2 + 1$. This guarantees a period of $2^{128} - 1$, thus ensuring that no key is repeated.

The evaluation of the superpoly for all 256 instances with different pseudorandom keys is performed inside the output collection module. After the 2^{n-m} queries, the intermediate vector contains the final evaluation of the superpoly for a single instance. The implementation of a modified Grain-128 architecture with $\times 32$ speedup allows us to evaluate the same cube for 32 subsequent rounds. That is, after the exhaustive simulation of all possible values of the superpoly, we get the results for the same simulation done with an increasing

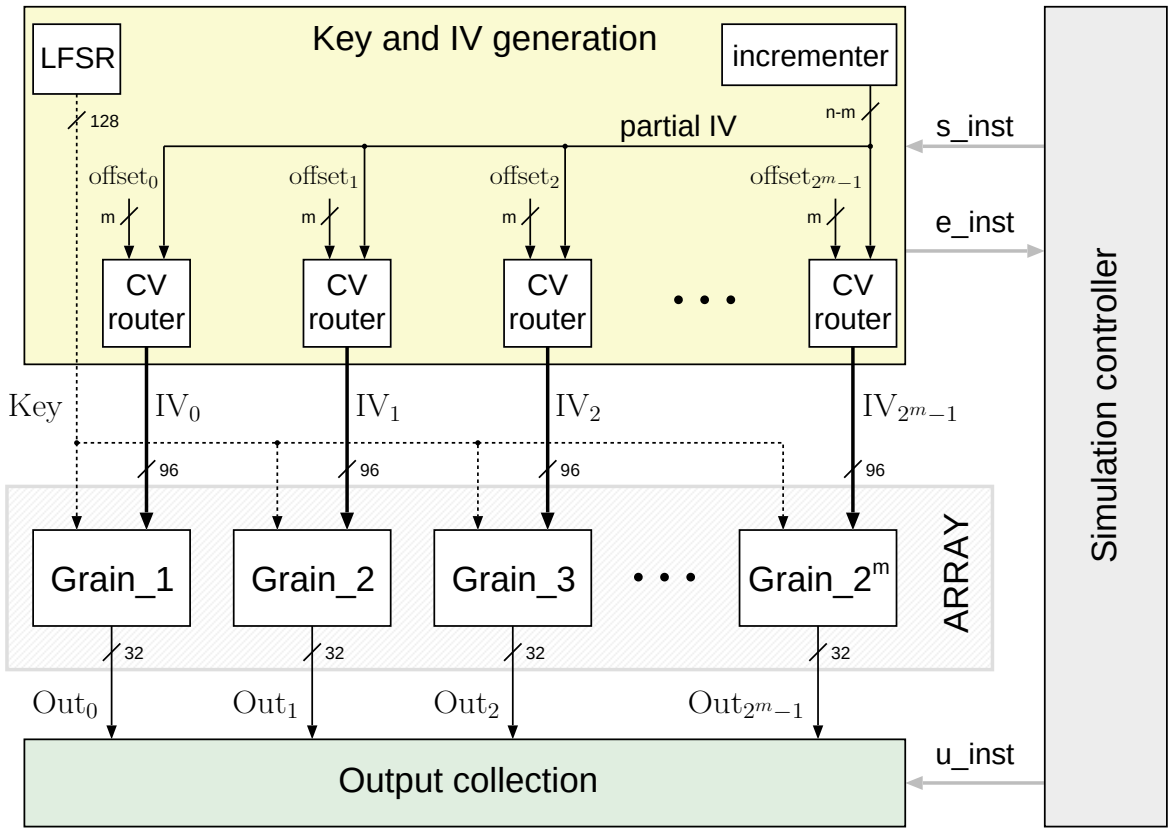


Fig. 3. Architecture of the FPGA cube module. The width of all signals is written out, except for the control signals in grey.

number of initialization rounds r, $32i \leq r < 32(i+1)$ and $i \in [1,7]$. This is particularly useful to test the maximal number of rounds attackable with a specific cube (we don't have to run the same initialization rounds 32 times to test 32 distinct round numbers).

Finally, 32 dedicated counters are incremented if the values of the according bit inside the intermediate result vector is zero or one, respectively. At the end of the repetitions, the counters indicate the proportion between zeros and ones for 32 different values of increasing rounds. This proportion vector can be constantly monitored using an IO logic analyzer.

Since the required size of a single Grain-128 core is 180 slices, up to 256 parallel ciphers can be implemented inside a Virtex-5 LX330 chip (cf. Table 1). This gives $m = 8$, hence decreasing the number of queries to 2^{n-8}. Table 2 presents the evaluation time for cubes up to dimension 50. The critical path has been kept inside the Grain-128 modules, so the working frequency of the cube machine is 200 MHz.

Estimate for an ASIC Implementation The utilization of an application-specific integrated circuit (ASIC) is a further solution to enhance the performances of cube testers on Grain-128. Like in the FPGA, several parallel cipher modules should run at the same time, decreasing the evaluation period of a cube. Using the ASIC results presented in [11,14], we could estimate a speed increase up to 400 MHz for a 90 nm CMOS technology. Evaluating a related area cost of about 10 kGE for a single Grain-128 module (broad estimate), we could take into account a single chip design of 4 mm×4 mm size, hosting the same number of Grain-128 elements of 256. This leads to a similar ASIC cube tester implementation, which is able to compute a cube

Table 2. FPGA evaluation time for cubes of different dimension with $2^m = 2^8$ parallel Grain-128 modules. Note that detecting nonrandomness requires the calculation of statistics on several trials, e.g., our experiments involved 64 trials with a 40-bit cube.

Cube dimension	30	35	37	40	44	46	50
Nb. of queries	2^{22}	2^{27}	2^{29}	2^{32}	2^{36}	2^{38}	2^{42}
Time	0.17 sec	5.4 sec	21 sec	3 min	45 min	3 h	2 days

in half the time of the FPGA. However, in this rough estimate we omitted several problematics related to ASIC design, like the expensive fabrication costs or the development of an interface to communicate the cube indices inside the chip.

6 Search for Good Cubes

To search for cubes that maximize the number of rounds after which the superpoly is still not balanced, we programmed a simple *evolutionary algorithm* (EA). Metaheuristic optimization methods like EA's seem relevant for searching good cubes, since they are generic, highly parametrizable, and are often the best choice when the topology of the search space is unknown. In short, EA's aim to maximize a *fitness function*, by updating a set of points in the search space according to some evolutionary operators, the goal being to converge towards a (local) optimum in the search space.

We implemented in C a simple EA that adapts the evolutionary notions of selection, reproduction, and mutation to cubes, which are then seen as individuals of a population. Our EA returns a set of cubes, and is parametrized by

- σ, the cube dimension, in bits.
- μ, the maximal number of mutations.
- π, the (constant) population size.
- χ, the number of individuals in the offspring.
- γ, the number of generations.

Algorithm 1 gives the pseudocode of our EA, where lines 3 to 5 correspond to the *reproduction*, lines 6 and 7 correspond to the *mutation*, while lines 8 and 9 correspond to the *selection*.

Algorithm 1 uses as fitness function a procedure that returns the highest number of rounds for which it yields a *constant* superpoly. We chose to evaluate the constantness rather than the balance because it reduces the number of parameters, thus simplifying the configuration of the search.

Algorithm 1 Evolutionary algorithm for searching good cubes.

```
1. initialize a population of π random σ-bit cubes
2. repeat γ times
3.     repeat χ times
4.         pick two random cubes □1 and □2 in the population of π cubes
5.         create a new cube with each index chosen randomly from □1 or □2
6.         choose a random number i in {1, ..., μ}
7.         choose i random indices in the new cube, replace them by random indices
8.     evaluate the fitness of the population and of the offspring
9.     replace population by the π best-ranking individuals
10. return the π cubes in the population
```

In practice, we optimized Algorithm 1 with ad hoc tweaks, like initializing cubes with particular "weak" indices, e.g., 33, 66, and 68; we indeed observed that these indices appeared frequently in the cubes found by

a vanilla version of our EA, which suggests that the distribution of monomials containing the corresponding bits tends to be lesser than that of random monomials. We later initialized the population by forcing the use of alleged weak indices in certain individuals, and experimental results did not contradict our conjecture.

Note that EA's can be significantly more complex, notably by using more complicated selection and mutation rules (see [12] for an overview of the topic).

The choice of parameters depends on the cube dimension considered. In our algorithm, the quality of the final result is determined by the population size, the offspring size, the number of generations, and the type of mutation. In particular, increasing the number of mutations favors the exploration of the search space, but too much mutation slows down the convergence to a local optimum. The population size, offspring size, and number of generations are always better when higher, but too large values make the search too slow.

For example, we could find our best 6-dimensional cubes ($\sigma = 6$) by setting $\mu = 3$, $\pi = 40$, $\chi = 80$, and $\gamma = 100$. The search then takes a few minutes. Slower searches did not give significantly better results.

7 Experimental Results

Table 3 summarizes the maximum number of initialization rounds after which we could detect imbalance in the superpoly corresponding to the first keystream bit. It follows that one can mount a distinguisher for 195-round Grain-128 in time 2^{10}, and for 237-round Grain-128 in time 2^{40}. The cubes used are given in Appendix B.

Table 3. Best results for various cube dimensions on Grain-128.

Cube dimension	6	10	14	18	22	26	30	37	40
Rounds	180	195	203	208	215	222	227	233	237

8 Discussion

8.1 Extrapolation

We used standard methods to extrapolate our results, using the generalized linear model fitting of the Matlab tool. We selected the Poisson regression in the "log" value, i.e. logarithm as canonical function and the Poisson distribution, since the achieved results suggested a logarithmic behavior between cube size and number of round. The obtained extrapolation, depicted on Fig. 4, suggests that cubes of dimension 77 may be sufficient to construct successful cube testers on the full Grain-128, i.e., with 256 initialization rounds.

If this extrapolation is correct, then a cube tester making $64 \times 2^{77} = 2^{83}$ chosen-IV queries can distinguish the full Grain-128 from an ideal stream cipher, against 2^{128} ideally. We add the factor 64 because our extrapolation is done with respect to results obtained with statistic over 64 random keys. That complexity excludes the precomputation required for finding a good cube; based on our experiments with 40-dimensional cubes, less than 2^5 trials would be sufficient to find a good cube (based on the finding of good small cubes, e.g., using our evolutionary algorithm). That is, precomputation would be less than 2^{88} initializations of Grain-128.

8.2 The Possibility of Key-Recovery Attacks

To apply key-recovery cube attacks on Grain-128, one must find IV terms with a linear superpoly in the key bits (or maxterms). In general, it is more difficult to find maxterms than terms with a biased superpoly, since one searches for a very specific structure in the superpoly. Moreover, the internal structure of Grain-128 seems to make the search for maxterms particularly difficult for reduced variants of the cipher: Initially, the

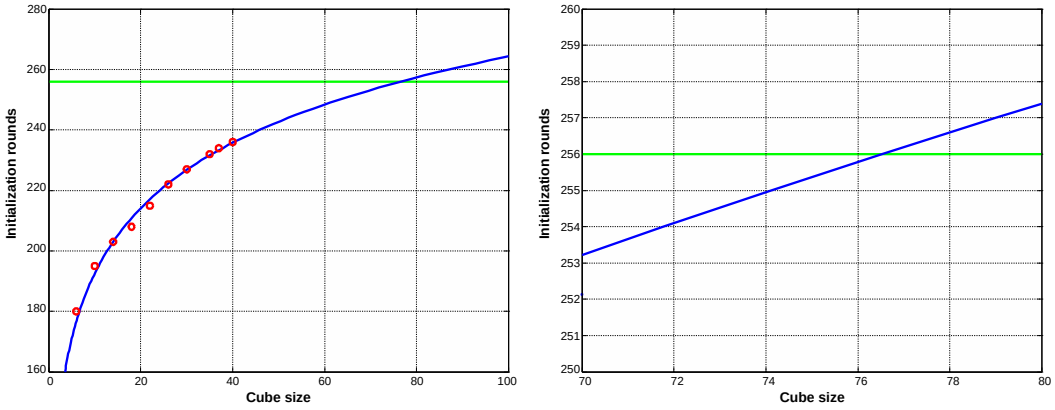


Fig. 4. Extrapolation of our cube testers on Grain-128, obtained by general linear regression using the Matlab software, in the "poisson-log" model. The required dimension for the full Grain-128 version is 77 (see zoom on the right).

key and IV are placed in different registers, and the key bits mix together extensively and non-linearly before mixing with the IV bits. Thus, the output bit polynomials of Grain-128 in the key and IV variables contain very few IV terms whose superpoly is linear in the key bits. A natural way to deal with these polynomials is to apply linearization by replacing non-linear products of key bits with new variables. The linearization techniques are more complicated than the basic cube attack techniques and thus we leave key-recovery attacks on Grain-128 as future work.

8.3 Observations on Grain-v1

Grain-v1 is the predecessor of Grain-128. Its structure is similar to that of Grain-128, but the registers are 80-bit instead of 128-bit, the keys are 80-bit, the IV's are 64-bit, and the initialization clocks the mechanism 160 times (see Appendix C).

The feedback polynomial of Grain-v1's NFSR has degree six, instead of two for Grain-128, and is also less sparse. The filter function h has degree three for both versions of Grain, but that of Grain-v1 is denser than that of Grain-128. These observations suggest that Grain-v1 may have a better resistance than Grain-128 to cube testers, because its algebraic degree and density are likely to converge much faster towards ideal ones.

To support the above hypothesis, we used a bitsliced implementation of Grain-v1 to search for good cubes with the EA presented in §6, and we ran cube testers (still in software) similar to those on Grain-128. Table 4 summarizes our results, showing that one can mount a distinguisher on Grain-v1 with 81 rounds of initialization in 2^{24}. However, even an optimistic (for the attacker) extrapolation of these observations suggests that the full version of Grain-v1 resists cube testers, and the basic cube attack techniques.

Table 4. Best results for various cube dimensions on Grain-v1.

Cube dimension	6	10	14	20	24
Rounds	64	70	73	79	81

9 Conclusion

We developed and implemented a hardware cryptanalytical device for attacking the stream cipher Grain-128 with cube testers (which give distinguishers rather than key recovery). We were able to run our tests on 256 instances of Grain-128 in parallel, each instance being itself parallelized by a factor 32. The heaviest experiment run involved about 2^{54} clockings of the Grain-128 mechanism.

To find good parameters for our experiments in hardware, we first ran light experiments in software with a dedicated bitsliced implementation of Grain-128, using a simple evolutionary algorithm. We were then able to attack reduced versions of Grain with up to 237 rounds. An extrapolation of our results suggests that the full Grain-128 can be attacked in time 2^{83} instead of 2^{128} ideally. Therefore, Grain-128 may not provide full protection when 128-bit security is required.

References

1. Jean-Philippe Aumasson, Itai Dinur, Willi Meier, and Adi Shamir. Cube testers and key recovery attacks on reduced-round MD6 and Trivium. In Orr Dunkelman, editor, *FSE*, LNCS. Springer, 2009. to appear.
2. Eli Biham. A fast new des implementation in software. In Eli Biham, editor, *FSE*, volume 1267 of *LNCS*, pages 260–272. Springer, 1997.
3. Manuel Blum, Michael Luby, and Ronitt Rubinfeld. Self-testing/correcting with applications to numerical problems. In *STOC*, pages 73–83. ACM, 1990.
4. Philippe Bulens, Kassem Kalach, Francois-Xavier Standaert, and Jean-Jacques Quisquater. FPGA implementations of eSTREAM phase-2 focus candidates with hardware profile. Technical Report 2007/024, ECRYPT eSTREAM, 2007.
5. Christophe De Cannière, Özgül Kücük, and Bart Preneel. Analysis of Grain's initialization algorithm. In *SASC 2008*, 2008.
6. Christophe De Cannière and Bart Preneel. Trivium. In *New Stream Cipher Designs*, volume 4986 of *LNCS*, pages 84–97. Springer, 2008.
7. Paul Crowley. Trivium, SSE2, CorePy, and the "cube attack", 2008. Published on `http://www.lshift.net/blog/`.
8. Itai Dinur and Adi Shamir. Cube attacks on tweakable black box polynomials. In Antoine Joux, editor, *EUROCRYPT*, volume 5479 of *LNCS*, pages 278–299. Springer, 2009.
9. Håkan Englund, Thomas Johansson, and Meltem Sönmez Turan. A framework for chosen IV statistical analysis of stream ciphers. In K. Srinathan, C. Pandu Rangan, and Moti Yung, editors, *INDOCRYPT*, volume 4859 of *LNCS*, pages 268–281. Springer, 2007.
10. Simon Fischer, Shahram Khazaei, and Willi Meier. Chosen IV statistical analysis for key recovery attacks on stream ciphers. In Serge Vaudenay, editor, *AFRICACRYPT*, volume 5023 of *LNCS*, pages 236–245. Springer, 2008.
11. Kris Gaj, Gabriel Southern, and Ramakrishna Bachimanchi. Comparison of hardware performance of selected phase II eSTREAM candidates. Technical Report 2007/026, ECRYPT eSTREAM, 2007.
12. David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, 1989.
13. Tim Good and Mohammed Benaissa. Hardware performance of eSTREAM phase-III stream cipher candidates. In *SASC*, 2008.
14. Tim Good, William Chelton, and Mohamed Benaissa. Review of stream cipher candidates from a low resource hardware perspective. Technical Report 2006/016, ECRYPT eSTREAM, 2006.
15. Tim Gueneysu, Timo Kasper, Martin Novotny, Christof Paar, and Andy Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers*, 57(11):1498–1513, 2008.
16. Martin Hell, Thomas Johansson, Alexander Maximov, and Willi Meier. A stream cipher proposal: Grain-128. In *IEEE International Symposium on Information Theory (ISIT 2006)*, 2006.
17. Martin Hell, Thomas Johansson, and Willi Meier. Grain - a stream cipher for constrained environments. Technical Report 2005/010, ECRYPT eSTREAM, 2005.
18. Martin Hell, Thomas Johansson, and Willi Meier. Grain: a stream cipher for constrained environments. *IJWMC*, 2(1):86–93, 2007.
19. Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, and Manfred Schimmler. Breaking ciphers with COPACOBANA - a cost-optimized parallel code breaker. In Louis Goubin and Mitsuru Matsui, editors, *CHES*, volume 4249 of *LNCS*, pages 101–118. Springer, 2006.

20. Xuejia Lai. Higher order derivatives and differential cryptanalysis. In *Symposium on Communication, Coding and Cryptography, in honor of James L. Massey on the occasion of his 60'th birthday*, pages 227–233, 1994.
21. Yuseop Lee, Kitae Jeong, Jaechul Sung, and Seokhie Hong. Related-key chosen IV attacks on Grain-v1 and Grain-128. In Yi Mu, Willy Susilo, and Jennifer Seberry, editors, *ACISP*, volume 5107 of *LNCS*, pages 321–335. Springer, 2008.
22. Sean O'Neil. Algebraic structure defectoscopy. Cryptology ePrint Archive, Report 2007/378, 2007.
23. Michael Vielhaber. Breaking ONE.FIVIUM by AIDA an algebraic IV differential attack. Cryptology ePrint Archive, Report 2007/413, 2007.

A Bitsliced Implementation of Grain-128

We present the C code of a function that, given 64 keys and 64 IV's (already bitsliced), returns the first keystream bit produced by Grain-128 with `rounds` initialization rounds.

```
typedef unsigned long long u64;

u64 grain128_bitsliced64( u64 * key, u64 * iv, int rounds ) {

   u64 l[128+rounds], n[128+rounds], z=0;
   int i,j;

   for(i=0; i<96; i++){
     n[i]= key[i];
     l[i]= iv[i];
   }
   for(i=96; i<128; i++){
     n[i]= key[i];
     l[i]= 0xFFFFFFFFFFFFFFFFULL;
   }
   for(i=0; i<rounds; i++){
     l[i+128] = l[i] ^ l[i+7] ^ l[i+38] ^ l[i+70] ^ l[i+81] ^ l[i+96];
     n[i+128] = l[i] ^ n[i] ^ n[i+26] ^ n[i+56] ^ n[i+91] ^ n[i+96] ^
     (n[i+ 3] & n[i+67]) ^ (n[i+11] & n[i+13]) ^ (n[i+17] & n[i+18]) ^
     (n[i+27] & n[i+59]) ^ (n[i+40] & n[i+48]) ^ (n[i+61] & n[i+65]) ^
     (n[i+68] & n[i+84]);

     z = (n[i+12] & l[i+8]) ^ (l[i+13] & l[i+20]) ^
     (n[i+95] & l[i+42]) ^ (l[i+60] & l[i+79]) ^
     (n[i+12] & n[i+95] & l[i+95]);
     z = n[i + 2] ^ n[i + 15] ^ n[i + 36] ^ n[i + 45] ^ n[i + 64] ^
     n[i + 73] ^ n[i + 89] ^ z ^ l[i + 93];

     l[i+128] ^= z;
     n[i+128] ^= z;
   }

   z = (n[i+12] & l[i+8]) ^ (l[i+13] & l[i+20]) ^
   (n[i+95] & l[i+42]) ^ (l[i+60] & l[i+79]) ^
   (n[i+12] & n[i+95] & l[i+95]);
   z = n[i + 2] ^ n[i + 15] ^ n[i + 36] ^ n[i + 45] ^ n[i + 64] ^
   n[i + 73] ^ n[i + 89] ^ z ^ l[i + 93];

   return z;
}
```

B Cubes for Grain-128

Table 5 gives the indices of the cubes used for finding the results in Table 3.

Table 5. Cubes used for Grain-128.

Cube dimension	Indices
6	33, 36, 61, 64, 67, 69
10	5, 28, 34, 36, 37, 66, 68, 71, 74, 79
14	5, 28, 34, 36, 37, 51, 53, 54, 56, 63, 66, 68, 71, 74
18	5, 28, 30, 32, 34, 36, 37, 62, 63, 64, 65, 66, 67, 68, 69, 71, 73, 74
22	4, 5, 28, 30, 32, 34, 36, 37, 51, 62, 63, 64, 65, 66, 67, 68, 69, 71, 73, 74, 79, 89
26	4, 7, 20, 22, 25, 28, 30, 31, 33, 36, 39, 40, 41, 51, 53, 54, 56, 57, 61, 62, 63, 64, 65, 66, 67, 68
30	4, 7, 20, 22, 25, 28, 30, 31, 33, 36, 39, 40, 41, 51, 53, 54, 56, 57, 59, 62, 65, 66, 69, 72, 75, 78, 79, 80, 83, 86
37	4, 7, 12, 14, 20, 22, 25, 28, 30, 31, 33, 36, 39, 40, 41, 51, 53, 54, 56, 57, 61, 62, 63, 64, 65, 66, 67, 68, 74, 75, 76, 77, 78, 79, 89, 90, 91
40	4, 7, 12, 14, 20, 22, 25, 28, 30, 31, 33, 36, 39, 40, 41, 51, 53, 54, 56, 57, 61, 62, 63, 64, 65, 66, 67, 68, 74, 75, 76, 77, 78, 79, 86, 87, 88, 89, 90, 91

C Grain-v1

Fig. 5 presents the structure of Grain-v1.

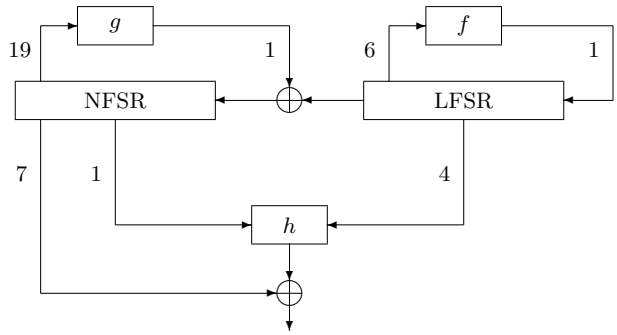


Fig. 5. Schematic view of Grain-v1's keystream generation mechanism (numbers designate arities). During initialization, the output bit is fed back into both registers, i.e., added to the output of f and g.

Cryptanalysis of KeeLoq with COPACOBANA

Martin Novotný[1] and Timo Kasper[2]

[1] Faculty of Information Technology
Czech Technical University in Prague
Kolejní 550/2
160 00 Praha 6, Czech Republic
email: novotnym@fit.cvut.cz
[2] Embedded Security Group
Ruhr-University Bochum
Universitätsstrasse 150,
44801 Bochum, Germany
email: tkasper@crypto.rub.de

Abstract. Many real-world car door systems and garage openers are based on the KeeLoq cipher. Recently, the block cipher has been extensively studied. Several attacks have been published, including a complete break of a KeeLoq access control system. It is possible to instantly override the security of all KeeLoq code-hopping schemes in which the secret key of a remote-control is derived from its serial number. The latter can be intercepted from the communication between a receiver and a transmitter. In contrast, if a random SEED is used for the key derivation, the cryptanalysis demands for higher computation power and may become infeasible with a standard PC.
In this paper we develop a hardware architecture for the cryptanalysis of KeeLoq. Our brute-force attack, implemented on the Cost-Optimized Parallel Code-Breaker COPACOBANA, is able to reveal the secret key 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. To obtain reasonable cryptographic strength against this type of attack, a 60-bit seed has to be used, for which COPACOBANA needs in the worst case about 1011 days for the key recovery. However, the attack is arbitrarily parallelizable and could thus be run on multiple COPACOBANAs to decrease the attack time.

Keywords: KeeLoq, COPACOBANA, cryptanalysis

1 Introduction

Electronic car or garage opening systems consist of remote controls, which replace traditional keys, and receivers which control the door.

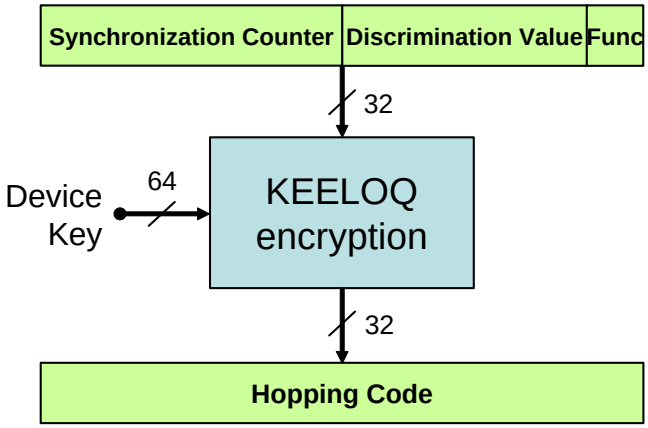


Fig. 1: KeeLoq encryption.

On having its button pressed a remote sends a hopping code to the receiver to open or close the door. A hopping code is generated by a KeeLoq encryption incorporating a 16-bit counter value, a 12-bit discrimination value and a 4-bit function value, as shown in Figure 1. While the counter is incremented in the remote each time a hopping code is generated, the discrimination and function values remain constant.

To obtain the device key on the side of the receiver, the serial number of the remote is either decrypted with a manufacturer key or xored with the manufacturer key, as shown in Figure 2 and in Figure 3a. Alternatively, a randomly generated seed value may by combined with the serial number for the key derivation. For the latter, Microchip proposes three scenarios: a) 28 bits of the serial number (N) are combined with 32 bits of the random seed (S) according to the pattern 0x0NNNNNNNSSSSSSSS (Scenario 2 in Figure 3b), b) 12 bits of the serial number are combined with 48 bits of the seed in the pattern 0x0NNNSSSSSSSSSSSS (Scenario 3 in Figure 3c), c) 60 bits of the seed in the pattern 0x0SSSSSSSSSSSSSSS (Scenario 4 in Figure 3d).

Since the KeeLoq cipher has been extensively studied [1], [2], [3], several different types of attack have been proposed. The attack described in [3] reveals the manufacturer key by means of power analysis. As the manufacturer key is shared by all devices of the same producer and since many commercial products derive the device keys from their serial numbers only (without using a seed), breaking the

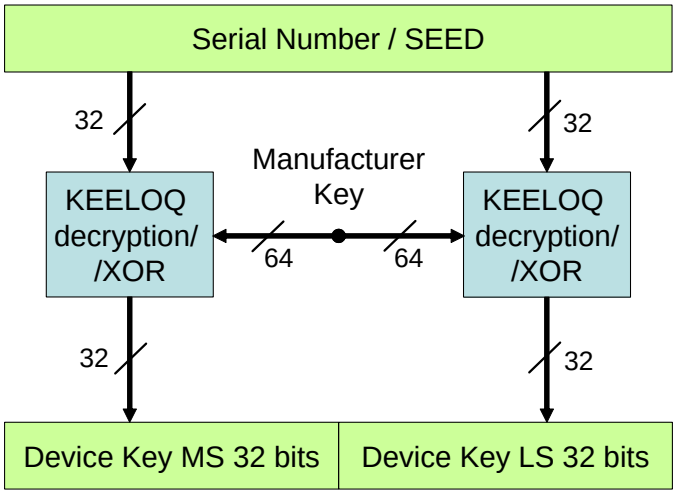


Fig. 2: Device key generation.

system is straightforward — the serial number is intercepted from the communication between the remote and the receiver, and the secret key of the remote is derived (Scenario 1 in Figure 3a).

The goal of this work is finding the correct Device Key when random seed is used for device key generation (Scenarios 2 through 4 in Figure 3). As illustrated in Figure 2, the 32 most significant bits (MSB) of the device key are derived from the higher 32 bits of the input value, while the lower 32 bits are generated from the lower 32 bits of the input. If a random seed is used, lower 32 bits of the device key are always random, while upper 32 bits may have either a fixed value (Scenario 2), or one of 2^{16} potential values (Scenario 3), or one of 2^{28} potential values (Scenario 4). Consequently, when implementing a brute-force attack, each combination of 32 MSBs of the device key may be precomputed in software and then combined with all 2^{32} combinations of 32 LSBs (generated in hardware by a counter), until the correct value of the device key is found.

2 KeeLoq Breaker

To break the cipher we need to intercept two hopping codes of the same device, generated from the same device key. Such hopping codes are generated from identical discrimination and function values, but from different counter values (see Figure 1). However, the difference

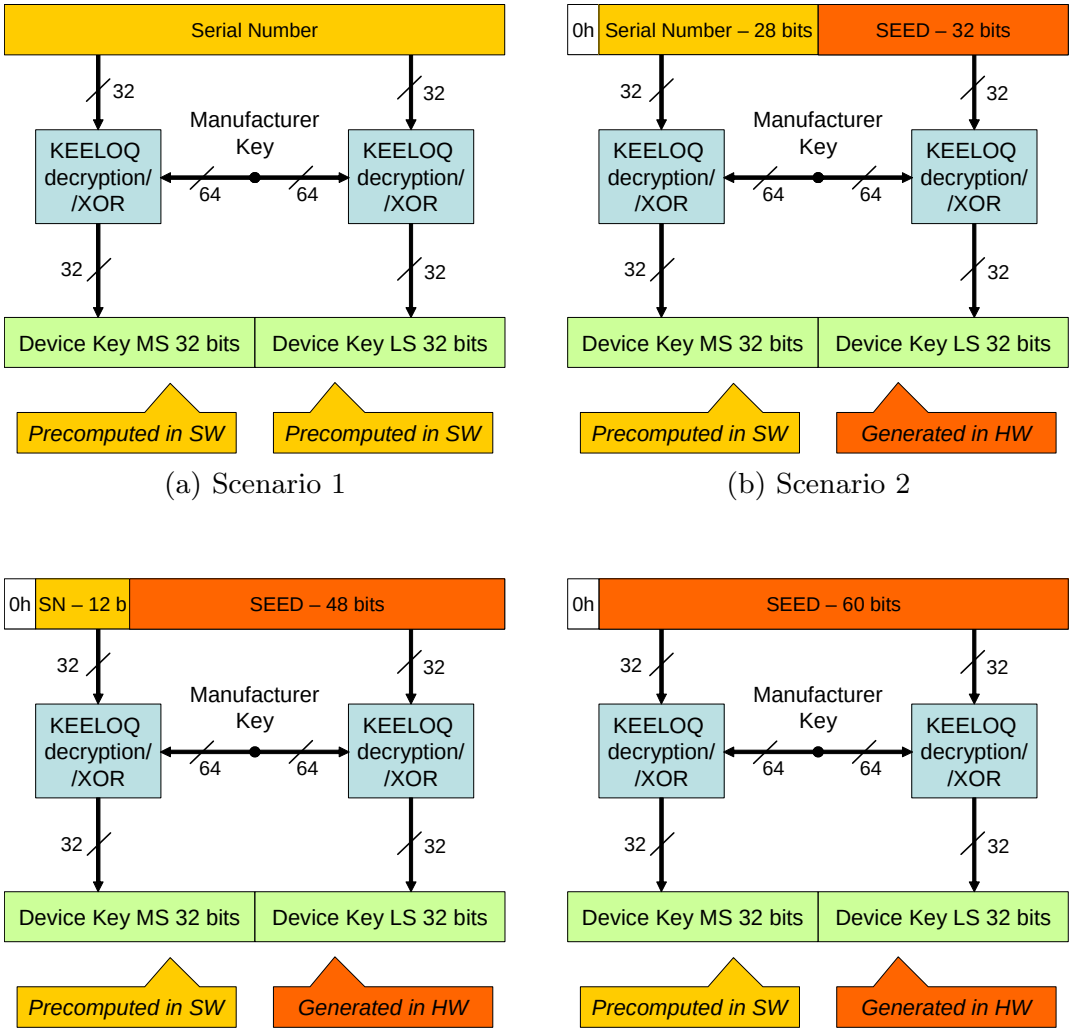


Fig. 3: Scenarios for device key generation.

between the counter values will be small, if the two consecutive (or almost consecutive) hopping codes are intercepted.

We implemented a brute-force attack on KeeLoq on the parallel computation cluster COPACOBANA [4]. This cluster has been designed to support cryptanalytical calculations. The cluster is equipped with 120 low-cost Xilinx Spartan3-1000 FPGAs, which communicate with the host computer via the controller board. Note, that it is possible to employ several COPACOBANAs in order to further increase the performance.

The diagram of the circuit implemented in each FPGA is shown in Figure 4. A candidate for the device key is found by means of

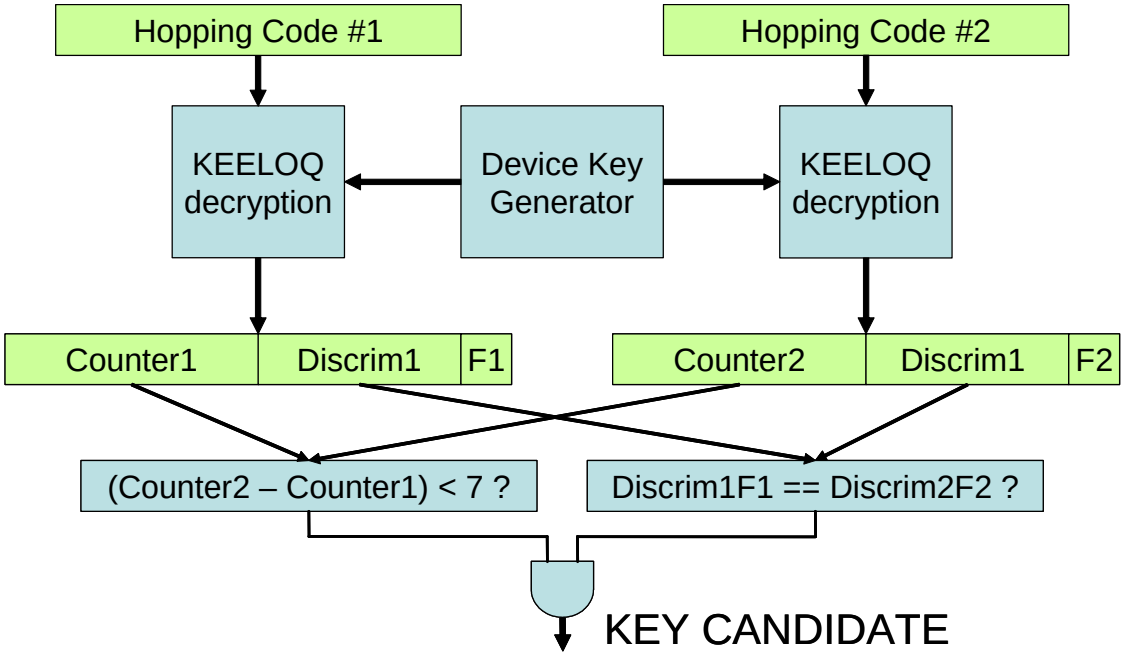


Fig. 4: KeeLoq breaker.

exhaustive key-search, if the decryptions of two intercepted hopping codes reveal identical discrimination values and moderately increased counter values.

The core of the implementation is a *Device Key Generator* consisting of a 32-bit register and a 32-bit counter. The register holds 32 MSBs of the device key (being precomputed in software and assigned by the host computer), while the counter is repeatedly increased to generate all possible values for the lower 32 bits of the device key. If all counter-values have been generated, and no key candidate has been found, the FPGA is assigned with the new value of upper 32 bits of the key.

A KeeLoq decryption is executed in 132 rounds. In our optimized implementation we unrolled both decryption units into a pipeline structure. Each path of the pipeline consists of 176 stages, i.e., each stage contains 4 rounds of the cipher (the number of stages was limited by available resources). The KeeLoq breaker occupies 6423 out of 7680 slices (83%) of the Xilinx Spartan 3-1000 FPGA. The maximum achievable clock frequency for the COPACOBANA was 110 MHz, i.e., each FPGA can test up to 110 million keys per second.

SEED length (bits)	1 FPGA (< 80 $)	1 COPACOBANA (< 10000 $)	100 COPACOBANAs (< 1000000 $)
32	39 secs	0.33 secs	3.3 msecs
48	29.6 days	5.9 hours	213 secs
60	332 years	1011 days	10.1 days

Table 1: Worst case times for the brute force attack on KeeLoq

3 Results and Conclusions

When a 32-bit seed is used, up to 2^{32} potential values of the device key need to be tested, in order to find the correct one. This takes $\frac{2^{32}}{120\times110\cdot10^6} \approx 0.33$ seconds on one COPACOBANA in the worst case. Finding the correct device key in case of a 48-bit seed takes up to $\frac{2^{48}}{120\times110\cdot10^6}$ seconds ≈ 5.9 hours on one COPACOBANA. For the 60-bit seed we need up to $\frac{2^{60}}{120\times110\cdot10^6}$ seconds ≈ 1011 days on one COPACOBANA. The attack is arbitrarily parallelizable and could thus be run on multiple COPACOBANAs to decrease the attack time. Worst case times for all possible seed lengths, and 1 FPGA, 1 COPACOBANA and 100 COPACOBANAs, respectively, are summarized in Table 1.

We conclude that using a 32-bit seed provides no security, since a key can be found in real-time. While a seed with 48 bits can be broken in less than 6 hours by one COPACOBANA, employing a 60-bit seed can provide reasonable security.

References

1. A. Bogdanov, "Attacks on the KeeLoq Block Cipher and Authentication Systems," in *3rd Conference on RFID Security 2007 (RFIDSec 2007)*, 2007. [Online]. Available: http://rfidsec07.etsit.uma.es/slides/papers/paper-22.pdf.
2. S. Indesteege, N. Keller, O. Dunkelman, E. Biham, and B. Preneel, "A Practical Attack on KeeLoq," in *Advances in Cryptology - EUROCRYPT 2008*, 2008.
3. T. Eisenbarth, T. Kasper, A. Moradi, C. Paar, M. Salmasizadeh, and M. T. M. Shalmani, "On the Power of Power Analysis in the Real World: A Complete Break of the KeeLoq Code Hopping Scheme." in *Advances in Cryptology - CRYPTO 2008*, 2008, pp. 203–220.
4. S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler, "Breaking Ciphers with COPACOBANA - A Cost-Optimized Parallel Code Breaker," in *Proceedings of CHES'06*, ser. LNCS, vol. 4249. Springer-Verlag, 2006, pp. 101–118.