SHARCS 2012

Special-purpose Hardware for Attacking Cryptographic Systems

17-18 March 2012
Washington D.C., U.S.A.

Organized by

Program Chairs:

Daniel J. Bernstein, University of Illinois at Chicago, USA
Kris Gaj, George Mason University, USA

Program Committee:

Daniel J. Bernstein, University of Illinois at Chicago, USA
Duncan A. Buell, University of South Carolina, USA
Kris Gaj, George Mason University, USA
Tim Güneysu, Ruhr-Universität Bochum, Germany
Tetsuya Izu, Fujitsu Laboratories, Japan
Tanja Lange, Technische Universiteit Eindhoven, Netherlands
Christof Paar, Ruhr-Universität Bochum, Germany
Christian Rechberger, Danmarks Tekniske Universitet, Denmark
Rainer Steinwandt, Florida Atlantic University, USA
Eran Tromer, Tel Aviv University, Israel
Ingrid Verbauwhede, Katholieke Universiteit Leuven, Belgium
Michael Wiener, Irdeto, Canada
Bo-Yin Yang, Academia Sinica, Taiwan

Subreviewers:

Jens Hermans
Markus Kasper
Nele Mentens
Amir Moradi
Anthony Van Herrewege
Frederik Vercauteren
Christopher Wolf
Tolga Yalcin
Ralf Zimmermann

Local Organization:

Jens-Peter Kaps, George Mason University, USA

Invited Speakers:

Stephen Budiansky, USA
Joe Hurd, Sally A. Browning, Galois, Inc., USA
Marc Stevens, Centrum voor Wiskunde en Informatica (CWI), Netherlands

Contributors:

Lejla Batina, Katholieke Universiteit Leuven, Belgium
Daniel J. Bernstein, University of Illinois at Chicago, USA
Alex Biryukov, University of Luxembourg, Luxembourg
Andrey Bogdanov, Katholieke Universiteit Leuven, Belgium
Hsieh-Chung Chen, Harvard University, USA
Chen-Mou Cheng, National Taiwan University, Taiwan
Tung Chou, Academia Sinica, Taiwan
Nicolas T. Courtois, University College London, UK
Itai Dinur, Weizmann Institute, Israel
Ian Goldberg, University of Waterloo, Canada
Johann Großschädl, University of Luxembourg, Luxembourg
Tim Güneysu, Ruhr-Universität Bochum, Germany
Ryan Henry, University of Waterloo, Canada
Daniel Hulme, University College London and NP-Complete Ltd
Tetsuya Izu, Fujitsu Laboratories, Japan
Lyndon Judge, Virginia Tech, USA
Elif Bilge Kavun, Ruhr-Universität Bochum, Germany
Jun Kogure, Fujitsu Laboratories, Japan
Tanja Lange, Technische Universiteit Eindhoven, Netherlands
Theodosis Mourouzis, University College London, UK
Ruben Niederhagen, Academia Sinica and Technische Universiteit Eindhoven
Christof Paar, Ruhr-Universität Bochum, Germany
Christian Rechberger, Danmarks Tekniske Universitet, Denmark
Peter Schwabe, Academia Sinica, Taiwan
Patrick Schaumont, Virginia Tech, USA
Adi Shamir, Weizmann Institute, Israel
Takeshi Shimoyama, Fujitsu Laboratories, Japan
Martijn Sprengers, Katholieke Universiteit Leuven, Belgium
Tolga Yalcin, Ruhr-Universität Bochum, Germany
Bo-Yin Yang, Academia Sinica, Taiwan
Masaya Yasuda, Fujitsu Laboratories, Japan
Ralf Zimmermann, Ruhr-Universität Bochum, Germany

Program and Table of Contents:

Saturday 17 March

Sunday 18 March

CAESAR: Cryptanalysis of the Full AES Using GPU-Like Hardware*

Alex Biryukov and Johann Großschädl
University of Luxembourg
Laboratory of Algorithmics, Cryptology and Security (LACS)
6, rue Richard Coudenhove-Kalergi, L–1359 Luxembourg
{alex.biryukov,johann.groszschaedl}@uni.lu

Abstract: The block cipher Rijndael has undergone more than ten years of extensive cryptanalysis since its submission as a candidate for the Advanced Encryption Standard (AES) in April 1998. To date, most of the publicly-known cryptanalytic results are based on reduced-round variants of the AES (respectively Rijndael) algorithm. Among the few exceptions that target the full AES are the Related-Key Cryptanalysis (RKC) introduced at ASIACRYPT 2009 and attacks exploiting Time-Memory-Key (TMK) trade-offs such as demonstrated at SAC 2005. However, all these attacks are generally considered infeasible in practice due to their high complexity (i.e. $2^{99.5}$ AES operations for RKC, 2^{80} for TMK). In this paper, we evaluate the cost of cryptanalytic attacks on the full AES when using special-purpose hardware in the form of multi-core AES processors that are designed in a similar way as modern Graphics Processing Units (GPUs) such as the NVIDIA GT200b. Using today's VLSI technology would allow for the implementation of a GPU-like processor reaching a throughput of up to 10^{12} AES operations per second. An organization able to spend one trillion US$ for designing and building a supercomputer based on such processors could theoretically break the full AES in a time frame of as little as one year when using RKC, or in merely one month when performing a TMK attack. We also analyze different time-cost trade-offs and assess the implications of progress in VLSI technology under the assumption that Moore's law will continue to hold for the next ten years. These assessments raise some concerns about the long-term security of the AES.

1 Introduction

Research on special-purpose hardware for cryptanalysis has a rich and illustrious history stretching back almost hundred years [25, 40]. In 1938, Polish mathematicians led by Marian Rejewski constructed the *Bomba Kryptologiczna* (or Bomba for short), an electromechanical machine that allowed them to break the German Enigma cipher by exhaustively trying all 17,576 rotor positions. This success was expanded by British cryptographers (most notably Alan Turing and Gordon Welchman), who designed ingenious cipher-breaking machines enabling the Allied forces to read Enigma-encrypted messages during World War II [44]. A parallel effort of cryptanalysis of another German cipher, the Lorenz SZ40/42, resulted in the construction of *Colossus*, one of the world's first programmable computers. Colossus contained 1,500 thermionic valves (vacuum tubes) and was able to process 5,000 characters per second.

In the 1980s, Pomerance et al [35] designed a hardware architecture called *Quasimodo* for factoring large integers using the quadratic sieve algorithm. Quasimodo was actually built but never functioned

*A revised version of this paper will be published in *Fundamenta Informaticae*, vol. 114, no. 3–4, pp. 221–237 under the title "Cryptanalysis of the Full AES Using GPU-Like Special-Purpose Hardware." Readers are requested to use the version in *Fundamenta Informaticae* for citation purposes.

properly. The *DES Cracker* (also known as *Deep Crack*) is a parallel key-search machine developed by the Electronic Frontier Foundation (EFF) in the late 1990s with an overall budget of just 210,000 US$ [14]. Deep Crack consists of about 1,500 custom chips and needs at most nine days to find a 56-bit DES key by "brute force." Also in the late 1990s, Shamir [38] proposed *TWINKLE*, an electro-optical device for performing the sieving step of the Number Field Sieve (NFS) algorithm. He estimated that a single chip of the size of a 6-inch GaAs wafer would allow one to factor a 512-bit number in reasonable time and, as a consequence, break 512-bit RSA keys. *TWIRL*, a successor of TWINKLE, could reduce the total sieving time for 512-bit numbers to less than ten minutes [39]. Even though both TWINKLE and TWIRL are purely hypothetical devices that were never built due to technical issues (e.g. too large chip area) and high cost, they received considerable attention in the cryptographic community and initiated a slew of follow-up research [16, 17]. Recent attempts to implement cryptanalytic devices mainly use FPGAs as underlying hardware platform [30, 31]. A typical example is *COPACOBANA*, an FPGA-based parallel machine for cryptanalysis that was successful in breaking some symmetric ciphers with a key size of up to 64 bits, e.g. KeeLoq, DES, and A5/1 [27, 21].

The advent of powerful yet inexpensive multi-core processors, especially Graphics Processing Units (GPUs), has triggered a large body of research to analyze their capabilities for cryptanalytic purposes [20]. GPUs are particularly well suited for the implementation of multiplication-intensive cryptanalytic algorithms that can be mapped efficiently onto a highly parallel architecture [3, 4]. Other cryptosystems performing mainly logical operations have also been successfully attacked on various high-end graphics platforms [12, 32]. While such software-based cryptanalysis on multi-core processors is relatively easy to implement, it does not reflect the potential of custom hardware, simply because GPUs are optimized for graphics (or multimedia) processing and not for cryptanalysis. The same holds, although to a lesser extent, for FPGA-based cryptanalytic hardware: An ASIC designed and optimized from the ground up to break a certain cryptosystem can reach higher clock frequencies (and consumes less power) than an FPGA implementing the same functionality.

In this paper, we study the cost of a hardware-based attack on AES-128 and AES-256 assuming that the attack complexity is bounded by 2^{100} computations. We are motivated by the recent progress in the cryptanalysis of AES-256 [7], which has provided an attack with a time and data complexity of $2^{99.5}$ and a memory complexity of 2^{78} in the related-key scenario (we will call this attack RKC, an abbreviation for *Related-Key Cryptanalysis*). While it is clear that implementing this highly specific attack has little practical impact due to its reliance on related keys, we believe that a more threatening secret-key attack of complexity 2^{100} would not be much different in terms of hardware implementation cost, and thus we can use the existing attack a case study. We also notice that the same hardware can, to a large extent, be re-used for a *Time-Memory-Key* attack (TMK, also known as multiple target attack) on AES-128 [9]. In such an attack it is assumed that a fixed plaintext is encrypted under many different secret keys, and the goal of the attacker is to find one of these keys. Given e.g. 2^{32} targets, the TMK attack has a one-time pre-computation complexity of 2^{96}, after which each new secret key can be found with a time complexity of 2^{80} and a memory complexity of 2^{56}.

We evaluate the cost of these two attacks on the full AES assuming special-purpose hardware in the form of multi-core AES processors that are realized in a similar way as modern Graphics Processing Units (GPUs) such as the NVIDIA GT200b [33]. Using state-of-the-art VLSI technology, it is possible to implement a GPU-like processor reaching a throughput of up to 10^{12} AES operations per second at a cost of only about 30 US$. An organization able to spend one trillion US$ (which is roughly a single-year defense budget of the US [42]) for designing and building a large-scale supercomputer based on such optimized processors could theoretically break the full 256-bit AES in a time frame of as little as one year when using RKC or another attack of similar complexity. One tenth of this budget (100 billion

US$) would be sufficient to mount a TMK attack on 2^{32} targets. This attack requires a one-time pre-computation phase of one year, after which a new key can be recovered every 2^{80} AES operations, or every eight minutes, on this "smaller" supercomputer.

Our contribution in this paper is twofold. First, we present the architecture of a GPU-like multi-core AES processor optimized for RKC and TMK attacks and discuss how the requirements for this special processor differ from that of standard high-speed AES hardware with respect to pipelining options, support for key agility, and memory bandwidth. Second, we analyze the production cost and performance of large-scale cryptanalytic hardware built of GPU-like processors and estimate the running time of the RKC and TMK attack on the full AES. More precisely, we try to provide a realistic lower bound for the time and energy required to perform these attacks when using a cryptanalytic supercomputer consisting of 10^{10} optimized AES processors. This supercomputer, which we call *CAESAR* ("Cryptanalytic AES ARchitecture"), is a hypothetical machine (like TWINKLE and TWIRL) since an actual implementation of the proposed GPU-like AES processor is beyond our resources. However, we point out that, unlike TWINKLE, our AES processor can be implemented with present-day VLSI technology since its silicon area, clock frequency, and power consumption are quite similar to that of a commodity GPU. We also analyze different time-cost trade-offs and try to assess the implications of progress in VLSI technology under the assumption that Moore's law will continue to hold for the next ten years.

2 Cryptanalytic Attacks on AES

The cipher Rijndael has been a subject of intensive cryptanalysis already during the AES-competition and in the past ten years after it has been adopted as a new encryption standard by NIST. In this section, we highlight the main advances in cryptanalysis of the AES and describe two of these approaches in more detail.

The first cryptanalysis of 6-round AES was provided by its designers [13], who have shown how to break six rounds of AES-128 using a *multiset attack* (historically called Square attack). During the AES competition, two other attacks were described. The first was the *partial-sum* approach [15], which used the same ideas as the designer's attack, but managed to reduce the time complexity for a 6-round attack from 2^{70} to 2^{44}. The second was a novel *functional-collision* technique [18], which also falls into a class of multiset attacks and is capable of breaking up to seven rounds of AES-128 marginally faster than an exhaustive key search. Rijndael was announced as a NIST standard in November 2001. In the following eight years, there were many attempts of cryptanalysis (boomerang attack, impossible differentials, algebraic attack, various related-key attacks); however, the progress was very slow and mainly restricted to related-key attacks on 192 and 256-bit AES. The best of these attacks reached seven rounds (out of ten) for AES-128, and ten (out of 12 or 14) rounds for AES-192 and AES-256, all with complexities close to that of an exhaustive search.

In 2009, new related-key and open-key attacks capable of breaking the *full* AES-192 and AES-256 were discovered. The attack on 256-bit AES initially had a complexity of 2^{96} data and time and worked for one out of 2^{35} keys, i.e. it had a total complexity of 2^{131} steps [8]. This attack used simple key relations of the form $K' = K \oplus C$, where K, K' are two unknown but related keys and C is a constant chosen by the attacker. In Section 2 of the same paper, a practical chosen-key distinguisher for AES-256 was demonstrated. Later in the same year, the attack on AES-256 was significantly improved to run with a time and data complexity of $2^{99.5}$ using a boomerang related-key attack and the *related subkey* setting in which $K' = F^{-1}(F(K) \oplus C) = R_C(K)$, where function F is one round of the AES-256 key-schedule and C is a constant chosen by the attacker [7]. Even though these attacks reveal a certain structural weakness of the key-schedule of AES-192 and AES-256, they are of no immediate threat in practice due to two

Table 1: Summary of attacks on AES.

Cipher	Attack/Result	Rounds	Data	Workload	Memory	Reference
AES-128	Multiset	6	2^{33}	2^{70}	2^{32}	[13]
	Collisions	7	2^{32}	2^{128}	2^{80}	[18]
	Partial sum	6	2^{35}	2^{44}	2^{32}	[15]
	Partial sum	7	$2^{128}-2^{119}$	2^{120}	2^{32}	[15]
	Boomerang	6	2^{71}	2^{71}	2^{33}	[6]
	Impossible diff.	7	$2^{112.2}$	$2^{117.2}$	2^{109}	[29]
	Boomerang - RK	7	2^{97}	2^{97}	2^{34}	[10]
AES-192	Rectangle - RK	9	2^{64}	2^{143}	?	[19]
	Rectangle - RK	10	2^{125}	2^{182}	?	[26]
	Boomerang - RK	12	2^{116}	2^{169}	2^{145}	[10]
AES-256	Rectangle - RK	10	2^{114}	2^{173}	?	[5, 26]
	Subkey Diff.	10	2^{48}	2^{49}	2^{33}	
	Differential - RK[a]	14	2^{131}	2^{131}	2^{65}	[8]
	Boomerang - RK	14	$2^{99.5}$	$2^{99.5}$	2^{78}	[7]

[a] The attack works for a weak key class, and the workload includes the effort to find related keys from the class.

factors: First, the related-key scenario is a very strong attacker model and, second, the attack requires a huge amount of both time and data. Nonetheless, they are the first attacks on the AES that have broken through the psychological 2^{100} complexity barrier, which may motivate cryptanalysts to pay increased attention to the AES in the coming years. A summary of attacks on AES is given in Table 1.

A completely different approach to cryptanalysis of block ciphers is possible by exploiting a Time-Memory-Key (TMK) trade-off. Such trade-offs can be used to invert any one-way function. The original Time-Memory trade-off was introduced by Hellman [22] and required a single pre-computation equal in complexity to the full exhaustive search. Later, Biryukov and Shamir [11] presented a Time-Memory-Data trade-off as a generalization of Hellman's method, which added more flexibility by introducing an extra *data* parameter into the trade-off equations and, as a consequence, allowed to considerably reduce the heavy pre-computation phase of the original trade-off. In [9], an application of trade-off attacks to multiple target attacks on block ciphers has been studied. The introduced TMK attack requires a fixed plaintext to be encrypted under D unknown secret keys and the goal of the attacker is to find one of these keys. It was shown that, in this scenario, it is impossible for a cipher with a keylength of n bits to stand to its complexity guarantee of 2^n since any cipher can be broken in time $O(2^n/D)$ and with considerably less memory (i.e. a lot better than what a straightforward birthday trade-off would suggest).

The following two subsections summarize the details of the related-key and trade-off attack on the full AES that are needed to understand our estimates for a large-scale hardware implementation of these attacks. The reader is referred to the original papers [7, 9] for a full description.

2.1 Summary of the Related-Key Boomerang Attack on AES-256

This type of attack is embedded in a scenario of four secret related keys; it needs $2^{99.5}$ time and data to find these keys. The attack works as follows. Repeat the following steps $2^{25.5}$ times:

1. Prepare a structure with all possible values in column 0, row 1, and the main diagonal (nine bytes in total), and constant values in the other bytes (2^{72} plaintexts).
2. Encrypt it on keys K_A and K_B and keep the resulting sets S_A and S_B in memory.

3. XOR Δ_C to all the ciphertexts in S_A and decrypt the resulting ciphertexts with K_C. Denote the new set of plaintexts by S_C.
4. Repeat previous step for the set S_B and the key K_D. Denote the set of plaintexts by S_D.
5. Compose from S_C and S_D all the possible pairs of plaintexts which are equal in certain 56 bits.
6. For every remaining pair check if the difference in $p_{i,0}, i > 1$ is equal on both sides of the boomerang quartet (16-bit filter).
7. Filter out the quartets whose difference can not be produced by active S-boxes in the first round (one-bit filter per S-box per key pair) and active S-boxes in the key schedule (one-bit filter per S-box), which is a $2 \cdot 2 + 2 = 6$-bit filter.
8. Gradually recover key values and differences simultaneously filtering out the wrong quartets.

The time and memory complexity of this attack can be evaluated as follows. From 2^{72} texts per structure we could compose 2^{144} ordered pairs, of which $2^{144-8\cdot 9} = 2^{72}$ pairs pass the first round. Thus, we expect one right quartet per $2^{96-72} = 2^{24}$ structures, and three right quartets out of $2^{25.5}$ structures. Let us now calculate the number of noisy quartets. Roughly $2^{144-56-16} = 2^{72}$ pairs come out of step 6. The next step applies a 6-bit filter, which means we get $2^{72+25.5-6} = 2^{91.5}$ candidate quartets in total. Note that we still *do not store these quartets*. For each quartet we check what key candidates it proposes on both sides of the boomerang; this process allows us to gradually reduce the number of candidate quartets to $2^{72.5}$ as shown in [7]. Each candidate quartet proposes 2^6 candidates for 11 key bytes for each of the four related keys. However, these bytes are strongly related, and so the number of independent key bytes on which the voting is performed is significantly smaller than 11×4, namely about 15. The probability that three wrong quartets propose the same candidates does not exceed 2^{-80}. Thus, it can be estimated that the complexity of the filtering step is 2^{78} in time and memory. In total, we recover $3 \cdot 7 + 8 \cdot 8 = 85$ bits of K_A (and 85 bits of K_C) with $2^{99.5}$ data and time and 2^{78} memory. The rest of the key bits can be recovered with negligible complexity compared to the main phase of the attack.

2.2 Summary of a Time-Memory-Key Attack on AES-128

Hellman's trade-off [22] is a well-known way to invert arbitrary one-way functions. The main idea is to iterate a one-way function on its own output, thereby computing a chain, and then to discard all the computed points keeping only the start-end point pairs in memory, sorted by the end points. During the attack, the attacker iterates the function starting from the given ciphertext and checks at each step if he has hit one of the end points[1]. He then picks the corresponding starting point and iterating from it finds the pre-image from the function. In the context of block ciphers with reasonably long keys this attack is typically not considered to be of a threat since the pre-computation time needed to build the tables containing all start-end points is the same as the exhaustive search of the key.

The main idea of the Time-Memory-Key (TMK) trade-off is that we can cover only a fraction of the search space if multiple targets are available. This is typically the case when messages (or files) with the same constant header are encrypted under different keys. Let us denote by $N = 2^n$ the size of the search space ($n = 128$ for AES-128). Given encryptions of a fixed plaintext under D_k different keys, we need to cover only N/D_k points of the space. Hence, we will use t/D_k tables instead of t, which means the memory requirements drop to $M = mt/D_k$ (here m is the number of start-end points in one Hellman's

[1] The need to perform memory access at each iteration can be avoided by using the idea of distinguished points, i.e. the attacker does not perform memory access until a point he obtained has some distinguishing feature (e.g. l leading zeroes).

Table 2: Comparison of TMD attacks on various ciphers.

Cipher	Key size	Keys (Data)	Time	Memory	Preprocessing
DES	56	2^{14}	2^{28}	2^{28}	2^{42}
Triple-DES	168	2^{42}	2^{84}	2^{84}	2^{126}
Skipjack	80	2^{32}	2^{32}	2^{32}	2^{48}
AES	128	2^{32}	2^{80}	2^{56}	2^{96}
AES	192	2^{48}	2^{96}	2^{96}	2^{144}
AES	256	2^{85}	2^{170}	2^{85}	2^{170}
Any cipher	k	$2^{k/4}$	$2^{k/2}$	$2^{k/2}$	$2^{3k/4}$
Any cipher	k	$2^{k/3}$	$2^{2k/3}$	$2^{k/3}$	$2^{2k/3}$

Table 3: Trade-off attacks on 128-bit key AES (and any other 128-bit key cipher).

Attack	Data Type	Keys (Data)	Time	Memory	Preprocessing
BS TMD	FKP	2^{8}	2^{120}	2^{60}	2^{120}
BS TMD	FKP	2^{20}	2^{100}	2^{58}	2^{108}
BS TMD	FKP	2^{32}	2^{80}	2^{56}	2^{96}
BS TMD	FKP	2^{43}	2^{84}	2^{43}	2^{85}

table). The time requirements $T = t/D_k \cdot t \cdot D_k = t^2$ are the same as in Hellman's original trade-off (we have less tables to check, but for more data points). Finally, the matrix stopping rule is $N = mt^2$, which is the condition to minimize the "waste" in the matrix coverage due to birthday paradox effects. Using this matrix stopping rule and eliminating the parameters m and t, we get the trade-off formula

$$N^2 = T(MD_k)^2.$$

Taking AES with 128-bit key as example and assuming an attacker given 2^{32} encryptions of a fixed text under different unknown keys, he can recover one of these keys after a single pre-processing of 2^{96} steps and using 2^{56} memory entries for table storage (2^{61} bytes) and 2^{80} time for the actual key-search.

Another important observation is that the attack is not exactly a chosen plaintext attack since the specific value of the fixed plaintext is irrelevant. Thus, in order to obtain an attack faster than exhaustive search, the attacker should first try to find out which plaintext is the most frequently used in the target application, collect the data for various keys, and then perform the actual attack. The results summarized in Table 2 compare favorably with the best attacks on such ciphers as DES, Triple-DES, Skipjack, and AES. Moreover, the scenario of TMD attacks is much more practical than that of related-key attacks. We provide several trade-off points for AES-128 in Table 3.

3 GPU-Like AES Processor for Cryptanalysis

In this section, we introduce the architecture and functionality of a high-speed AES processor optimized for cryptanalytic attacks following the TMK trade-off and the RKC method as discussed above. From an architectural point of view, this AES processor is basically a homogenous multi-core system with local cache and shared memory, similar to present-day Graphics Processing Units (GPUs) such as NVIDIA's GT200b [33]. It consists of a large number of programmable high-speed AES engines that can work in parallel, controlled by a small number of general-purpose processing units. The AES engines are optimized for the requirements and characteristics of the cryptanalytic attacks described in Section 2.

The AES engines of our processor differ greatly from a "conventional" high-speed AES hardware implementation such as the one that Intel announced to integrate into the Westmere micro-architecture and its successors. For example, in our processor the plaintexts to be encrypted do not need to be loaded from "outside", but are either constant (in case of the RKC attack) or can be easily generated on chip (in case of the TMK trade-off). The same holds for the keys; they are either constant over a large number of encryptions or can be generated on-chip. Another major difference between our AES engine and general-purpose AES hardware is that we do not need to support a mode of operation, which allows for pipelining the datapath.

There exists a rich literature on high-speed AES hardware architectures, targeting both FPGA and standard-cell implementations. Hodjat and Verbauwhede present in [23, 24] the design of an AES datapath that fulfills most of the requirements for use in cryptanalysis mentioned above. Their datapath has a width of 128 bits and implements both inner-round and outer-round pipelining, which means that a new plaintext can be fed into the circuit every clock cycle. Every round is performed in four cycles and the plaintext has to pass through a total of 10 rounds, which results in a latency of 41 clock cycles altogether (including one cycle for the initial key addition). Hodjat and Verbauwhede also report implementation results based on a 0.18 μm standard cell library. Due to the massive pipelining, the AES datapath can be clocked with a relatively high frequency of 606 MHz, yielding a throughput of 77.6 Gbit/s. The overall silicon area is about 473k gates for a 10-round implementation; the area of a 14-round datapath (for keys up to 256 bits) can be estimated to be roughly 660k gates. However, the throughput is independent of the length of the datapath (and also of the key size) since the plaintexts are always processed at a rate of one 128-bit block per cycle.

As mentioned before, the proposed multi-core processor for cryptanalysis of the AES follows the architectural model of modern GPUs such as the NVIDIA GT200b[2]. The GT220b architecture is based on a scalable processor array and consists of 240 streaming-processor cores (so-called "shader" cores), each of which can execute up to three floating-point operations per cycle (e.g. two multiplications and one addition). Assuming that the shader cores are clocked with a frequency of 1476 MHz (e.g. GeForce GTX 285 video card [33]), the theoretical performance of the GT200b exceeds 1000 single-precision GFLOPS. For comparison, a high-end CPU such as the Intel Core-i7 reaches just slightly more than 100 GFLOPS when clocked at its maximum frequency of 3.33 GHz, i.e. the performance gap between current-generation CPUs and GPUs is about an order of magnitude. The GT200b consists of 1.4 billion transistors (i.e. 350M gates) covering a 470 mm^2 die area built on a 55 nm CMOS process [33].

Our AES processor is a multi-core system consisting of 500 AES engines based on Hodjat's design as sketched above. Each AES engine has an area of 660k gates, which amounts to a total of 330M gates for 500 engines. When including other building blocks (e.g. host interface, small local memory, interface to external memory, etc.), it can be expected that the overall silicon area of our AES processor will be roughly comparable to that of the GT200b. However, we assume the AES engines to be clocked with a frequency of 2.0 GHz, which should be easily possible when considering that Hodjat's implementation reached a frequency of 606 MHz on basis of an old 0.18 μm process that is significantly slower than the recent 55 nm TSMC technology. Of course, cranking up the clock speed will also increase power consumption, but the additional heat can be controlled by better cooling, as will be discussed in more detail in Section 5. Each AES engine can encrypt plaintexts at a rate of one 128-bit block per cycle, yielding an overall throughput of $500 \times 2 \cdot 10^9 = 10^{12}$ AES operations per second. Interestingly, these 10^{12} AES operations per second match exactly the 1000 GFLOPS per second of the GT200b.

[2]Recently, NVIDIA introduced a new generation of GPUs based on the Fermi architecture, which consists of 3.0 billion transistors and is manufactured in a 40 nm process. Even though Fermi-based GPUs, such as the GF110, are superior to the GT200b in both performance and energy efficiency, we decided to stick with the GT200b as reference since it has been widely used for the implementation of cryptographic software and is, therefore, well known in the research community.

4 Memory Throughput and Storage Requirements

Looking at the raw complexity figures of the RKC and TMK attack, it is obvious that memory capacity (and throughput) is the most critical issue after computation time. In fact, the main bottleneck of many high-performance applications is memory bandwidth, i.e. the speed with which data can be transferred between memory (i.e. RAM) and the functional units of the processor where the actual computation is performed. However, in analogy to our argumentation from previous section, we have to point out first that both cryptanalytic attacks considered in this paper have very special requirements with respect to memory and storage, which differ greatly from the requirements of "general-purpose" applications. This difference is especially pronounced with respect to key agility. While support of key agility is important for many "conventional" hardware implementations, it is not an issue for the RKC attack since there are only four keys in total. In case of the TMK attack, key-agility is important, but the keys can be generated on chip. The plaintext is kept fixed during the whole attack.

Given the high performance of our processor (500 AES operations per clock cycle), one may expect that memory bandwidth is a limiting factor since plaintexts can hardly be loaded at a rate of $500 \times 16 = 8000$ bytes per cycle. Fortunately, the plaintexts processed in both the RKC and TMK attack do not need to be loaded from off-chip memory, but can be generated on-chip. In the former case (RKC attack), the plaintexts can be generated in a very straightforward way using a plain counter. The TMK attack, on the other hand, executes encryption chains in the pre-computation phase, i.e. the output of an AES operation is input to the next one, whereby a simple modification of the output (e.g. a bit permutation) is carried out in between. However, this modification can be easily implemented in hardware and does not impact the throughput of the AES processor. The situation is similar for the ciphertexts. When performing an RKC attack, only very few ciphertexts are actually stored, i.e. the throughput with which data is moved to off-chip storage is several orders of magnitude lower than the AES processing rate.

Even though only a small fraction of the ciphertexts are actually stored, the storage requirements of the RKC attack are still enormous, namely 2^{78} bytes. This amount is needed for storage of four 2^{72} structures of 16-bytes and is unavoidable unless a better differential is found. The attack also needs an array of 2^{78} counters in the final stage, but this part can be optimized to consume less memory. Storage of such size can only be realized in a distributed fashion, e.g. by attaching a high-capacity harddisk to each AES processor. The CAESAR supercomputer described in the next section consists of $3 \cdot 10^{10}$ AES processors (and therefore we also have $3 \cdot 10^{10}$ harddisks), which means each harddisk must provide a capacity of slightly less than 10 TB. However, the state-of-the-art (as of 2011) are harddisks with a capacity of 2.5 TB, costing 100 US$ when purchased in large quantities. Nonetheless, we argue that the enormous storage requirements do not render the RKC attack infeasible, at least not when taking into account recent technical innovations. For example, researchers at the A*STAR Institute of Materials Research and Engineering have been able to fabricate magnetic storage media with a density of 3.3 Tb per inch^2, thereby improving the state-of-the-art by a factor of six [1]. Hitachi is striving to make a technology called Two-Dimensional Magnetic Recording (TDMR) ready for mass production, which could boost storage density to up to 10 Tb per inch^2 [45]. Therefore, it is not unrealistic to assume that 100 TB harddisks can be mass produced for just 100 US$ within the next 5–10 years. Such a 100 TB harddisk could be shared by 10 AES processors. Consequently, the overall cost of storage for the RKC attack (i.e. $3 \cdot 10^9$ harddisks of 100 TB each) would amount to roughly 300 billion US$.

The storage requirements of the TMK attack are $2^{56} \cdot 16 \cdot 2 = 2^{61}$ bytes, which will cost in 5–10 years only 2.3 million US$ (or 92 million US$ with present-day technology and prices). As mentioned in Section 2, only the start and endpoints of the encryption chains generated in the pre-computation phase of the TMK attack are actually stored. The situation is similar for an RKC attack since only a very small fraction of the ciphertexts actually needs to be stored. In order to simplify the estimation of the time

required for an RKC or TMK attack, we assume that storing data on the harddisks, as well as any subsequent operation accessing the stored data, does not slow down the attack (i.e. the overall attack time is primarily determined by the AES operations and not by accesses to memory or storage). This assumption is justified in the context of the present paper for two reasons. First, the rate at which data is transferred to and from storage is a factor of (at least) 2^{21} lower than the AES processing rate. Second, as stated in Section 1, we aim to estimate a *lower bound* for the time and energy required to perform an attack.

5 Evaluation of Attack Time and Energy

To assess the feasibility of the RKC and TMD attack using "GPU-like" special-purpose hardware, we make the following assumptions about the adversary. We assume an extremely powerful adversary with huge resources in terms of both financial means and expertise in cryptanalysis, which can be expected to be the case for the national security agencies and/or defense departments of certain countries. More precisely, we assume that the adversary has a budget of 1 trillion (i.e. 10^{12}) US$ at its disposal for the design and manufacturing special-purpose hardware (i.e. GPU-like processors). Based on these highly optimized processors, the adversary can build a large-scale supercomputer for cryptanalytic attacks on the AES; we call this supercomputer *CAESAR* (short for Cryptanalytic AES ARchitecture). A budget of 1 trillion US$ is not completely unrealistic when considering that the overall amount the US spends for military and national defense is estimated to be between 880 billion and 1.03 trillion US$ in fiscal year 2010 [42].

We furthermore assume that the adversary has additional funds to cover other expenses such as designing the AES processor, designing and implementing dedicated storage for the TMD attack, operating the CAESAR supercomputer for a certain period of time (which is primarily energy costs), and so on. The exact amount of money needed for these additional expenses depends on many different factors (e.g. the resources of the adversary). For example, the adversary could be an organization that possesses its own power plants, which significantly reduces the cost for operating large server farms to house the CAESAR supercomputer. In any case, it can be expected that these additional costs will be considerably below the 1 trillion US$ we assume for the manufacturing of AES processors; a reasonable estimation is 500 billion US$. Again, a total funds of 1.5 trillion US$ is not completely unrealistic when taking the annual budget deficit of the US as reference, which is expected to exceed 1.4 trillion US$ in the fiscal year 2009 [2].

The next question to answer is how many AES processors can be produced for 1 trillion US$. We take again the NVIDIA GT200b as reference since our optimized processor housing 500 AES engines has roughly the same silicon area as the GT200b, which means that the manufacturing costs should be very similar. Unfortunately, we were not able to find a reliable source for the manufacturing cost of a GT200b processor. However, what is publicly known are the retail prices of complete graphics/video cards containing the GT200b. For example, the GeForce GTX 285 [33], a graphics card equipped with a GT200b processor clocked at 1476 MHz, retails for less than 300 US$. The GeForce GTX 295 [34] is a graphics card housing two GT200b processors that costs less than 400 US$. However, it must be considered that both are complete graphics cards that do not only contain GT200b chips, but also large amounts of fast memory and several other components. Furthermore, we have to take into account that the quoted retail prices include gains for the producer and retailer(s), NRE costs, as well as other costs such as VAT. Therefore, it can be assumed that manufacturing a GT200b chip costs significantly less than 100 US$. The cost of one of our AES processors will be even much lower since, for example, the NRE costs are negligible when producing a very large number of chips. Taking all this into account, we can estimate a lower bound of 30 US$ for the manufacturing cost of a single AES processor.

Having a budget of 1 trillion (i.e. 10^{12}) US$ for chip production (and assuming a reasonably high yield) means that the adversary gets a total of about $3 \cdot 10^{10}$ AES processors, each of which can perform 10^{12} AES operations per second (see Section 3). Consequently, the overall throughput of all processors of CAESAR amounts to roughly $3 \cdot 10^{22}$ AES operations per second. The RKC attack as described in Section 2 requires the adversary to perform $2^{99.5} \approx 9 \cdot 10^{29}$ AES operations, which can be accomplished in just $3 \cdot 10^7$ seconds (i.e. approximately one year) on the CAESAR supercomputer. Of course, these estimations are based on "best-case" (yet not unreasonable) assumptions and should be considered as a lower bound for the execution time of this key-recovery attack given a budget of 1 trillion US$ for chip production.

The situation is similar for the TMK trade-off attack in the sense that the AES operations dominate the overall execution time by far. However, TMK attacks, as mentioned in Section 2, are performed in two phases; an off-line (i.e. pre-computation) phase in which tables containing the start and endpoints of encryption chains are generated, and an online phase in which pre-images of data points are searched in the tables. Let us consider an example of a TMK attack in which the adversary is given 2^{32} encryptions of a fixed plaintext under different (but unknown) keys. To recover one of these keys, the adversary has to carry out 2^{96} AES operations during the pre-computation phase, as well as 2^{80} AES operations in the online phase. Similar to the RKC attack, the inputs for the AES operations carried out in the pre-computation phase do not need to be loaded from an external source, but can be generated on-chip[3]. Only the start and end-point of the encryption chain are actually stored in the table, which means that the memory bandwidth can be orders of magnitude lower than the AES throughput. In our case, the pre-computed tables contain 2^{56} data points (i.e. 128-bit ciphertexts) altogether. If we assume again $3 \cdot 10^{10}$ GPU-like AES processors with an overall throughput of $3 \cdot 10^{22}$ AES operations per second, the 2^{96} AES operations carried out in the off-line phase take about 30.6 days. The 2^{80} AES operations of the on-line phase are negligible in relation to the execution time of the off-line phase, which means the overall attack time is primarily determined by the pre-computation of the tables. However, this pre-computation is a one-time effort because the same set of tables can be used to recover other keys. If one is willing to wait for one year until the pre-computation is finished, then he needs less than one tenth of the attack budget. On such a "smaller" supercomputer, solutions will still be generated at an amazing speed of eight minutes per 128-bit key.

5.1 Further Considerations

Besides execution time and memory requirements, there are a number of other factors that need to be taken into account when studying the feasibility of a large-scale supercomputer for cryptanalysis of the AES like CAESAR. In the following, we try to estimate the time it takes to manufacture $3 \cdot 10^{10}$ AES processors and the energy these processors consume when clocked with a frequency of 2.0 GHz.

A state-of-the-art fab for chip production, such as the one mentioned in [36], has an overall capacity of 300,000 wafers per month. Given a diameter of 300 mm, the silicon area of a single wafer amounts to 70,685 mm^2. In Section 3, we argued that a GPU-like processor housing 500 AES engines would have roughly the same gate count as the NVIDIA GT200b, hence it is sensible to assume that its silicon area will be in the same range, namely 470 mm^2 on basis of the 55 nm TSMC technology. Consequently, 150 AES processors can theoretically be obtained from a 300 mm wafer. However, given a typical yield of 75% and taking edge dies into account, it can be estimated that we get out some 100 AES processors per wafer. A high-capacity fab would be able to produce $3 \cdot 10^7$ chips in one month, or $3.6 \cdot 10^8$ chips

[3] More precisely, the input (i.e. plaintext) of a given AES encryption is always fixed and the output (i.e. ciphertext) of the previous AES encryption is used as a new key, after a simple modification (e.g. a fixed bit permutation). This simple modification of the output bits can be easily implemented in hardware and does not impact the throughput of the AES processor.

per year. Consequently, the total production time for $3 \cdot 10^{10}$ AES processors amounts to approximately 83 years. Accordingly, the time would drop to one year when the chip production is distributed to 83 high-capacity fabs. Note that the fab mentioned in [36] was constructed in 18 months and required an investment of 1 billion US$. A well-funded adversary (as assumed in this paper) may even consider to construct its own high-capacity fabs and operate these fabs solely for the production of GPU-like AES processors.

NVIDIA's GeForce GTX285, a graphics card equipped with a GT200b processor, has a maximum power consumption of 204 W [33]. In this card, each of the 240 shader cores of the GT200b is clocked with a frequency of 1476 MHz. On the other hand, the GeForce GTX295 houses two GT200b, but their shaders are clocked with a slightly lower frequency of 1242 MHz. Its maximum power consumption is 289 W as specified in [34]. However, it must be considered that these figures refer to the power consumption of the "whole" graphics card, which includes besides the GT200b processor(s) also several other components, in particular large amounts of memory. Therefore, it can be estimated that a GT200b processor clocked at 1476 MHz consumes approximately 100 W. Our AES processor is operated at a slightly higher frequency (2.0 GHz[4] instead of 1476 MHz) and, as a consequence, its power consumption will rise by the same factor to 135 W. The power consumed by a mechanical harddisk depends, among other things, on its spin speed and operation mode (e.g. read/write, idle, standby). A current-generation 7200 rpm harddisk, such as one of the Western Digital Caviar Black series [41], has an average power dissipation of 10.7 W when reading or writing, 8.2 W when idle, and 1.3 W when in standby or sleep mode. In contrast, so-called "green" harddisks with a spin speed of up to 5400 rpm (e.g. Western Digital Caviar Green series) have power consumption figures of 6 W (read/write mode), 5.5 W (idle mode), and 0.8 W (standby mode). Consequently, a state-of-the-art harddisk dissipates less than one tenth of the power of one of our AES processor. The power consumed by all $3 \cdot 10^{10}$ AES processors amounts to a whopping 4 TW, i.e. $4 \cdot 10^{12}$ W. For comparison, the average total power consumption of the US was 3.34 TW in 2005 [43]. In summary, it can be concluded that the most limiting factor of attacking AES using special-purpose hardware is neither the computation time nor the memory requirements, but the power consumption of the hardware.

Does this enormous power consumption of our CAESAR supercomputer render RKC (respectively TMK) attacks with a complexity of $2^{99.5}$ (respectively 2^{96}) completely impossible? Not necessarily when we consider Moore's law: transistor sizes (and also power consumption) shrunk significantly with every new process generation that was introduced during the past two decades. It is expected that Moore's law will continue to hold—and transistor sizes will continue to shrink—for another ten years, though at a slightly lower rate than in the past [28]. For example, TSMC estimates a transistor size of only 7 nm in 2020, which is eight times smaller than the transistor size of the 55 nm TSMC technology under which the GT200b processor is produced. Using a 7 nm technology for our AES processor would result in a power consumption that is only a fraction of the 135 W we used for the evaluation above. Estimations beyond the ten-year horizon are rather difficult since future VLSI technology must not necessarily be silicon-based. However, the following historical example may help to understand the progress in VLSI technology. The first supercomputer that reached a performance of 1 TFLOPS (i.e. 1000 GFLOPS) was the ASCI Red, built in 1997 by Intel and operated by Sandia National Labs. ASCI Red housed almost 10,000 Pentium Pro processors and had a power consumption of roughly 500 kW. Today, a single GPU like the GT200b reaches the same performance, but does so at a power consumption of only 100 W (see above). Consequently, the power consumption per TFLOPS dropped from 500 kW in 1997 to 100 W in 2010, which corresponds to a factor of 5,000.

[4] The increased amount of heat due to the higher clock frequency can be handled through better cooling, e.g. by adding a liquid cooling system. Liquid cooling, even with "warm" water, can be 4000 times more effective than air cooling [37].

5.2 Outlook into the Future

In this subsection, we briefly mention some factors that can significantly decrease the cost of hardware attacks on AES in the future (10–20 years from now). These factors are:

- Moore's law continuing to hold for another ten years, albeit at a slightly reduced speed.
- Cryptanalytic breakthroughs can entail spectacular reductions in attack complexity. However, the cryptanalytic progress for AES does not follow a steady and predictable flow. It is hard to make any predictions based on the time-line of the past attacks since they were very sporadic.
- Computers based on spin (so-called magneto-electronics or spintronics) may significantly reduce power consumption.
- The use of optical computers may also significantly reduce the power consumption of large-scale cryptanalytic hardware.
- Progress in miniaturization may lead to magnetic storage cells of the size of a few atoms. The currently smallest memory-storage element, developed at IBM Research, consists of only 12 iron atoms.
- 3D optical data storage is one of the technologies that could increase storage density and hence decrease the memory cost of attacks. For example, optical disks of 1 PB (i.e. 10^{15} bytes) having the size of a DVD (and a similar price) are conceivable. Such disks will use more than 100 optical layers.
- Electronic quantum holography: Superimposing images of different wavelengths into the same hologram on copper medium can increase memory density spectacularly. For example, a density of 35 bits per electron (which is way below the supposed limit of one atom representing one bit) was demonstrated in 2009 using this technique[5].

6 Conclusions

In this paper, we investigated the feasibility of large-scale hardware attacks on AES-128 and AES-256 bounded by a time complexity of 2^{100}. We described CAESAR, a hypothetical supercomputer consisting of $3 \cdot 10^{10}$ GPU-like AES processors, each of which can reach a throughput of 10^{12} AES operations per second. CAESAR could be built with a total budget of roughly 1.5 trillion US$ (or with 1 trillion US$ solely spent for chip fabrication) and would be capable of performing up to $3 \cdot 10^{22}$ AES operations per second, or approximately $9 \cdot 10^{29} \approx 2^{99.5}$ AES operations in a year. Table 4 summarizes the main characteristics and capabilities of CAESAR along with the complexities of the TMK and RKC attack on AES-128 and AES-256, respectively. Our evaluation shows that a TMK trade-off attack on AES-128 using 2^{32} targets is well within reach with current VLSI technology. CAESAR requires about 30 days for the pre-computation phase, after which each new 128-bit key out of the pool of 2^{32} targets can be found in negligible time. A smaller variant of CAESAR costing 100 billion US$ is able to break a new key every eight minutes, but needs a year to pre-compute the tables. We also studied the RKC attack on AES-256 and identified the huge memory complexity (and resulting storage cost) of 2^{78} as a limiting

[5] Overall, in the field of digital storage there seem to be several competing technologies, which are of very different physical nature and in which progress happens in sudden leaps, rather than a monotone growth. There is also a negative effect of well-developed technologies that come close to their physical limits, but still act as a barrier to the development of new revolutionary ideas due to high initial costs.

Table 4: Main characteristics of CAESAR and summary attack complexities.

One AES engine:	660K gates	2GHz clock speed	
One AES processor:	500 AES engines[a]	10^{12} AES ops/ s	30 US$[b]
CAESAR supercomputer:	$3 \cdot 10^{10}$ AES processors	$3 \cdot 10^{22}$ AES ops/s	1 trillion US$
AES chip production:	83 high capacity fabs	approx. 1 year	83 bln US$
Power consumption:	135 W per processor	4 TW = $4 \cdot 10^{12}$ W	
RKC Attack:	$9 \cdot 10^{29}$ AES ops	approx. 1 year	
Storage RKC:	2^{78} bytes		300 bln US$[c]
TMK Attack (2^{32} targets):	$0.8 \cdot 10^{29}$ pre-computation	30.6 days	
TMK Attack (2^{32} targets):	10^{24} ops per AES key	negligible	
Storage TMK:	2^{61} bytes		92 mln US$

[a] A 470 mm^2 die on 55nm TSMC CMOS process with 330M gates.
[b] This is a lower bound.
[c] Estimate for the next 5–10 years.

factor, even though CAESAR could perform the required $2^{99.5}$ AES operations in roughly one year. In summary, our work shows that the main bottlenecks of large-scale cryptanalytic hardware for breaking the AES are neither execution time nor production cost, but rather power consumption and high memory complexity. Therefore, we recommend cryptanalysts to focus on attacks with a time complexity of up to 2^{100} and a memory and data complexity of less than 2^{70}.

References

[1] ARS Technica. Researchers add a dash of salt to hard drives for capacities up to 18TB. Available online at `http://arstechnica.com/gadgets/news/2011/10/researchers-increase-hard-drive-density-sixfold-with-salt.ars`, 2011.

[2] BBC News. US deficit hits record $1.4tn. Available online at `http://news.bbc.co.uk/2/hi/8296079.stm`, 2009.

[3] D. J. Bernstein, H.-C. Chen, M.-S. Chen, C.-M. Cheng, C.-H. Hsiao, T. Lange, Z.-C. Lin, and B.-Y. Yang. The billion-mulmod-per-second PC. In *SHARCS '09: Special-Purpose Hardware for Attacking Cryptographic Systems*, pp. 131–144, Lausanne, Switzerland, Sept. 2009.

[4] D. J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang. ECM on graphics cards. In *Advances in Cryptology — EUROCRYPT 2009*, vol. 5479 of *Lecture Notes in Computer Science*, pp. 483–501. Springer Verlag, 2009.

[5] E. Biham, O. Dunkelman, and N. Keller. Related-key boomerang and rectangle attacks. In *Advances in Cryptology — EUROCRYPT 2005*, vol. 3494 of *Lecture Notes in Computer Science*, pp. 507–525. Springer Verlag, 2005.

[6] A. Biryukov. The boomerang attack on 5 and 6-round reduced AES. In *Advanced Encryption Standard — AES 2004*, vol. 3373 of *Lecture Notes in Computer Science*, pp. 11–15. Springer Verlag, 2005.

[7] A. Biryukov and D. Khovratovich. Related-key cryptanalysis of the full AES-192 and AES-256. In *Advances in Cryptology — ASIACRYPT 2009*, vol. 5912 of *Lecture Notes in Computer Science*, pp. 1–18. Springer Verlag, 2009.

[8] A. Biryukov, D. Khovratovich, and I. Nikolic. Distinguisher and related-key attack on the full AES-256. In *Advances in Cryptology — CRYPTO 2009*, vol. 5677 of *Lecture Notes in Computer Science*, pp. 231–249. Springer Verlag, 2009.

[9] A. Biryukov, S. Mukhopadhyay, and P. Sarkar. Improved time-memory trade-offs with multiple data. In *Selected Areas in Cryptography — SAC 2005*, vol. 3897 of *Lecture Notes in Computer Science*, pp. 110–127. Springer Verlag, 2006.

[10] A. Biryukov and I. Nikolic. Automatic search for related-key differential characteristics in byte-oriented block ciphers: Application to AES, Camellia, Khazad and others. In *Advances in Cryptology — EUROCRYPT 2010*, vol. 6110 of *Lecture Notes in Computer Science*, pp. 322–344. Springer Verlag, 2010.

[11] A. Biryukov and A. Shamir. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In *Advances in Cryptology — ASIACRYPT 2000*, vol. 1976 of *Lecture Notes in Computer Science*, pp. 1–13. Springer Verlag, 2000.

[12] J. W. Bos, T. Kleinjung, R. Niederhagen, and P. Schwabe. ECC2K-130 on Cell CPUs. In *Progress in Cryptology — AFRICACRYPT 2010*, vol. 6055 of *Lecture Notes in Computer Science*, pp. 225–242. Springer Verlag, 2010.

[13] J. Daemen and V. Rijmen. *The Design of Rijndael: AES – The The Advanced Encryption Standard*, vol. ?? of *Information Security and Cryptography*. Springer Verlag, 2002.

[14] Electronic Frontier Foundation. *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design*. O'Reilly Media, 1998.

[15] N. Ferguson, J. Kelsey, S. Lucks, B. Schneier, M. Stay, D. Wagner, and D. Whiting. Improved cryptanalysis of Rijndael. In *Fast Software Encryption — FSE 2000*, vol. 1978 of *Lecture Notes in Computer Science*, pp. 213–230. Springer Verlag, 2001.

[16] W. Geiselmann, F. Januszewski, H. Köpfer, J. Pelzl, and R. Steinwandt. A simpler sieving device: Combining ECM and TWIRL. In *Information Security and Cryptology — ICISC 2006*, vol. 4296 of *Lecture Notes in Computer Science*, pp. 118–135. Springer Verlag, 2007.

[17] W. Geiselmann and R. Steinwandt. Non-wafer-scale sieving hardware for the NFS: Another attempt to cope with 1024-bit. In *Advances in Cryptology — EUROCRYPT 2007*, vol. 4515 of *Lecture Notes in Computer Science*, pp. 466–481. Springer Verlag, 2007.

[18] H. Gilbert and M. Minier. A collision attack on 7 rounds of Rijndael. In *Proceedings of the 3rd Advanced Encryption Standard Candidate Conference*, pp. 230–241. National Institute of Standards and Technology, 2000.

[19] M. Gorski and S. Lucks. New related-key boomerang attacks on AES. In *Progress in Cryptology — INDOCRYPT 2008*, vol. 5365 of *Lecture Notes in Computer Science*, pp. 266–278. Springer Verlag, 2008.

[20] R. E. Graves. *High Performance Password Cracking by Implementing Rainbow Tables on NVIDIA Graphics Cards (IseCrack)*. M.Sc. Thesis, Iowa State University, Ames, IA, USA, 2008.

[21] T. Güneysu, T. Kasper, M. Novotný, C. Paar, and A. Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers*, 57(11):1498–1513, Nov. 2008.

[22] M. E. Hellman. A cryptanalytic time-memory tradeoff. *IEEE Transactions on Information Theory*, 26(4):401–406, July 1980.

[23] A. Hodjat and I. Verbauwhede. Speed-area trade-off for 10 to 100 Gbits/s throughput AES processor. In *Proceedings of the 37th Asilomar Conference on Signals, Systems, and Computers (ACSSC 2003)*, vol. 2, pp. 2147–2150. IEEE, Nov. 2003.

[24] A. Hodjat and I. Verbauwhede. Area-throughput trade-offs for fully pipelined 30 to 70 Gbits/s AES processors. *IEEE Transactions on Computers*, 55(4):366–372, Apr. 2006.

[25] D. Kahn. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Scribner, rev. sub. edition, 1996.

[26] J. Kim, S. Hong, and B. Preneel. Related-key rectangle attacks on reduced AES-192 and AES-256. In *Fast Software Encryption — FSE 2007*, vol. 4593 of *Lecture Notes in Computer Science*, pp. 225–241. Springer Verlag, 2007.

[27] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler. Breaking ciphers with COPACOBANA – A cost-optimized parallel code breaker. In *Cryptographic Hardware and Embedded Systems — CHES 2006*, vol. 4249 of *Lecture Notes in Computer Science*, pp. 101–118. Springer Verlag, 2006.

[28] R. Kwong. TSMC warns Moore's law may have 10 years left. Fiancial Times Tech Blog, available online at `http://blogs.ft.com/techblog/2010/04/tsmc-warns-moores-law-may-have-10-years-left`, Apr. 2010.

[29] J. Lu, O. Dunkelman, N. Keller, and J. Kim. New impossible differential attacks on AES. In *Progress in Cryptology — INDOCRYPT 2008*, vol. 5365 of *Lecture Notes in Computer Science*, pp. 279–293. Springer Verlag, 2008.

[30] N. Mentens, L. Batina, B. Preneel, and I. M. Verbauwhede. Time-memory trade-off attack on FPGA platforms: UNIX password cracking. In *Reconfigurable Computing: Architectures and Applications — ARC 2006*, vol. 3985 of *Lecture Notes in Computer Science*, pp. 323–334. Springer Verlag, 2006.

[31] G. Meurice de Dormale, P. Bulens, and J.-J. Quisquater. Collision search for elliptic curve discrete logarithm over GF(2^m) with FPGA. In *Cryptographic Hardware and Embedded Systems — CHES 2007*, vol. 4727 of *Lecture Notes in Computer Science*, pp. 378–393. Springer Verlag, 2007.

[32] K. Nohl, E. Tews, and R.-P. Weinmann. Cryptanalysis of the DECT standard cipher. In *Fast Software Encryption — FSE 2010*, vol. 6147 of *Lecture Notes in Computer Science*, pp. 1–18. Springer Verlag, 2010.

[33] NVIDIA Corporation. GeForce GTX 285: A Powerful Single GPU for Gaming and Beyond. Specification, available online at `http://www.nvidia.com/object/product_geforce_gtx_285_us.html`, 2010.

[34] NVIDIA Corporation. GeForce GTX 295: A Powerful Dual Chip Graphics Card for Gaming and Beyond. Specification, available online at `http://www.nvidia.com/object/product_geforce_gtx_295_us.html`, 2010.

[35] C. B. Pomerance, J. W. Smith, and R. S. Tuler. A pipeline architecture for factoring large integers with the quadratic sieve algorithm. *SIAM Journal on Computing*, 17(2):387–403, Apr. 1988.

[36] Samsung Electronics Co. Ltd. Samsung and Siltronic start joint production of 300mm wafers in Singapore. Press release, available online at `http://www.samsung.com/us/aboutsamsung/news/newsIrRead.do?news_ctgry=irnewsrelease&news_seq=9345`, 2008.

[37] Scientific American. Warm water flowed through supercomputers to cool down their heat. Available online at `http://www.scientificamerican.com/article.cfm?id=microchannel-warm-liquid-cooling`, 2010.

[38] A. Shamir. Factoring large numbers with the TWINKLE device (Extended abstract). In *Cryptographic Hardware and Embedded Systems — CHES '99*, vol. 1717 of *Lecture Notes in Computer Science*, pp. 2–12. Springer Verlag, 1999.

[39] A. Shamir and E. Tromer. Factoring large numbers with the TWIRL device. In *Advances in Cryptology — CRYPTO 2003*, vol. 2729 of *Lecture Notes in Computer Science*, pp. 1–26. Springer Verlag, 2003.

[40] S. Singh. *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography*. Anchor Books, 2000.

[41] Western Digital Corporation. WD Caviar Black Desktop Hard Drives. Specification sheet, available for download at `http://www.wdc.com/wdproducts/library/SpecSheet/ENG/2879-701276.pdf`, 2011.

[42] Wikipedia. Military Budget of the United States. Available online at `http://en.wikipedia.org/wiki/Military_budget_of_the_United_States`, 2010.

[43] Wikipedia. Orders of Magnitude (Power). Available online at `http://en.wikipedia.org/wiki/Orders_of_magnitude_(power)`, 2010.

[44] J. E. Wilcox. *Solving the Enigma: History of the Cryptanalytic Bombe*. Center for Cryptologic History, National Security Agency, 2001.

[45] R. Wood, M. Williams, A. Kavcic, and J. Miles. The feasibility of magnetic recording at 10 Terabits per square inch on conventional media. *IEEE Transactions on Magnetics*, 45(2):917–923, Feb. 2009.

Better than Brute-Force — Optimized Hardware Architecture for Efficient Biclique Attacks on AES-128

Andrey Bogdanov, Elif Bilge Kavun, Christof Paar, Christian Rechberger, and Tolga Yalcin

K.U. Leuven, Belgium; RUB, Germany; DTU, Denmark

Abstract. Biclique cryptanalysis was recently used to claim the first attack on AES-128 without related keys. However, compared to a simple brute force attack, the method is more complicated, and its data requirements are much higher (2^{88} chosen ciphertexts). Still, the theoretical speed gain over brute force search was small.
We show in this paper for the first time a practical implementation of a biclique attack on AES, using a parallel FPGA machine and ASICs, with a practical data complexity of only 16 chosen plaintexts, and still gaining almost a factor 2 over optimized brute force search.

1 Introduction

After 15 years of intensive cryptanalytic efforts, the full version of Advanced Encryption Standard (AES) has recently been claimed vulnerable to a new class of cryptanalytic techniques based on so called *bicliques*. Starting as an extension to meet-in-the-middle attacks on hash functions (initial structures) [6, 11], bicliques have demonstrated their potential in attacks on the full AES [10], full IDEA [8], and several other designs. As presented in [2, 5], the theoretical complexity of bicliques attacks is lower than that of brute force attacks for the ciphers considered. However, it is an open research problem whether such a reduction in complexity can be achieved in an actual implementation.

The paper at hand answers this question in the affirmative. As a test case, we considered AES, which is arguably the most widely used cipher in the world. In this paper, we demonstrate that with a carefully optimized hardware architecture it is possible to accelerate an AES brute-force attack by a factor of almost 2. Given the complexity of the attack *relative to the extremely simply structure of a brute-force attack*, our finding is somewhat surprising. Interestingly, the gain of a factor of approximately 2 is close to the predicted theoretical improvement which can be achieved through biclique attacks. Even though we only considered AES for our architecture, it seems highly likely that a similar approach can be used for other ciphers which can be attacked with bicliques [1, 9, 5]. We used the RIVYERA parallel FPGA machine [13], a successor of the COPACOBANA special-purposed machine [7], as implementation platform. RIVYERA is among the most powerful hardware-based cryptanalytical machine available (outside government agencies, that is), and provides thus absolute numbers for the attack which are of high practical relevance. In order to obtain fair results, we first designed and implemented an architecture for a highly optimized brute-force attack against AES. The key subspace to be searched can be defined by the user. Using this as a starting point, we implemented a biclique attack which improved the complexity.

Our paper demonstrates that the structural weakness on which the biclique attack is based can in fact be exploited for practical attacks. As a consequence, applications which do not make use of the full key space of ciphers such as AES (e.g., because there is not enough entropy in the system for key generation) have to be re-calibrated with respect to the actual security they provide. Methods such as hashing a low-entropy string, e.g., one derived from a user-entered password, will render our bicliques architecture very difficult. However, it seems very likely that there are many applications, e.g., in the embedded domain, which simply use n bits, $n < 128$, as direct input to AES. For such applications our hardware-based attack can constitute a threat.

2 Low data complexity biclique cryptanalysis of AES-128

In here, we describe our new low data complexity key recovery for AES-128. It requires only 16 chosen plaintexts, works with computational complexity $2^{126.89}$ and has success probability 1.

Our key recovery uses the ideas of bicliques, though we do not introduce the theoretical concept of bicliques formally here and choose to describe the flow of the key recovery in a more concrete way to favour the simplicity and intelligibility of description. In this way, we also aim to facilitate the understanding of our subsequent implementation in hardware. For background on biclique cryptanalysis of block ciphers, the reader is referred to [2].

2.1 AES-128

For the sake of clarity, we will be trying to reuse as much notation as possible from [2]. This also applies to the notations regarding the description of AES-128.

AES-128 is a block cipher with 128-bit internal state and 128-bit key K. The internal state is represented by a 4×4 byte matrix, and the key is represented by a 4×4 matrix. The plaintext is xored with the key, and then undergoes a sequence of 10 rounds. Each round consists of four transformations: nonlinear bytewise SubBytes, the byte permutation ShiftRows, linear transformation MixColumns, and the addition with a subkey AddRoundKey. MixColumns is omitted in the last round.

SubBytes is a nonlinear transformation operating on 8-bit S-boxes. The ShiftRows rotates bytes in row r by r positions to the left. The MixColumns is a linear transformation with branch number 5, i.e. in the column equation $(u_0, u_1, u_2, u_3) = MC(v_0, v_1, v_2, v_3)$ only 5 and more variables can be non-zero.

We address two internal states in each round as follows: #1 is the state before SubBytes in round 1, #2 is the state after MixColumns in round 1, #3 is the state before SubBytes in round 2, ..., #19 is the state before SubBytes in round 10, #20 is the state after ShiftRows in round 10 (MixColumns is omitted in the last round).

The key K is expanded to a sequence of keys $K^0, K^1, K^2, \ldots, K^{10}$, which form a 4×60 byte array. Then the 128-bit subkeys $\$0, \$1, \$2, \ldots, \$14$ come out of the sliding window with a 4-column step. The keys in the expanded key are formed as follows. First, $K^0 = K$. Then, column 0 of K^r is the column 0 of K^{r-1} xored with the nonlinear function (SK) of the last column of K^{r-1}. Subsequently, column i of K^r is the xor of column $i-1$ of K^r and of column i of K^{r-1}.

2.2 Overall procedure

For AES-128, we divide the entire space of 2^{128} keys into 2^{124} non-overlapping groups of 2^4 keys each. We start modifications in the first rounds of AES-128, as opposed to the attacks of [2], where the modifications are done in the last rounds.

In each key group, we fix a base key and enumerate all the other keys in the key group with respect to it. The enumeration is performed in a way that allows to efficiently compute the intermediate states corresponding to the keys tested in a key group. To attain that, we modify the base key at two byte positions independently (in 2^2 ways each) and follow the propagation of those modifications forwards and backwards.

Now, starting with those intermediate states, a meet-in-the-middle key recovery with partial matching (in several state bytes) and splice-and-cut technique (going over the encryption oracle from the generated plaintexts to the corresponding ciphertexts) is applied. For each combination of plaintexts and intermediate states, corresponding to some keys, it is tested if there is a match.

2.3 Key space partitioning

The AES-128 key is entirely defined by the first subkey \$0 which coincides with the master key. We enumerate the groups of first subkeys by 2^{124} base keys. The base keys are all possible 2^{124} 16-byte values with two bytes that have zeros in two bits each:

where $x = (x_0x_1x_2x_3x_4x_500)_2$ and $y = (y_0y_1y_2y_3y_4y_500)_2$ with two lest significant bits set to zero. Within each group of keys, these bit values in the base key are also fixed, delivering a fixed known 16-byte base key for each group.

The 2^4 keys in a group are enumerated by all possible byte differences $a = (000000a_1a_0)_2$ and $b = (000000b_1b_0)_2$ with respect to the base key of this group:

with bits a_1, a_0, b_1 and b_0 running over all possible values.

This yields the full coverage of the first subkey space by the 2^{124} non-intersecting groups of 2^4 keys each, thus, providing the full coverage of the AES-128 key space.

2.4 Key testing

At this point, we assume that a base key is fixed, which defines a group of 2^4 keys. We apply a meet-in-the-middle technique with matching on several bytes of ciphertext. In the group of keys, the accommodated cost of the computation of those matching bytes from above and below for one key is significantly less than one AES-128 run. This is the major source of the complexity reduction.

In each key group, we modify the keys in the key search by a- and b-differences. The modifications are such that over all key groups in total there are only 16 plaintext-ciphertext pairs. The key modifications define plaintexts $P_{a,b}$. The corresponding ciphertexts are derived from $P_{a,b}$ using the encryption oracle.

The key recovery procedure is two-step. First, we generate intermediate states $S_{a,b}$ at #5 efficiently. Second, starting with $S_{a,b}$, we compute several bytes of ciphertext and check for a match with the ciphertexts obtained from $P_{a,b}$.

The first step of generating $S_{a,b}$ is illustrated in Figure 1 and can be outlined as follows:

1. Encrypt the all-zero plaintext with the base key (base computation in Figure 1) and store the state S_0 at #5 and \$2.
2. For each a, inject the a difference (a-modification in Figure 1) in the first subkey (two bytes). Compute and store S_a at #5 and \$2. for each a. Only the bytes depending on the a-modification have to be computed for that, see the figure.
3. For each b, inject the b difference (b-modification in Figure 1) in the first subkey (two bytes). Compute and store S_b at #5 and \$2 for each b. Only the bytes depending on the b-modification have to be computed for that, see the figure.
4. Now inject each combination of differences a and b, 2^4 times altogether (the a- and b-modification in Figure 1) — for each key in the key group. Each combination gives one plaintext $P_{a,b}$ and one state value $S_{a,b}$ at #5 and \$2. The plaintext $P_{a,b}$ and the intermediate state $S_{a,b}$ of the a-and b-modification are influenced both by a and b. $P_{a,b}$ can be obtained by direct combination of the zero plaintext in the base computation and plaintexts P_a and P_b in the a- and b-computations. In order to compute $S_{a,b}$, one combines the relevant parts of S_0, S_a and S_b. Note that the upper right byte of $S_{a,b}$ depends on both S_a and S_b. It can be computed by just xoring the corresponding bytes of S_a and S_b.

Thus, having computed S_a and S_b separately for a and b, one efficiently derives $S_{a,b}$, given the structure of the AES.

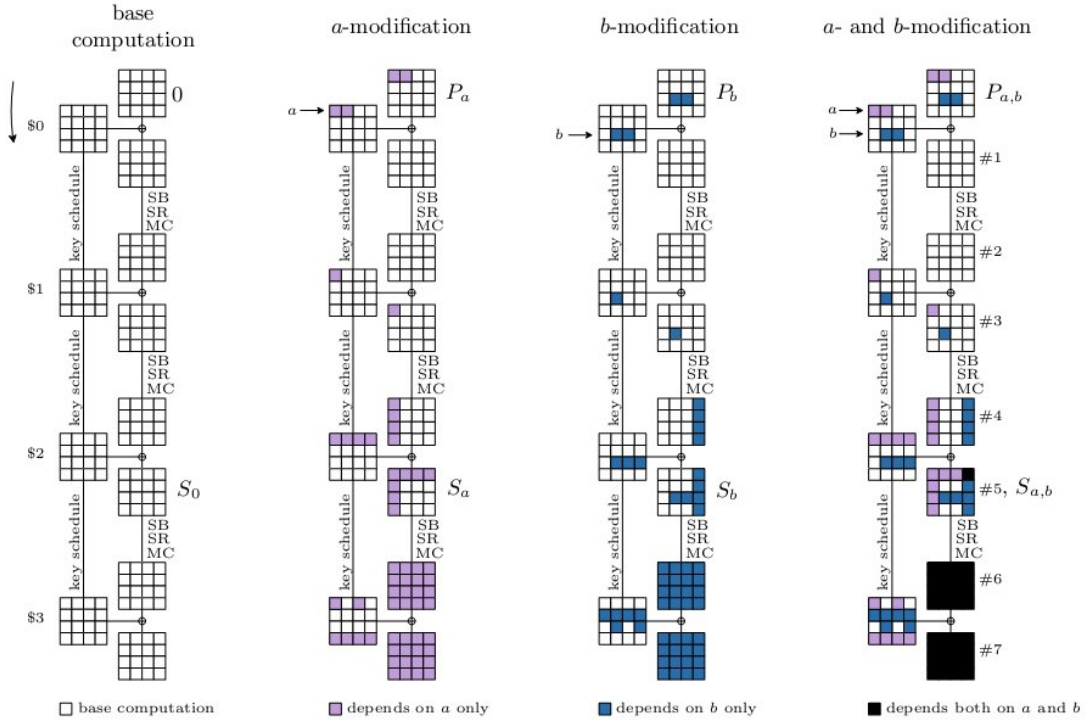


Fig. 1. Step 1: Savings of computations in the first three rounds

The second step is matching $S_{a,b}$ with the ciphertext which requires recomputations of some S-boxes of the cipher for every key. The procedure is as follows and is illustrated in Figure 2:

1. In SubBytes of round 3, right after #5, we need to recompute only one S-box for each key. The computation of the MixColumns operation will depend on both key modifications.
2. In SubBytes and MixColumns of rounds 4–7, all bytes need to be recomputed.
3. In SubBytes of round 8, all S-boxes are recomputed. In MixColumns of round 8, we only need to recompute 4 bytes.
4. In SubBytes and MixColumns of round 9 as well as in SubBytes of round 10, we 4 bytes are recomputed.
5. We match the recomputed 4 bytes of #21 with the ciphertext obtained using the encryption oracle from $P_{a,b}$.
6. Test each of the surviving keys using at most one additional plaintext-ciphertext pair.

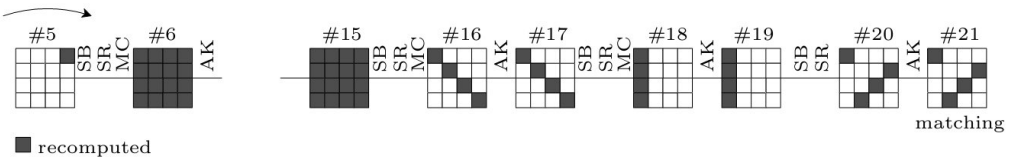


Fig. 2. Step 2: Recomputation at matching

2.5 Complexities

Within each of 2^{124} groups, the 2^2 a-only and 2^2 b-only propagations are computed for a part of the cipher. The states $S_{a,b}$ can be combined directly from S_a and S_b. The computational complexity to precompute all S_a and S_b in each key group is at most 0.3 AES-128 runs (the first step). Once $S_{a,b}$ and the ciphertexts are given, the computational complexity is mostly determined by the parts of states one has to recompute for each combination of a- and b-propagations to enable matching. This requires about 7.12 AES-128 runs to test all 16 keys in the key group (the second step). Handling false positives has a relatively negligible computational complexity, since it happens once among 2^{32} keys tested on average. This delivers a computational complexity of

$$2^{124}(0.3 + 7.12) = 2^{126.89} \text{ AES executions.}$$

As to the data complexity, varying 2 bits in a and 2 bits in b modifications propagate to a maximum of 4 bits changed in total in the plaintext. The change is with respect to the zero base plaintext (chosen plaintexts of the base computation). This yields data complexity of only 16 chosen plaintexts (this is to be compared with 2^{88} chosen ciphertexts needed in the biclique cryptanalysis of AES-128 in [2]).

Since the coverage of the key space by groups around base keys is complete, we cannot miss the right key and the success probability is 1.

The memory requirement is the storage of 2^2 a-only and 2^2 b-only propagation values, which is negligible, since one has to store only relevant parts.

3 Implementation

The biclique attack on AES-128 has been adapted for hardware implementation as outlined in the previous sections and realized on the RIVYERA [13] computing cluster. The RIVYERA computing-architecture consists of 128 Xilinx Spartan3 XC3S500 high performance FPGAs with an equivalent computing power of 640 million system gates and is ideal for parallel computing applications, including cryptanalysis. In addition, the low energy-consumption and the improved bus-system makes RIVYERA a perfect choice for setting up high-density and large-scale FPGA clusters where supercomputer performance is required. On the other hand, our design is also synthesized in ASIC technology to compare the advantage factor with the gain that FPGA implementation provides.

For a realistic and fair quantification of the performance improvement obtained by the biclique attack, first, an optimized brute-force attack on AES-128 was realized. In the implementation of both attacks, highest possible performance on the RIVYERA and ASIC platforms has been targeted, and for this purpose, several different implementations have been evaluated.

During the evaluation phase, we observed that the performance bottleneck, both in terms of speed and area, is caused by the S-boxes. Therefore, most of the design effort was concentrated on high-speed and low-slice count realization of S-boxes. While implementations involving utilization of dual-port RAMs as lookup table based S-boxes resulted in the best slice numbers, they also suffered from poor speed performance, mainly due to the high routing delays between S-boxes and the other functional blocks implemented on slices. On the other hand, fully combinational implementation of lookup tables for S-boxes resulted in the worst performance both in terms of speed and area. The best choice left was to implement the S-boxes using composite field inverters [4]. Again, various composite fields were evaluated and the best speed and area performance was obtained via the $GF(((2^2)^2)^2)$ composite field in [12]. Furthermore, aggressive pipelining was applied to both the S-boxes as well as the other computation units within the rounds for higher speed-up and as a consequence fully unfolded pipelined designs were realized for both the naïve and biclique attacks.

In the following subsections, implementation details of both designs will be given together with the performance figures and comparisons, as well as the architectural details.

3.1 Reference point: Optimized brute-force attack on AES-128

In the implementation of the optimized brute-force attack on AES-128, our target was to achieve the highest possible performance on the RIVYERA platform in order to have a fair comparison. We achieved our target by evaluating several different implementations with different stages of pipeline.

For recovering the secret key of a given plaintext-ciphertext pair by brute-force attack, all possible key combinations should be tested. Considering AES-128, this corresponds to 2^{128} tests. In our implementation, we were able to fit two optimized brute-force attack cores on each of the 128 FPGAs on the RIVYERA engine. The most-significant 7-bits of the 128-bit key is used as the FPGA identifier, and are therefore fixed for each FPGA. The following 1-bit is used as the core identified. The rest 120 bits were generated by a pipelined, high-speed 120-bit counter, independently for each core.

Figure 3 shows the explained input key scheme together with one of the two AES cores. Each of the these naïve cores consists of 10 unrolled and fully pipelined AES rounds, where the known plaintext is also taken as an input and the corresponding ciphertext is compared with the output ciphertext. The output matching is also done in a pipelined manner: In the first stage of the pipeline, comparison on each byte is done separately resulting in 16 independent bytewise comparison results. In the next stage, the 16-bit result vector is checked to see if it is all 1's. The pipelined key counter and comparison stages are built to ensure the same target frequency is achieved within the rounds.

In the following subsections, the proposed architecture with the details of pipeline stages and the implementation aspects for FPGA and ASIC is presented.

The proposed architecture The highest speed implementation of the AES algorithm can be achieved via a fully unrolled and pipelined design [4]. Sharing the datapath for different rounds not only results in additional multiplexer delays, but it also renders the pipeline within rounds impossible. However, pipelining rounds is not sufficient. It should also be applied within each round to achieve the highest possible throughput. The improved brute-force attack is implemented as 10 unrolled full rounds (Figure 4), where an 11-stage pipeline is implemented within each round. To get the best speed/area performance, we implemented different number of pipeline stages to the S-boxes, MixColumns modules and in between the rounds. With less number of pipeline stages, we were able to fit up to 4 cores in one FPGA. However, in that case the maximum operating frequency dropped by more than a factor of 2, resulting in a worse average throughput per FPGA. Therefore, we opted for 2 core per FPGA and targeted to achieve the highest possible frequency by utilizing the unused slices as additional pipeline stage. We have tried several different pipeline configurations, as well as asymmetric pipeline construction. In the end, we opted for an 11-stage pipeline identical in every round, which gave the optimum performance result, and used it in each round.

As already stated in previous sections; one round of AES is composed of four steps, which are repeated in 10 rounds for 128-bit key and these steps are SubBytes (S-box), ShiftRows, MixColumns and AddKey. ShiftRows step is only an interconnection with no logic cost and AddKey step is XORing the data and the subkey. In MixColumns step, the elements of data in each column are permuted using a chain of XOR operations. The most expensive step is the S-box phase, as mentioned before. It is the slowest operation of AES. In S-box step, the input is considered as an element of $GF(2^8)$. First, its multiplicative inverse is calculated and then an affine transformation over $GF(2)$ is applied. The S-box can be implemented using finite field

operations. However, the calculation of the inverse of elements is expensive. An algorithm can be used to calculate the multiplicative inversion in $GF(2^8)$ using the $GF(((2^2)^2)^2)$ composite field operations [12]. Although the composite field implementation is area-efficient, it has a very long critical path. This long critical path is the main bottleneck in terms of overal speed and throughput. It is the main reason why we ended up with so many (11) stages of pipeline for each round. 8 of these stages are solely used within the S-Box.

Since pipelining the S-box directly affects the number of registers, it has to be done cautiously. Otherwise, increase in the area can be much more than expected if the pipeline registers are not placed properly. In our design, the critical path with the S-Box is broken in 8 stages, where 6 of those stages lie inside the inverter. Figure 5 shows the pipelined architecture of S-box phase and its components.

In addition to the pipelined stages inside S-box, there are 3 more pipeline stages between operational steps: One after the SubBytes operation, one after the MixColumns operation and finally one after the KeyAdd operation. Note that, pipeline registers are also added to key scheduling to be synchronized with the state processing. Therefore, in total, optimum pipelined implementation has 11 pipeline stages for each round of AES. Figure 6 shows the pipelined architecture of one AES round.

Implementation on FPGA Our proposed design is first implemented on RIVYERA, where 128 Xilinx Spartan3 XC3S500 FPGAs are used. The Synplicity tool is used for the synthesis of the design, which gives superior results with respect to native Xilinx ISE synthesizer, especially in register based design. For the place and route phase, the Xilinx ISE tool 12.3 is used, with the "continue on impossible" option on for the routing. The performance results of our proposed architecture on FPGA is shown in Table 1.

Table 1. FPGA implementation results for naïve attack FPGA (2 cores/FPGA)

Slice Utilization	% FPGA Utilization	Maximum Frequency (MHz)	Keys tested/sec/FPGA
26949 / 33278	80.98	263.16 MHz	526×10^6

Implementation on ASIC The proposed design is also realized on ASIC. The register based structure of our design, which uses no FPGA specific features, allowed us to directly use the existing RTL code for ASIC implementation. Therefore, we were able to use same design for both implementations. In the implementation process, Cadence Encounter RTL Compiler v10.1 for synthesis. The implementation has been synthesized with 45 nm generic NANGATE standard cell library. In the synthesis, typical operating conditions were assumed. The performance results of our proposed architecture on ASIC is shown in Table 2.

Table 2. ASIC synthesis results for naïve attack core

Core Area (GE)	Maximum Frequency (MHz)	Average Power (mW)	Keys tested/mW
362181	2480	622.937	3.98×10^6

3.2 Biclique Attack

We have realized two hardware architectures for the biclique attack. The first one is a conceptual architecture that maps the theoretical attack directly to hardware. It is a storage based architecture, and heavily relies on precomputation and storage of biclique states within the memory. However, practical limitations associated with memory usage makes this architecture practically inefficient. Therefore we have only used it to model our actual architecture, which depends on

on-the-fly computation of biclique states. We name it as the "recomputation" or "virtual storage" architecture. In addition to its lower memory requirements, it is also suited better for a pipelined implementation, making it possible to re-use the modules implemented for the naïve attack. In the following subsections, the details of each architecture will be outlined together with implementation results.

Conceptual Attack Architecture Using Storage Figure 7 shows the storage based architecture. It is based on precomputation of all base states as well as delta and nabla differences for each biclique trail, and storing them inside the memory. Then for each biclique state byte, the corresponding base and delta and/or nabla state bytes are read from the memory, and summed. The operation of this architecture can be summarized as follows:

- For a new key group, the precomputation engine computes all the base and biclique difference states and stores them in the memory. Practically, the precomputation engine is composed of a single S-Box, a single (or even a partial) MixColumns module and as many bytes of memory required for the target data complexity.
- While the precomputation for the new key group is in progress, the precomputed values for the previous key group are used by the biclique brute-force attack engine to attack that key group. The engine reads the precomputed base state and biclique difference state values stored inside the memory and sums them to form the biclique trails.

In theory, this seems to be the most resource efficient approach. It requires only a single S-Box and RAM based memories. However, there are several practical obstacles to make this approach infeasible for hardware implementation, especially on FPGA:

- The case for the single S-Box is only valid for relatively high data complexities, i.e. the single S-Box should have clock cycles sufficient to compute all target base and biclique state bytes until the key group changes. The key group change occurs in every 2^{2d} cycles. This means the total number of base and biclique bytes on the biclique trails should be less than 2^{2d}. For example, for the trails we have chosen, the first three rounds are on the biclique part of the attack. This means 48 base state bytes. There are 7 bytes on the delta trail and 6 bytes on the nabla trail. The sum of these two has to be multiplied by 2^d, resulting in 13×2^d biclique states. On top of this, S-Box precomputation numbers for key expansion has to be added as well. This is another 8 base states and 2×2^d biclique states (2^d for delta and 2^d for nabla, respectively). As a result, a total number of $56 + 15 \times 2^d$ S-Box precomputations are required, and it should be completed within 2^{2d} cycles, i.e. $56 + 15 \times 2^d \leq 2^{2d}$. This means, d should at least be equal to 4, which corresponds to 256 plaintext-ciphertext pairs. Furthermore, there is a "black" byte on round-3, which corresponds to a non-linear combination of the delta and nabla trails and requires its own S-Box. The whole cycle computation gets even worse, when a pipelined S-Box is used.
- Memory access will be a major problem. Each of the base, delta and nabla difference states have to have their seperate memories to allow parallel reads. The memories should be dual-port memories or registers, since the precomputation engine will need values of precomputed bytes of the preceeding round in order to precompute bytes of a target round. Furthermore, a "ping-pong memory" scheme is required: The currently computed states are stored to the "ping" memory, while the precomputed states are read from the "pong" memory. There will be a lot of independent memory module which are operated in parallel and has to be accessed in parallel. This poses a practical interconnection delay problem due to fixed topologies of RAMs on the FPGA. It even applies to ASICs, where the physical size of the RAMs are a function of the size of the memory. For very small memories (for example 256 byte memories,

which will be needed in the case of $d = 4$), the decoding logic of the memory becomes the determining factor for the physical size of the memory module.

Of course, it is possible to overcome these obstacles, at least partially. A larger d can be selected (for example, $d = 5$). This will leave enough cycles for all the state computations, even considering the internal pipeline of the S-Box. It will also make the memory sizes (also physically) feasible for implementation, at least on ASIC. Chosen biclique trails can be modified to ones which require lower number of plaintext-ciphertext pairs, i.e. 2^d instead of 2^{2d}, in which case only 32 pairs will be sufficient for $d = 5$. It is also possible to make this architecture reconfigurable by a simple controller that determines the source and target of the S-Box. It will then be much simpler to change the biclique trails.

However, the interconnection delay problem on the distantly located FPGA memories will persist in any of the cases above. One solution would be to replace RAM based memories with registers. While solving most of the memory size related problems, this will also mean use of several of the FPGA resources (slices) for storage only. In our case, this is undesirable, since our main target is to fit as many of biclique attack engines on one FPGA as possible in order to achieve a high advantage factor.

Considering all these factors, we opted not to implement this architecture, and search for alternative approaches instead.

Actual Attack Architecture Using Recomputation (Virtual Storage) Our recomputation model is based on the idea of computing the base and biclique difference state bytes on-the-fly instead of precomputation. This approach not only allows us to prevent RAM usage, but it also allows us to seamlessly integrate the biclique parts of the attack into the rest of the architecture. In this approach, the base states are computed using serialized AES rounds, where biclique-specific S-Boxes are operated in parallel with the serialized round in order to compute the biclique difference states for delta and nabla trails. The computed base state bytes are serially sent to the next round without any additional temporary storage, while the biclique states are partially stored. Therefore, this approach is not completely storage-free. However, the partial storage is embedded into the serial pipeline running parallel with the serialized flow of base states. In this sense, it can be considered as a virtual storage. This structure also minimizes the interconnection by reducing the whole data flow effectively to two byte-serial shift-registers running in parallel (and a third one for the key expansion). A single large double buffer (for serial-to-parallel conversion) is required only at the crossing from biclique processing to regular parallel rounds. However, it practically is the first pipeline stage of the first regular round, therefore its effective cost is zero.

We start with the detailed description of the recomputation architecture with its overall schematic shown in Figure 8. The arhitecture consists of three main parts:

- Biclique rounds in the beginning: The first three rounds realize the biclique part of the attack. Each of these rounds are individually designed to implement the biclique trails given in Section 2.4 using the "recomputation" model which will be explained below.
- Regular rounds in the middle: Rounds four to seven of AES-128 are regular pipelined rounds similar to the ones used in the naïve attack. However, due to the limited resources left on the FPGA after fitting four biclique attack engines, we had to implement a 7-stage pipeline instead of 11. This resulted in an 8% drop in the maximum frequency.
- Reduced rounds in the end: Rounds eight to ten are reduced versions of the 7-stage pipelined regular rounds. Reduction comes from partial matching at the output. Instead of the full 16-byte output, only 4-bytes are used for ciphertext matching. These bytes are chosen in a way to ensure minimal S-Box usage as shown in Figure 1. Partial matching results in a 2% reduction in slice usage per core. It also gives an extremely low probability (1 in every 2^32)

of false alarm rate. Each false alarm key is sent to the main processor on the RIVYERA engine for offline verification. With such a low probability, this scheme requires virtually zero communication bandwidth between each FPGA and the main processor.

As a first step of the realization of the biclique rounds, we carefully selected the data complexity d. As explained before, there are 2^{2d} cycles between two consecutive key groups. Therefore, computation of a full set of states (both base and biclique) must be completed within this period. On the other hand, full AES state is composed of 16-bytes. Since we are trying to keep the S-Box cost as low as possible to be congruent with the original attack scenario, we started with the assumption of using a single S-Box for the computation of base states within one round. This simply corresponds to $d = 2$ with full utilization of all 16 cycles between two key groups for the computation of all state bytes within a single AES round. This can be easily achieved via a byte-serial AES implementation similar to [3], which is also the most compact AES implementation reported so far.

Selection of a higher d will result in more biclique states to be recomputed, and a more complex interconnection network. Lowering d to the minimum value of 1 will require double the number of S-Boxes for the base state computations. Selection of $d = 2$ will require a total number of 16 plaintext-ciphertext pairs, which is also easily implementable, especially considering that only 4-bytes of the 16 possible ciphertexts have to be stored. It can efficiently be implemented as a lookup table on the FPGA slices.

After the selection of $d = 2$, we designed each of the rounds in a custom manner as explained below:

- **Round-1:** This round is a modified version of the serial AES structure given in [3]. Of course, the serial AES structure is unrolled to form only a single AES round within the overall pipeline. Since the S-Box is the first element with the serial data flow, it can also be implemented with internal pipeline without distorting the overall data flow. Its only effect will be additional initial pipeline delay. However, the main difference is the byte order. In the original serial design (as with most other reported serial implementations), data processing begins with the most significant (leftmost) byte and proceeds towards the least significant (rightmost) byte. In our case, we start with the rightmost byte. This is done basically to be compatible with the key generator, which is a byte-serial up-counter. It also starts with the least significant byte, which is incremented by adding 1 on to the current value. The carry from that addition goes to the next byte as the addition value, and so on, until the core ID and FPGA ID bits, which are fixed and selected via multiplexers. Therefore the first byte going into the serial AES Round-1 is the least significant byte. There is no additional cost of processing state bytes in reverse, except for the change in the order of operations in MixColums block. There are no biclique trails, hence no biclique state processing, in this round. The Round-1 module is shown in Figure 9.
- **Round-2:** This round is very similar to Round-1, except, in this round there is additional logic for biclique state processing. In parallel with the serial AES round identical to Round-1, a second S-Box is operated in 6 of the 16 cycles of each key group. These 6 cycles are equally shared between the delta and nabla traces, for each of which there exists 3 non-zero difference values. As shown in Figure 10, two temporary byte sized registers hold the values of incoming base state bytes 6 and 0. As with the incoming base state values, the output of the base state S-Box for bytes 6 and 0 are also stored in two other temporary registers. To the state byte 6 (which comes first due to reverse order byte processing), non-zero biclique value of $0x01$, $0x02$ and $0x03$ are added in order and sent to the additional biclique S-Box. The outputs of this S-Box for three biclique values are added to the output of the base S-Box for byte 6 in order to get the nabla difference, which is serially pushed into a FIFO register of depth-6 and width-8 (i.e. 1 byte). The same is repeated for the delta differences, but this time using the

base state byte 0. When all 6 byte delta and nabla values are ready, they are kept inside the FIFO until base state processing of all 16 bytes in the serial AES round are complete. Each of the delta and nabla byte triplets are then padded with a zero byte (that corresponds to the $0x00$ biclique difference and parallely loaded into two rotating registers, whose outputs are added onto the base state output in order to provide the biclique state inputs for Round-3. The key processing of this round is identical to that of Round-2, since the only biclique key byte operations are direct addition of four possible biclique differences, which is incorporated into Round-3 key expansion for convenience.

- **Round-3:** The last biclique round is also based on serial processing of state bytes, but it is largely different from the other two rounds. Instead of a single serial path for base state bytes and a partial serial path for biclique difference states, there are four serial paths operating in parallel. Instead of computing base and biclique difference states seperately, the four serial paths calculate the final states of the third round directly. In this round, there are four base state bytes (5,7,9,11), six delta biclique state bytes (0,1,2,3,4,8), five nabla biclique state bytes (6,10,13,14,15) and one combined biclique state byte. Each of the delta and nabla state bytes requires $2^d = 4$ S-Box computations (including the zero biclique value of $0x00$). The combined state byte requires $2^{2d} = 16$ S-Box computations, and each of the base state bytes requires a single S-Box computation. This adds up to a total of $11 \times 4 + 1 \times 16 + 4 \times 1 = 64$ S-Box computations that have to be completed within 16 cycles. Hence with a perfect timing, exactly 4 S-Boxes on 4 serial paths are sufficient. Since in this round, our aim to calculate the final states (i.e. after S-Box operation) directly, we have to also consider the directly added non-zero biclique values that should ideally come from the key expansion of Round-2. However, we incorporate it into this round by directly adding the target value on the input byte of the round. In this scheme, each input byte is either passed as is (in case of a base byte), summed with the delta/nabla non-zero biclique value, summed with the delta/nabla S-Box output difference value calculated in Round-2, or a combination of both. In case of the single combined state byte, value added onto the input byte 12 is not "either delta or nabla", but instead "both delta and nabla", therefore resulting in 16 disctinct values at the S-Box output. The only costs associated with this scheme are two 2-bit counters for direct biclique values and multiplexers at the adder (XOR) input to choose a combination of the biclique inputs. As shown in Figure 11, although the work load of each serial path is exactly 16 S-Box computations per key group, distribution of the output bytes per path varies: Path-0 serves bytes 15, 11, 9, 7, 5, 4, 2; path-1 serves bytes 14, 10, 6, 1; path-2 serves bytes 13, 8, 3, 0; and path-3 serves only byte-12 (combined biclique state byte). The outputs for each byte are then directed to FIFOs of 4-byte depth (if delta or nabla bicliques), or a FIFO of 16-byte depth (if combined biclique), or a single byte register (if base); in a similar way as done in Round-2. The next and final stage is a combination of rotational registers (of corresponding sizes), which are parallelly loaded from FIFOs at the beginning of each key group, and then rotated every 4 cycles (if delta biclique), or every cycle (if nabla or combined biclique) in order to form the 2^{16} possible input combinations at the MixColumns module (which is a parallel implementation). Key expansion module is a combination of the regular key expansion unit (also used in Round-1 and Round-2) and a register bank. Since the base states in the key expansion require only 4 S-Box computations, the required $3 + 3 = 6$ biclique S-Box computations can also be done using the same S-Box within the allowed 16 cycles, with careful timing.

Implementation on FPGA We have optimized our design in a way to fit four biclique engines (each with its own key generation and output matching circuitry) on each of the 128 Xilinx Spartan3 XC3S500 FPGAs on RIVYERA. As in the naïve attack implementation, we used Synplicity for synthesis. Also, the place and route was done using Xilinx ISE 12.3, again with the "continue on impossible" option on for the routing. Our major problem was the extemely tight utilization

of resources. Due to the slices required for RIVYERA configuration, the upper limit of the FPGA area that we were allowed to use was practically 92%. Initially, slice utilization was more than 96%. We had to hand-tailor parts of the design, especially simple looking control logic sections (mostly redundant registers for temporary storage due to poorly trimmed timing controls). Only after than, we were able to fit into 92% of each FPGA. Without the "continue on impossible" option, maximum reachable frequency was around 207 MHz. After enabling that option, the operating frequency increased to 236 MHz, drastically. We did not use any of the block RAMs on the FPGA, due to the register based pipelined structure of our design. Table 3 lists the performance results of our proposed architecture on FPGA.

Table 3. FPGA implementation results for biclique attack FPGA (4 cores/FPGA)

Slice Utilization	% FPGA Utilization	Maximum Frequency (MHz)	Keys tested/sec/FPGA
30720 / 33278	92.31	236.22 MHz	945×10^6

Implementation on ASIC As in the naïve attack case, we also synthesized the biclique attack architecture on ASIC using the NANGATE generic 45 nm standard cell library. As stated before, the register based structure of our design allowed us to directly use the existing RTL code. Again, Cadence Encounter RTL Compiler v10.1 was used for synthesis with typical operating conditions were assumed. Surprisingly, the advantage factor obtained for both FPGA and ASIC were different by 20% (1.8 in FPGA, 1.45 in ASIC) when time-area product was considered. In case of using power as the target advantage factor (keys tested per mW) for ASIC implementation, the advantage factor was again 1.8. Performance results of our proposed biclique attack architecture on ASIC are shown in Table 4.

Table 4. ASIC synthesis results for biclique attack core

Core Area (GE)	Maximum Frequency (MHz)	Average Power (mW)	Keys tested/mW
163912	1548	211.545	7.32×10^6

4 Conclusion

We have presented optimized hardware architectures for (1) brute-force key-search of AES, and (2) biclique key-search for AES using a tailor-made attack with practical data complexity. Despite the biclique technique being structurally more complicated, we can report almost a factor 2 speed and cost gain. For this conclusion we studied both FPGA and ASIC-based approaches.

For all combinations of approaches, the S-box implementations are in fact the bottlenecks. Hence, future improvements for AES circuits targeting this bottleneck will likely affect all our implementations in the same way. This further strengthens our conclusion that real-world cost saving for AES key search can indeed be achieved using bilcique cryptanalysis, *regardless* of the used technology.

References

1. 3rd Generation Partnership Program. Document 2: Kasumi specification. technical specification 35.202. release 5. version 5.0.0.
2. Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique Cryptanalysis of the Full AES. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 344–371. Springer, 2011.
3. Panu Hämäläinen, Timo Alho, Marko Hännikäinen, and Timo D. Hämäläinen. Design and implementation of low-area and low-power aes encryption hardware core. In *DSD*, pages 577–583. IEEE Computer Society, 2006.

4. Alireza Hodjat and Ingrid Verbauwhede. Area-throughput trade-offs for fully pipelined 30 to 70 gbits/s aes processors. *IEEE Trans. Computers*, 55(4):366–372, 2006.
5. Dmitry Khovratovich, Gaëtan Leurent, and Christian Rechberger. Narrow Bicliques: Cryptanalysis of Full IDEA . To appear in Eurocrypt 2012.
6. Dmitry Khovratovich, Christian Rechberger, and Alexandra Savelieva. Bicliques for preimages: Attacks on Skein-512 and the SHA-2 family. To appear at FSE'12. Available online at http://eprint.iacr.org/2011/286.pdf, 2011.
7. Sandeep S. 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 *Lecture Notes in Computer Science*, pages 101–118. Springer, 2006.
8. Xuejia Lai, James L. Massey, and Sean Murphy. Markov ciphers and differential cryptanalysis. In *Advances in Cryptology – CRYPTO '91*, pages 17–38. Springer-Verlag, 1991.
9. Mitsuru Matsui. New block encryption algorithm misty. In Eli Biham, editor, *FSE*, volume 1267 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 1997.
10. NIST. FIPS-197 Advanced Encryption Standard (AES).
11. Yu Sasaki and Kazumaro Aoki. Finding Preimages in Full MD5 Faster Than Exhaustive Search. In Antoine Joux, editor, *EUROCRYPT*, volume 5479 of *Lecture Notes in Computer Science*, pages 134–152. Springer, 2009.
12. Akashi Satoh and Sumio Morioka. Hardware-focused performance comparison for the standard block ciphers aes, camellia, and triple-des. In Colin Boyd and Wenbo Mao, editors, *ISC*, volume 2851 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2003.
13. SciEngines. RIVYERA S3-500. http://www.sciengines.com/products/computers-and-clusters/rivyera-s3-5000.html.

A Block Diagrams

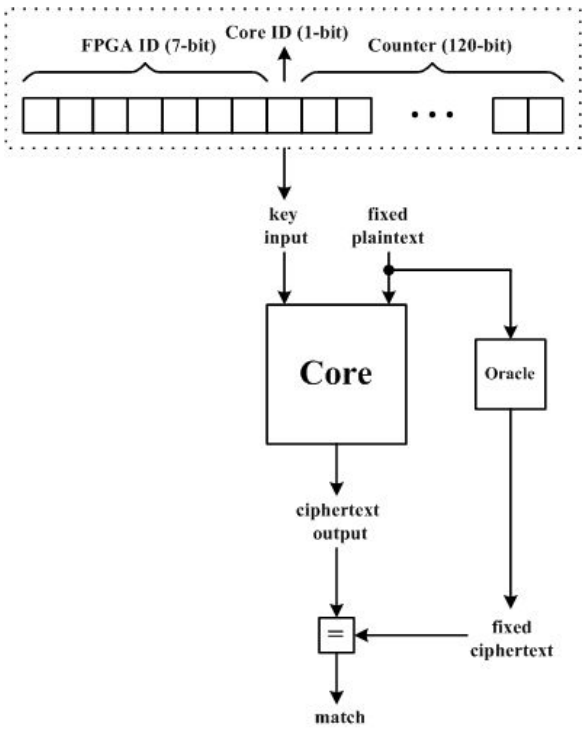


Fig. 3. General architecture of the proposed AES core with key generator scheme

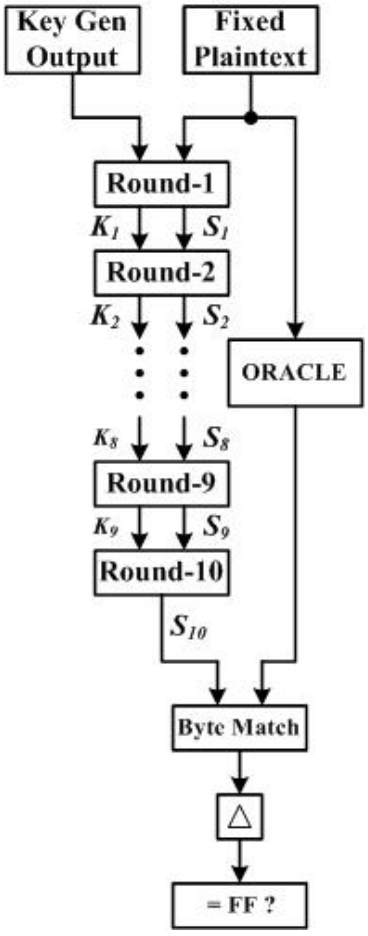


Fig. 4. AES core block diagram

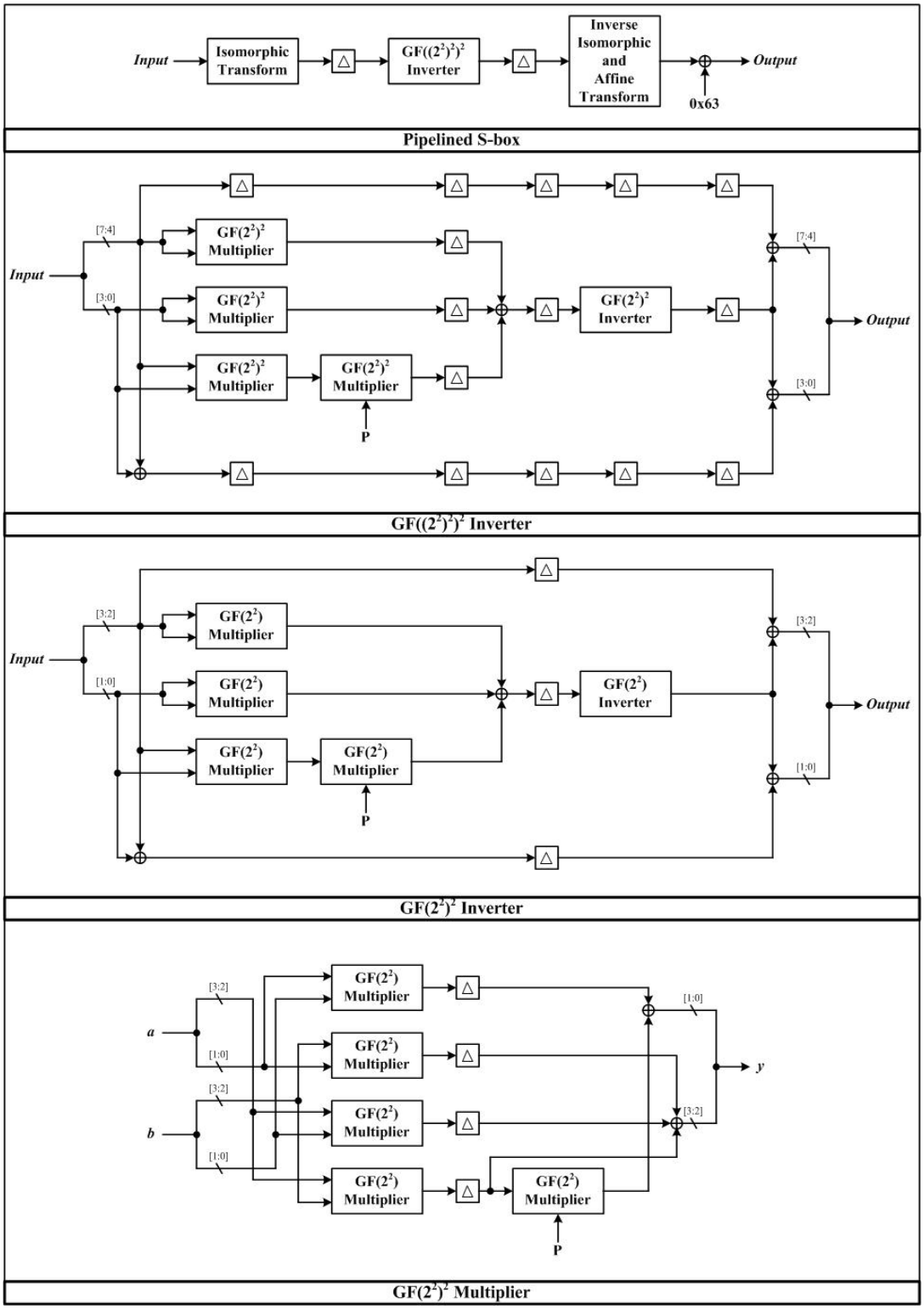


Fig. 5. Pipelined S-box and its components

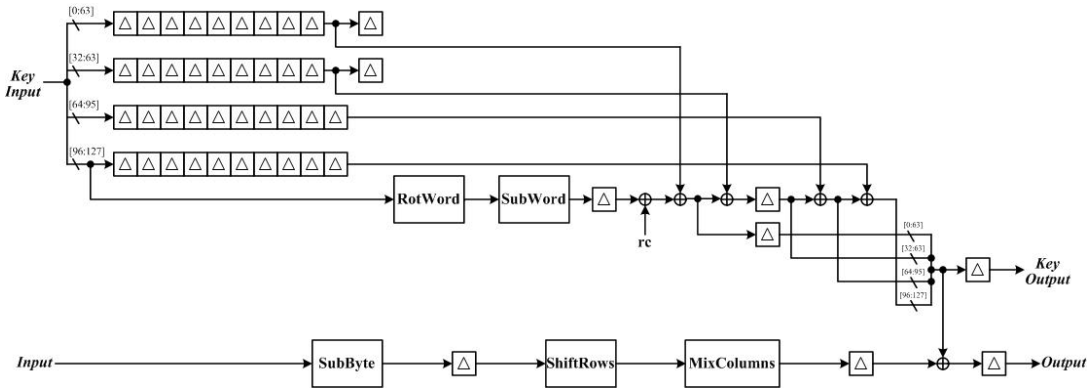


Fig. 6. One pipelined AES round

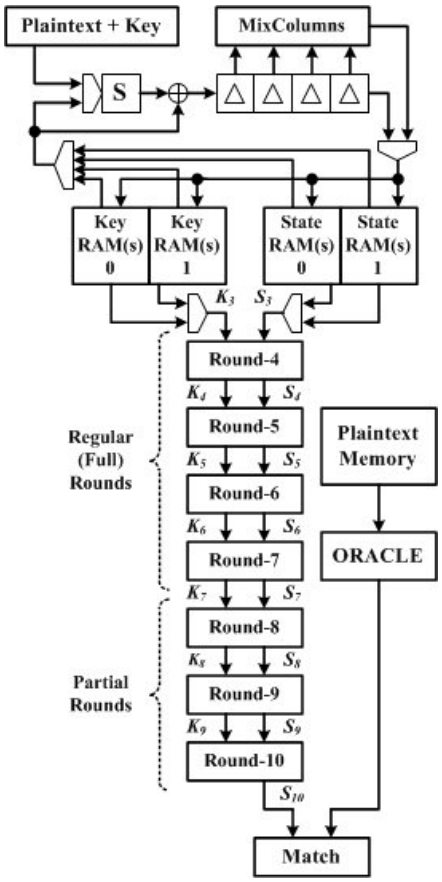


Fig. 7. Conceptual biclique attack architecture

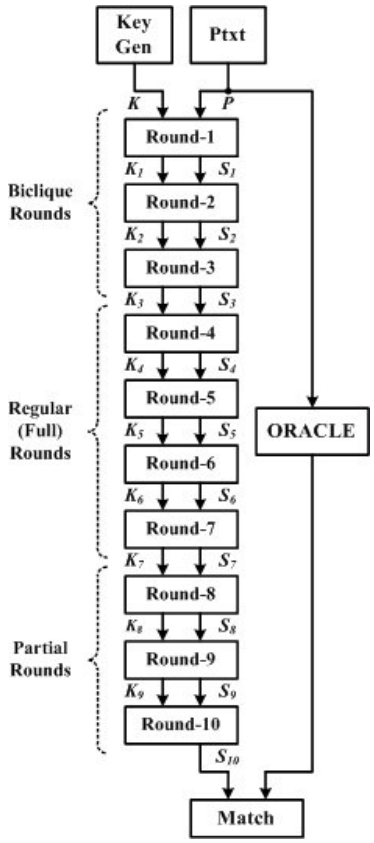


Fig. 8. Recomputation based biclique attack architecture

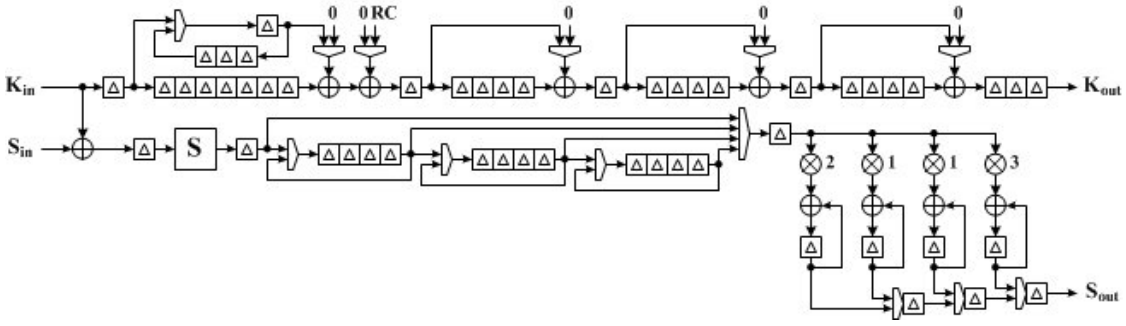


Fig. 9. Biclique attack engine Round-1

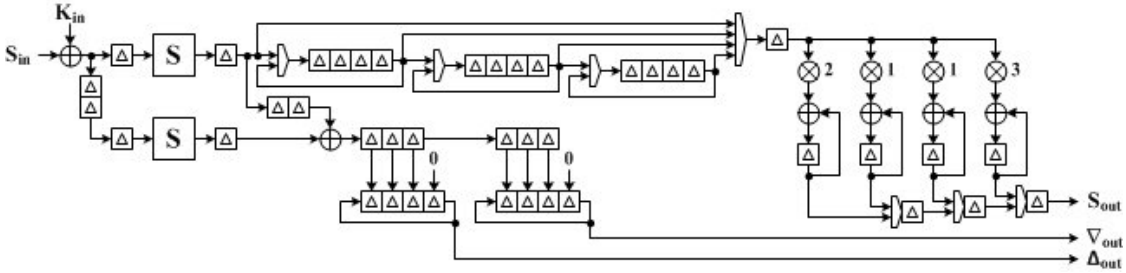


Fig. 10. Biclique attack engine Round-2

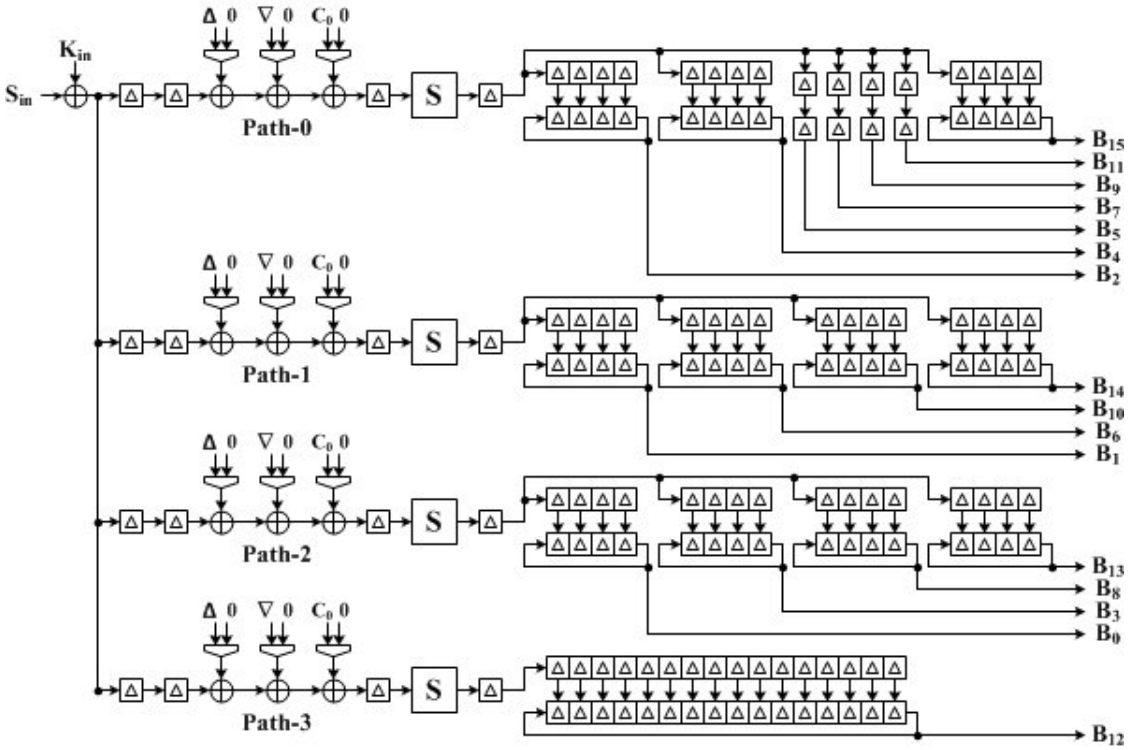


Fig. 11. Biclique attack engine Round-3

Speeding up GPU-based password cracking

Martijn Sprengers[1] and Lejla Batina[1,2]

[1] Radboud University Nijmegen, ICIS/Digital Security group
Heyendaalseweg 135, 6525 AJ Nijmegen, The Netherlands
sprengers.martijn@kpmg.nl, lejla@cs.ru.nl
[2] K.U.Leuven ESAT/SCD-COSIC and IBBT
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
lejla.batina@esat.kuleuven.be

Abstract. Recent advances in the graphics processing unit (GPU) hardware challenge the way we look at secure password storage. GPUs have proven to be suitable for cryptographic operations and provide a significant speedup in performance compared to traditional central processing units (CPUs). This research presents a proof of concept for the impact of launching an exhaustive search attack on the MD5-crypt password hashing scheme using modern GPUs. We show that it is possible to achieve a performance of 880 000 password hashes per second, using various optimization techniques. For our implementation, executed on a standard GPU, we obtain a speed-up of a factor of 30 when compared with equally priced CPU hardware. With this performance increase, 'complex' passwords with a length of 8 characters are now becoming feasible to crack even with inexpensive hardware.

Keywords: GPU, MD5-crypt, password hashing schemes, password cracking

1 Introduction

Since the 90's graphics processing units (GPUs) have been significantly improved. They have proven to be very suitable for processing parallel tasks and calculating floating point operations. While the advantages of GPUs in other areas (like graphical design and game-industry) have already been recognized, the cryptographic community was not able to use them due to the lack of support for integer arithmetic instructions and the lack of user-friendly programming APIs. However, GPU producers have dealt with those shortcomings and it is especially the parallel design of a GPU that makes it suitable for some cryptographic applications.

Many software services provide an authentication system that relies on a user name and password combination. To make sure that these passwords are still safe, even if the security of the password storage cannot be guaranteed, it is common to use a cryptographic hash function to calculate the digest of the password and store this together with the users credentials. Due to the cryptographic properties of the hash function, the digest of the password is

not easily reversible. Therefore the probability that an adversary learns partial information about the password should be proportional to the work he invests and the predictability of the password distribution. However, it is the latter property that fails for human generated passwords and therefore an adversary can mount an exhaustive search attack on the domain of the authentication mechanism. Password hashing schemes have been designed to disable such an attack, by increasing the complexity of the calculations with techniques like key-stretching [1,2]. In addition, schemes like Password-Based Key Derivation Function (PBKDF2) [3] have been proposed to derive a cryptographic key from low entropy passwords, e.g. by going beyond simple hashing. However, with the introduction of cryptography on new hardware platforms, such as FPGAs [4] and GPUs, it is doubtful if these schemes still provide enough security.

1.1 Related work

Thompson et al. [5] showed that GPUs could be used for general purpose computing, in addition to specific graphics applications. Not long after, Cook et al. [6] experimented with the applications of GPUs for cryptography. However, due to the lack of integer arithmetic and support of API's, no remarkable speedup was gained. When general programming API's for GPUs became available, Goodman et al. [7] were first to realize that contemporary GPUs can outperform high-performance CPUs on symmetric cryptographic computations, yielding speedups of at most 60 times for the DES symmetric key algorithm. Several GPU implementations of the AES algorithm followed [8,9,10,11]. In addition, cryptographic hash functions, such as MD5 [12,13] and Blowfish [14], and password hashing schemes, such as DES-crypt [15], have been implemented on graphic cards too, yielding substantial speed ups over CPUs.

The suitability of GPUs for asymmetric cryptography was first investigated by Szerwinski et al. [16] and Harrison et al. [17]. In particular, GPUs for the cryptanalysis of asymmetric cryptosystems was thoroughly investigated by Bernstein et al. [18]. They showed that it is possible to reach up to 481 million modular multiplications per second on a Nvidia GTX 295, in order to break the Certicom elliptic curve cryptosystem (ECC) challenge [19,20].

1.2 Contribution

This research investigates the impact of launching an exhaustive search attack on authentication mechanisms that use password hashing schemes based on a cryptographic hash function (such as MD5). We focus on efficient implementations of these password schemes on GPUs in order to initiate massive parallel execution paths at low cost compared to a typical CPU. In particular, we review the MD5-crypt password hashing scheme. MD5-crypt is used as the standard password hashing scheme in most Unix variants, such as BSD and Linux. Moreover, corporations like Cisco have it employed in their routers and the RIPE Network Coordination Centre stores the MD5-crypt hashes, used to authenticate their users, in public. If the security of MD5-crypt fails, it will

have a large impact on the confidentiality and integrity of systems and services. Since the security properties and design of this scheme have been the basis for other hashing schemes, e.g. SHA-crypt [21] and the PBKDF2 framework, our findings suggest that the security of similar schemes should be revisited.

The remainder of this paper is organized as follows. Section 2 describes the GPU architecture and its execution model. Section 3 elaborates on the possibilities for efficient implementations of the MD5-crypt password hashing scheme on a modern GPU. We describe some of our GPU optimization strategies in Section 4. Section 5 contains our experimental evaluation on which we base our conclusions (Section 6).

2 GPU hardware layout

There are multiple GPU manufactures, each using different chip sets and architectures. However, all GPU devices are based on the Single Instruction Multiple Thread (SIMT) architecture, which we describe in the following section. Our research focuses on Nvidia GPUs and their Compute Unified Device Architecture (CUDA) execution model [22,23].

2.1 The SIMT architecture

The SIMT architecture describes a multiprocessor (a common GPU has several multiprocessors, each consisting of multiple thread processors) that can create, manage, schedule and execute threads in groups. A group of N parallel threads is called a *warp*. Every thread has its own instruction address counter and register state. Individual threads in the same warp start together at the same program address, but can execute and branch independently.

A warp executes one common instruction at a time, so full efficiency is realized when all N threads of a warp agree on their execution path. If a thread in a warp diverges via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path. When all paths are completed, the threads converge back to the original execution path. Branch divergence occurs only within a warp. Different warps on different multiprocessors execute independently regardless of whether they are executing common or disjoint code paths.

To maximize utilization, a GPU relies thus on *thread-level parallelism*. Utilization is therefore directly linked to the number of warps residing on a multiprocessor. At every instruction issue time, a warp scheduler selects a warp that is ready to execute its next instruction and issues the instruction to the active threads of the warp. The number of clock cycles it takes for a warp to be ready to execute its next instruction is called *latency* (which actually depends on the number of clock cycles it takes to issue a memory request). Full utilization is then achieved when the warp scheduler always has an instruction to issue for some warp at every clock cycle during that latency period. This is called *latency*

hiding. The number of instructions required to hide a latency of L clock cycles depends on the throughput of these instructions. For example, if we assume that a multiprocessor issues one instruction per warp over 4 clock cycles and maximum throughput for all instructions is achieved, then the number of instructions to hide the latency should be $L/4$.

The SIMT architecture is related to Single Instruction Multiple Data (SIMD) vector organizations in the sense that a single instruction controls multiple processing elements. The major difference between the SIMD and SIMT architectures is the fact that SIMD vector organizations expose the width (and thus the number of threads that can run concurrently on one processor) to the software. SIMT instructions on the other hand specify the execution and branching behavior of a single thread. This enables programmers to write *thread-level* parallel code for independent threads as well as *data-parallel* code for coordinated threads. Actually, the programmer can even ignore the SIMT behavior; however, substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge.

2.2 The CUDA execution model

The mapping between the API and the hardware is an important aspect of the CUDA architecture. A *kernel* is defined as a function that is executed by a *thread*, which are identified by their unique thread id. Every *thread processor* on the device executes the kernel via the SIMT principle. Multiple threads are part of a *thread block*, which is executed by one *multiprocessor*. This multiprocessor contains multiple thread processors. Instructions are issued in warps and all thread processors of one streaming multiprocessor are always issued the same instruction. Several concurrent thread blocks can reside on one multiprocessor, limited by multiprocessor resources such as shared memory and number of available registers. In addition, physical limits of the GPU hardware apply. For the GPU platform used in this research, the maximum number of thread blocks and the maximum number of threads that can reside on one multiprocessor are 8 and 1024 respectively. Threads within a thread block can cooperate and synchronize, while this does not hold for threads in different blocks. Multiple thread blocks make up a *grid*. Every grid contains a predefined number of thread blocks and a kernel is launched on the *device* as a grid of thread blocks. An overview of the CUDA execution model is shown in Figure 1.

The number of threads in a thread block and the number of thread blocks in a grid can be determined at compile time or runtime. This allows programs to transparently scale to different GPUs and hardware configurations. The hardware is free to schedule thread blocks on any multiprocessor as long as it fits in the warp size (which is 32 for the tested platform). When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps that get scheduled by a warp scheduler for execution. The way a block is partitioned into warps is always the same. Each warp contains threads of consecutive, increasing thread id's with the first warp containing thread 0. The configuration of

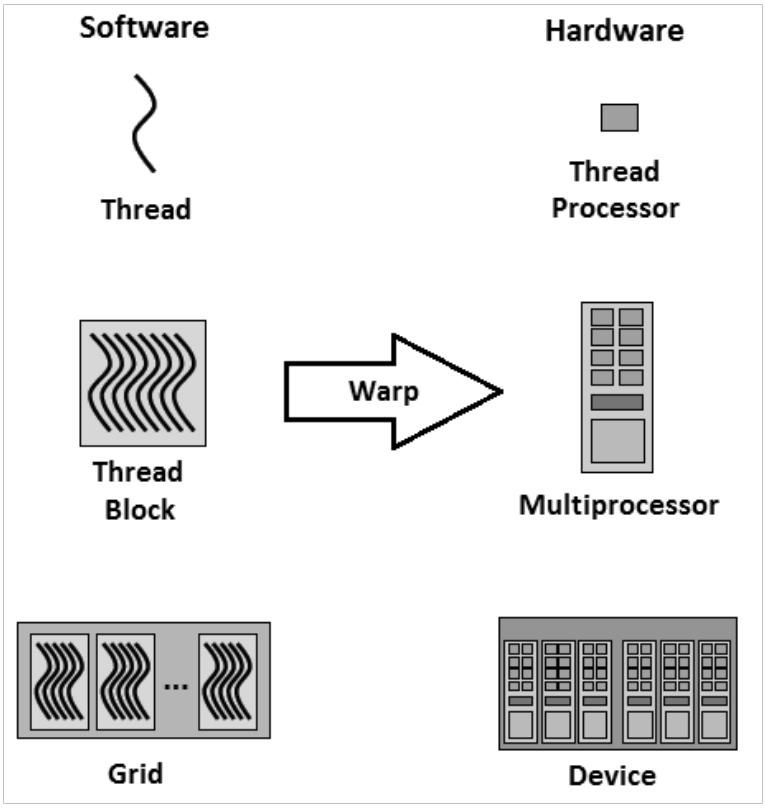


Fig. 1. The CUDA execution model (after [23])

a kernel (in number of grids, thread blocks and threads) significantly influences the execution speed. We will discuss optimal kernel configuration in Section 4.

In addition to the execution model, the memory layout is important for optimization strategies too. CUDA contains five different classes of physical memory. The properties of the types of memory that are relevant for our research are shown in Table 1. Since *local* memory physically resides on the *global* memory,

Memory	Location chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	No	R/W	1 thread	Thread
Shared	On	n/a	R/W	1 Block	Block
Global	Off	No	R/W	All threads	Host
Constant	Off	Yes	R	All threads	Host
Texture	Off	Yes	R	All threads	Host

Table 1. Features of Nvidia's GPU device memory (after [22])

we can use the following types:

- *Global memory:* Global memory physically resides in the device memory (RAM) and therefore has a latency of 400 to 600 clock cycles per access. While this latency is relatively high, it can be hidden by the warp scheduler (although not completely).
- *Registers:* Every multiprocessor has a set of 32-bit registers that are shared by all threads assigned to that multiprocessor. Basically, every access to a register by the thread processor is immediate, which means that it takes zero extra clock cycles per instruction. Delays may occur by read-after-write dependencies and register bank conflicts.
- *Shared Memory:* Every multiprocessor has its own shared memory that is shared by all threads assigned to that multiprocessor. Because shared memory resides on-chip, it is much faster than global memory. In fact, uncached shared memory latency is roughly 100 times lower than global memory latency. Shared memory is not initialized automatically and should therefore be assigned before compile time. Because every thread in a block can access the shared memory and the kernel configuration is not always known at compile time, the programmer has to align the memory and make sure that every thread accesses its own piece of shared memory.

3 Parallelization of MD5-crypt

MD5-crypt is a password hashing scheme that uses the MD5 cryptographic hash function [24] to securely store a user's password. The algorithm applies the MD5 function 1002 times, while concatenating the user's password, a random salt and the result of the previous round in a pseudo-random way (See Appendix Figure 7 for a schematic overview of the algorithm and its pseudo-code). Since the output of the last round serves as input for the next round, the algorithm depletes the parallelization of the 1002 rounds. However, an exhaustive search attack to find a matching password for a given hash can be parallelized easily. In fact, exhaustive searches are *embarrassingly parallel* (as stated in [25]), since the parallel processing units of the underlying hardware do not have to interact or cooperate with each other. Every processing unit tries a set of possibilities and compares them with a target. The only cooperation that is needed, is the division of the search space and a general stop message that terminates the search when one processing unit has found a match. If we map this approach to MD5-crypt, every processing unit needs to calculate the MD5-crypt output hashes of all the candidate passwords given in the input set and match them with a target hash.

To estimate the performance of MD5-crypt on a GPU architecture, we define a simple theoretic model that is based on the number of arithmetic instructions needed to complete one round of the password hashing scheme. Since it is not very hard to estimate the instruction throughput of the GPU hardware platform, this model can be used to compare the performance of the hashing scheme on different architectures and determines the maximum speedup that can be achieved. To define this model, only arithmetic and logic operations are taken into account.

To determine the number of arithmetic instructions needed to complete one round of MD5-crypt, it is necessary to determine the number of instructions for MD5 first. This hash function consists of 64 rounds (where $0 \leq t < 64$) in which the following function is called [26]:

$$Q_{t+1} = Q_t + ((Q_{t-3} + f_t(Q_t, Q_{t-1}, Q_{t-2}) + W_t + AC_t) \\ <<< RC_t) \text{ for } 0 \leq t < 64.$$

In the equation, AC_t is an addition constant and RC_t is a rotation constant. The 512-bit input block is partitioned into sixteen consecutive 32-bit words $m_0, \ldots, m_{15}$ and expanded to 64 words $(W_t)_{t=0}^{63}$ for each step using the following relations:

$$W_t = \begin{cases} m_t & \text{for } 0 \leq t < 16, \\ m_{(1+5t) mod 16} & \text{for } 16 \leq t < 32, \\ m_{(5+3t) mod 16} & \text{for } 32 \leq t < 48, \\ m_{(7t) mod 16} & \text{for } 48 \leq t < 64. \end{cases}$$

Furthermore, f_t is a non-linear function (depending on the round number t) which is defined as:

$$f_t = \begin{cases} F(X,Y,Z) = (X \wedge Y) \oplus (\overline{X} \wedge Z) & \text{for } 0 \leq t < 16, \\ G(X,Y,Z) = (Z \wedge X) \oplus (\overline{Z} \wedge Y) & \text{for } 16 \leq t < 32, \\ H(X,Y,Z) = X \oplus Y \oplus Z & \text{for } 32 \leq t < 48, \\ I(X,Y,Z) = Y \oplus (X \vee \overline{Z}) & \text{for } 48 \leq t < 64. \end{cases}$$

In every round Q requires 3 additions, one cyclic rotation and applies one subfunction. The cyclic rotation requires 2 shifts and 1 addition. Table 2 shows the total number of operations for the subfunctions.

Subfunction	Logic Operations	16 applications of Q
F	4	176
G	4	176
H	2	144
I	3	160
Total after 64 rounds		656

Table 2. Instruction count of the elementary MD5 functions.

Every subfunction is called 16 times, and together with the arithmetic operations of Q and the cyclic rotation, the total number of native arithmetic operations for one application of the MD5-compression function will be 656 instructions. Since MD5-crypt basically applies the MD5-compression function 1002 times, it means that one application of MD5-crypt costs 657312 native arithmetic instructions. It is now possible to calculate performance estimates based on the specifications of the GPU. For a Nvidia GeForce GTX 295 with

a clock speed of 1242 MHz and 480 thread processors, the theoretical estimate of the performance is $\frac{1242*480}{657312} * 10^6 \approx 900.000$ MD5-crypt hashes per second. Note that this calculation only takes the native arithmetic instructions into account, not the memory accesses. Since MD5-crypt needs numerous calls to memory in order to generate the inputs (i.e. the concatenation of the password, salt and result of the previous round) and memory accesses are rather slow, the actual performance depends on the memory organization of the algorithm. Therefore, MD5-crypt is a memory intensive algorithm.

4 Optimization strategies

There are several optimization strategies for GPUs, but since it is the memory that determines the bottleneck in the execution speed of the MD5-crypt algorithm, we will only present the most significant memory optimization strategy and one optimization strategy for determining the correct execution configuration settings (such as threads per block and gridsize). Amongst memory and execution configuration optimizations, we also used instruction, control flow and specific algorithm optimizations.

4.1 Shared memory optimization

Although the warp scheduler uses *latency hiding* to cover the global memory latency, the access time of shared memory is almost as fast as register access time and use of shared memory is therefore preferred over global memory. However, all virtual threads (at least a thread block) that run on one multiprocessor have to share the available shared memory and so concurrent threads can access other thread's memory addresses. Therefore, the developer has to manage the memory accesses himself.

Individual threads can access their shared memory addresses concurrently with the use of equally sized 32-bit wide memory modules, called *banks*. Because banks can be accessed simultaneously, any memory load or store of n addresses that spans n distinct memory banks can be serviced simultaneously, yielding an effective bandwidth that is n times as high as the bandwidth of a single bank. On our test device, the shared memory has 16 banks that are organized in such a way that successive 32-bit words are assigned to successive banks, i.e. interleaved. A memory request for a complete warp is split into two memory requests, one for each half-warp, that are issued independently. Each bank has a bandwidth of 32 bits per clock cycle and since there are 16 banks on a device and the warp size is 32, accessing all the banks for the threads in a full warp will take two clock cycles. *Bank conflicts* arise when multiple threads from the same half-warp access the same bank. The warp is then serialized, and every thread that accesses that bank is then executed sequentially.

To minimize bank conflicts, it is important to understand how memory addresses map to memory banks and how memory requests could be optimally scheduled in a specific implementation. For example, consider the following listing:

```
__shared__ unsigned int data[THREADS_PER_BLOCK][16];
           unsigned int *buffer = data[threadId];
```

Above, `shared[THREADS_PER_BLOCK][16]` is a matrix of integers in the shared memory where every thread can access its corresponding row, calculated by its `threadId`.

No bank conflicts can arise between threads that belong to the first- and second half warp respectively. The hardware splits a memory request that has bank conflicts into as many separate conflict-free requests as necessary, decreasing the effective bandwidth by a factor equal to the number of separate memory requests. In our implementation, every thread needs 16 32-bit words to store the input for the MD5-crypt function. If we store these arrays successively, the modular arithmetic of the device will calculate the corresponding bank automatically. However, this causes a 16-way bank conflict (all 16 threads of a half warp access the first bank, which will lead to 16 serialized requests): thread 0 accesses address 0 which is mapped to bank 0 ($0 \equiv 0 \mod 16$), thread 1 accesses address 16 which mapped to bank 0 ($16 \equiv 0 \mod 16$),..., thread 15 accesses address 240 which is also mapped to bank 0 ($240 \equiv 0 \mod 16$) (see Figure 2). We used *strided* access to solve this bank conflict: every thread now uses 17 words to store the input and leaves one word unused. The modular arithmetic of the device will now calculate the correct corresponding banks for all the threads in a half-warp, as shown in Figure 3 (which is an example with 4 banks). However, this solution does increase the total amount of shared memory needed to run the kernel and shared memory is sparse. As a consequence, less threads per block can be configured to run concurrently on a multiprocessor, which will result in a small loss in performance.

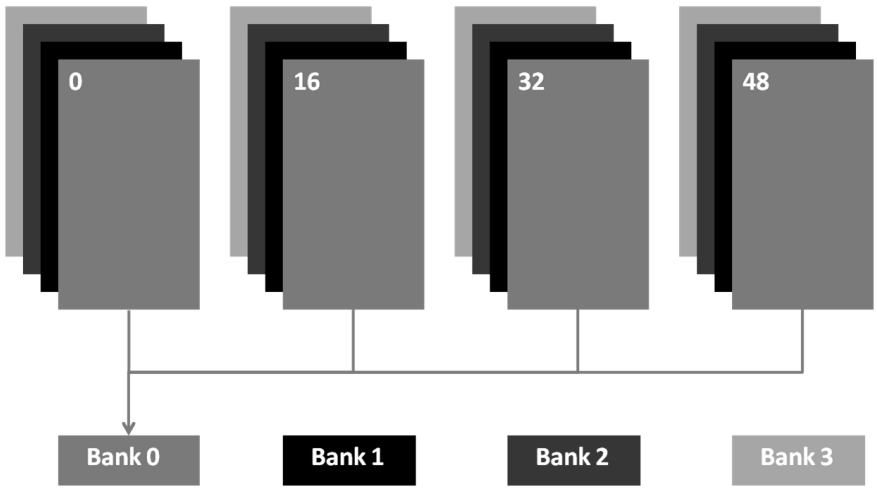


Fig. 2. Serialized access: bank conflicts arise when multiple threads from the same half-warp access the same bank.

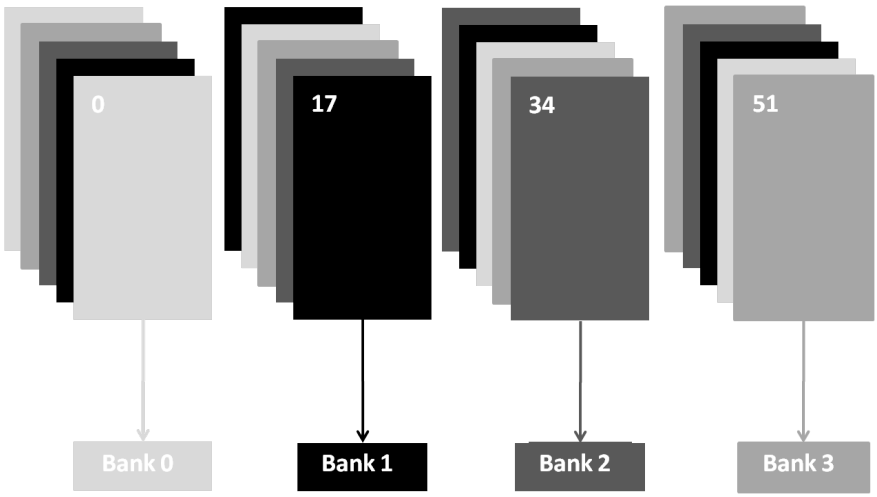


Fig. 3. A stride can solve bank conflicts at the cost of increased shared memory usage.

4.2 Execution configuration optimizations

One of the major keys to optimal performance is to keep the hardware as busy as possible. This implies that the workload should be equally shared between all the multiprocessors of a device. If the work is poorly balanced across the multiprocessors, they will deliver suboptimal performance. It is therefore important to optimize the execution configuration for a given kernel. The key concept that helps to achieve optimal performance is *occupancy*.

Occupancy is specified as the metric that determines how effectively the hardware is kept busy by looking at the active warps on a multiprocessor. This metric originates from the fact that thread instructions are executed sequentially and therefore the only way to hide latencies and keep the hardware busy when the current warp is paused or waiting for input, is to execute other warps that are available on the multiprocessor. Occupancy is therefore defined as $\frac{W_a}{W_{max}}$, where W_a is the number of active warps per multiprocessor and W_{max} is the maximum number of possible active warps per multiprocessor (which is 32 for our test device). Occupancy is kernel (and thus application) dependent, which means that some kernels achieve higher performance with lower occupancy. However, low occupancy always interferes with the ability to hide memory latency, which results in a decrease of performance. On the other hand, higher occupancy does not always implies higher performance, but it may help to cover the latencies and achieve a better distribution of the workload.

5 Experimental evaluation

To measure the performance of our implementation and to compare it against implementations on other hardware platforms, we use the *password hashes per second* metric. This enables us to estimate the time it takes to crack real pass-

words. Let N be the size of the search space, then this metric is measured by counting the number of clock cycles it takes for all N threads to:

- Generate a candidate password.
- Calculate a MD5-crypt hash.
- Compare the hash against the target hash.

To compute the execution time (in seconds s), the total number of clock cycles C is divided by the number of clock cycles that a given device can execute per second. Finally, the number of hashes per second is calculated as $\frac{N}{s}$. Our implementation supports multiple GPUs, but we have ran our tests on one Nvidia GeForce 295 GTX (which has in fact 2 GPUs on its board). To compare the implementation with CPU hardware, we used an equally priced Intel i7 920 processor.

5.1 Algorithm optimizations

We describe and measure multiple optimizations for increasing the execution speed of the computations on a GPU. After we have defined a baseline (point 1), four optimizations are compared to this baseline (point 2 to 5). Then, the optimizations are combined (point 6 and 7).

1. *Baseline.* This is the implementation which does not use any optimization at all. Moreover, all the variables are stored in the local (i.e. global) memory and the allocation of the memory is done automatically.
2. *Constant memory.* The variables that do not change during the execution, such as target hash, salt and character set, are now stored in the constant memory. The other variables are still stored in local memory.
3. *Shared memory (bank conflicts).* The following variables are stored in shared memory, since they are accessed often during execution time:
   - Input buffer (`int data[THREADS_PER_BLOCK][16]`).
   - Resulting hash (`int final[THREADS_PER_BLOCK][4]`).
   - Password (`unsigned char password[THREADS_PER_BLOCK][8]`).

   However, bank conflicts and thus warp serializes occur, which slow down the execution. The non-changing variables are still stored in the local memory.
4. *Shared memory (no bank conflicts).* The allocation of the shared memory is now done is such a way that no bank conflicts occur. With a strided access pattern every thread can access its own bank such that warp serializes are kept to a minimum, which increases the performance (at the cost of a little increase in total shared memory used).
5. *Optimized MD5.* The MD5-compression function is statically optimized. Depending on the password length, only the necessary calculations are performed (e.g. for passwords with a length smaller than 9 characters). The size of the input buffer can then be decreased from 16 to 10 (`int data[THREADS_PER_BLOCK][10]`).
6. *Shared and constant memory (no bank conflicts).* Now both the changing (e.g. input buffer) and non-changing variables (e.g. salt) are kept in shared and constant memory respectively.

7. *Optimized MD5 with shared and constant memory.* In addition to point 6, the optimized version of the MD5-compression function is used as well.

During all measurements, the configuration parameters are kept equal. The number of thread blocks (gridsize) is set to '1200' (since this is a multiple of '60', which is the number of multiprocessors on a Nvidia GTX 295) and the number of threads per block (blocksize) is set to '160' (since this is the maximum number for all optimizations to have enough shared memory available). In Figure 4 the performance increase per optimization is shown in black, combined optimizations are shown in gray. The figure shows that the shared memory (without

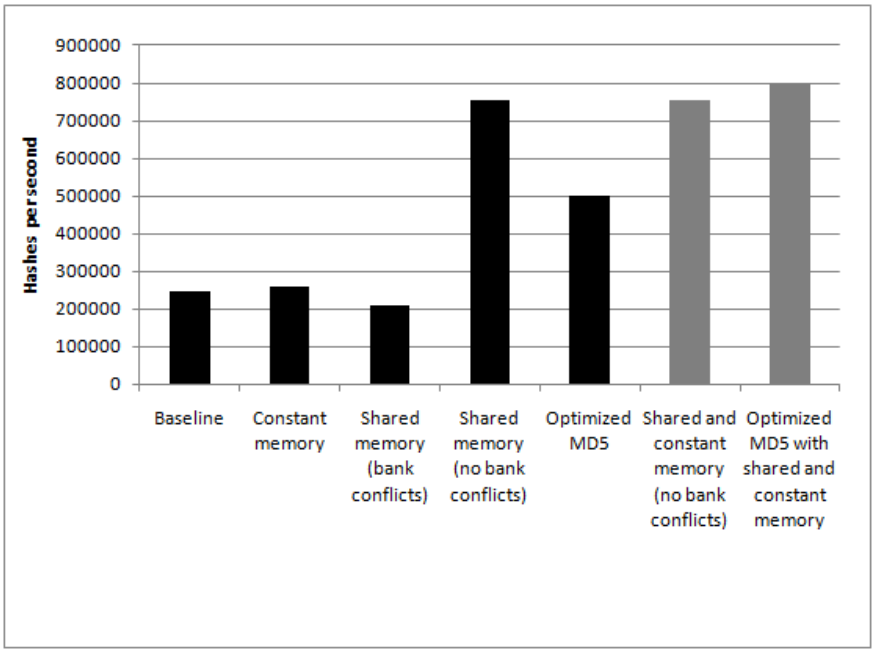


Fig. 4. Performance increase per optimization, executed on a Nvidia GeForce GTX 295.

bank conflicts) and MD5-compression function optimization achieve speedups of 3 and 2 times the baseline respectively. However, when we combine both optimizations, the speedup is only a little over 3 times compared to the baseline. This can be explained by the fact that the MD5-compression function optimization reduces the number of memory calls. While this optimization achieves a significant performance increase when all variables are stored in local memory, only little performance increase is gained when all variables are stored in shared memory (since this type of memory has low latency). Furthermore, the figure also shows that there is very little performance increase when the non-changing variables are stored in constant memory and the other variables are stored in shared memory compared to storing all the variables in shared memory (point 6 and 4 respectively). This can be explained by the fact that the compiler stores frequently used variables in the constant memory by itself and keeps a copy in

local memory too (when a constant cache miss arises). Finally, the figure shows that storing variables in the shared memory while not solving the bank conflicts even degrades the performance compared to the baseline.

5.2 Configuration optimizations

As stated in Section 4.2, occupancy is defined as $\frac{W_a}{W_{max}}$. The influence of occupancy on our most optimal implementation ("Optimized MD5 with shared and constant memory") is shown in Figure 5. The number of threads per block determines W_a and thus the final performance. Due to physical limits of the available

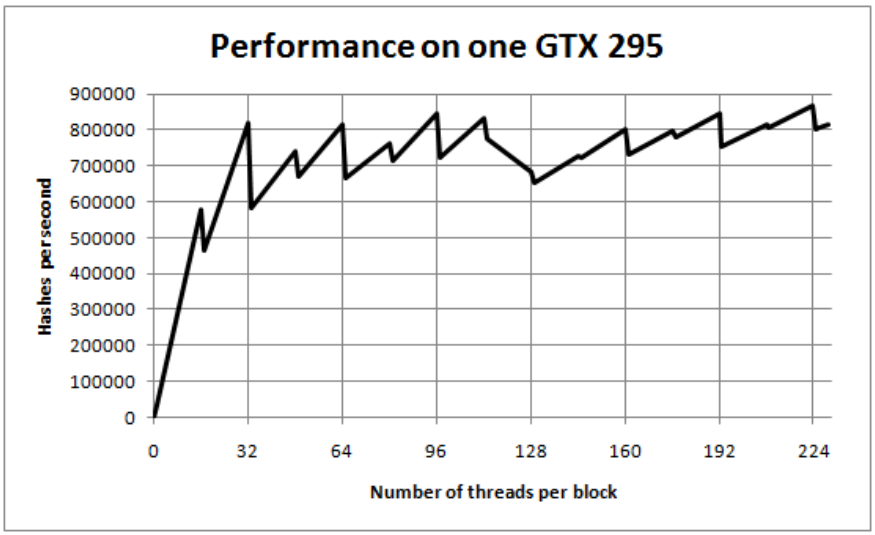


Fig. 5. Influence of the number threads per block on the performance of our most optimal implementation, executed on a Nvidia GeForce GTX 295. Note that "hashes per second" refers to password hashes, i.e. MD5-crypt operations.

shared memory (16384 bytes per block), the maximum number of threads per block for our implementation is smaller than the physical limit. For example, if we store the password (8 bytes + 1 byte for strided access), input buffer (4 * (10 + 1) bytes) and resulting hash (4 * (4 + 1) bytes) in shared memory, the maximum number of threads per block for our implementation is $\lfloor \frac{16384}{73} \rfloor = 224$. Therefore W_a (active warps per multiprocessor) is 7, which results in an occupancy of $\frac{7}{32} \approx 22\%$.

The figure shows a characteristic behavior with stair-like graphs. Multiples of the warp size (32) and half warp size (16) result in more optimal performance, since this ensures that no incomplete warps are executed. In addition, the number of threads per block should be higher then $8 \times 24 = 192$, since the latency of register read-after-write dependencies is approximately 24 cycles and a multiprocessor has 8 thread processors.

5.3 Comparison against CPU implementations

We compared our implementation to the fastest MD5-crypt CPU implementation known to us, which is incorporated in the password cracker 'John the Ripper' [27]. However, John the Ripper does not have support for parallel CPU implementations of MD5-crypt. In order to fully use all four cores of the Intel i7 920, we used the OpenMP library to parallelize our CPU implementation[3]. Figure 6 shows the performance of MD5-crypt implementations on equally priced GPU and CPU hardware. Our GPU implementation (about 880 000 hashes per second) achieves a speedup of 28 times over our CPU implementation (about 32 000 hashes per second) and a speedup of 104 times over the John the Ripper CPU implementation (about 8500 hashes per second). With this performance increase and using a few high-end GPUs, 'complex' passwords with a length of 8 characters can be searched effectively.

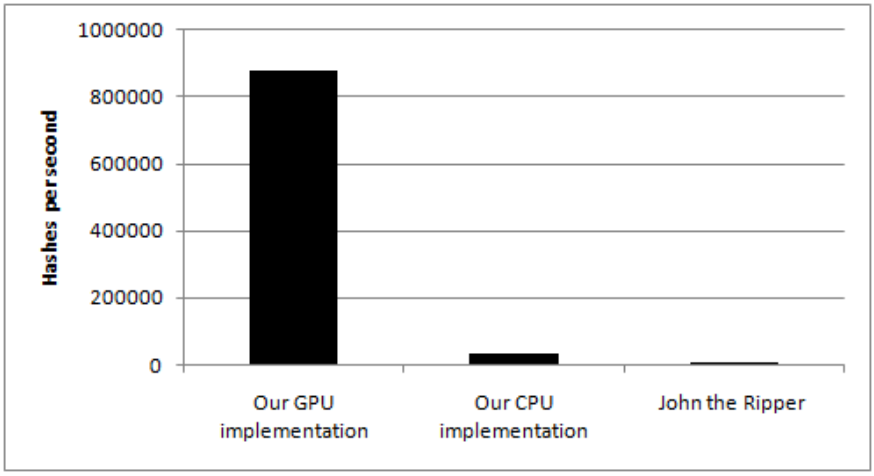


Fig. 6. Performance comparison of different implementations on different architectures.

5.4 Speedup comparison against other cryptographic implementations

Table 3 shows the different speedups GPUs can achieve over CPUs. Note that not all experiments were carried out on equally priced hardware, which influences the final outcome. Moreover, process variations in hardware and constant improvements of GPU and CPU chips, CPU implementations, and CPU technologies (e.g. the use of the SSE instruction set) will lead to changes in the GPU / CPU run-time ratio. However, the table proves that GPUs can be efficiently used to perform cryptographic operations, or at least can act as a cryptographic 'co-processor'. The major difference between the speedup for hashing and other

[3] Please note that optimizing the CPU implementation was not in scope of this research, therefore a port of our GPU implementation is used.

Work	Cryptographic type	Algorithm	Speed up GPU over CPU
[18,20]	Asymmetric	ECC	4-5
[9]	Symmetric	AES	5-20
[28]	Symmetric	AES	4-10
[17]	Asymmetric	RSA	4
This work	Hashing	MD5-crypt	25-30

Table 3. Speed up GPU over CPU for different cryptographic applications.

cryptographic types comes from the fact that hash functions are solely build out of native arithmetic instructions (such as shifts, additions and logical operators) that have high throughput on a typical GPU. RSA and ECC, in contrast, are build out of modular multiplications and other modular arithmetic operations, which have lower throughput than native arithmetic instructions. In addition, password hashing allows for a maximal parallelization whereas, for example, with AES not all modes of encryption can be used for parallelization (e.g. Cipher-block Chaining or Cipher Feedback mode).

5.5 Consequences for practical use of password hashing schemes

We have shown that in the field of exhaustive searches on the password hashing scheme MD5-crypt, GPU's can significantly speed up the cracking process. To determine if this speedup is significant enough to attack systems that use such password hashing schemes, we have to see how our implementation performs in terms of cracking time. To see whether the passwords are crackable in a feasible amount of time, we defined four password 'classes':

1. Passwords consisting of only lowercase ASCII characters (26 in total)
2. Passwords consisting of lowercase and numeric ASCII characters (36 in total)
3. Passwords consisting of lowercase, numeric and uppercase ASCII characters (62 in total)
4. Passwords consisting of lowercase, numeric, uppercase and special ASCII characters (94 in total)

Table 5.5 shows the search times for four password classes and some password lengths, given our maximum performance on a Nvidia GeForce GTX 295 (880 000 hashes per second).

To decrease the percentage of passwords that can be recovered with exhaustive searches on prevalent graphics hardware, the following methods could be used:

- Increase the entropy of the user password by enforcing a password policy.
- Increase the complexity of the password hashing scheme in such a way that one user is able to hash his password, but exhaustive searches take too much

Length	26 characters	36 characters	62 characters	94 characters
4	0,5 Seconds	2 Seconds	16 Seconds	2 Minutes
5	13 Seconds	1 Minute	17 Minutes	2 Hours
6	5 Minutes	41 Minutes	18 Hours	10 Days
7	2 Hours	1 Days	46 Days	3 Years
8	2 Days	37 Days	8 Years	264 Years
9	71 Days	4 Years	488 Years	20647 Years
10	5 Years	132 Years	30243 Years	2480775 Years

Table 4. Exhaustive search times of some password lengths on a Nvidia GeForce GTX 295 with a performance of 880 000 hashes per second.

time to complete for multiple candidate passwords. This can be achieved by increasing the number of calls to the underlying hash function (e.g. by increasing the number of iterations in the key-stretching technique).

The first method increases the search space exponentially while the second method only linearly increases the time needed to iterate over the search space. Therefore, it is better to enforce users to increase the entropy in their passwords. Even so, if all users would pick passwords with high entropy, the use of password hashing schemes would be superfluous.

If we want to increase the complexity of password hashing schemes in such a that they may be able withstand exhaustive search attacks on current hardware, we could use the following rule of thumb: *One application of the password hashing scheme should not execute in less than 10 milliseconds on specialized hardware*[4]. If we apply this rule to MD5-crypt, the number of iterations should be increased with four orders of magnitude (from 1000 to 10 000 000 iterations).

6 Conclusions

In this work we have shown that GPUs can be used for launching exhaustive search attacks on password hashing schemes that rely on the key strength of user-chosen passwords. The possibility to parallelize the attacks enabled us to optimize the implementation of one password hashing scheme, MD5-crypt. Our implementation achieves a speedup of two orders of magnitude over the best known existing CPU implementation. This performance approaches the theoretical speed limit on a CUDA enabled GPU. Moreover, with this performance increase, 'complex' passwords with a length of 8 characters are now becoming feasible to crack.

[4] The designer of MD5-crypt used this rule of thumb to determine the number of iterations in his algorithm.

7 Acknowledgements

We would like to thank Mr. P.H. Kamp for his comments and discussions on the design of his MD5-crypt algorithm, which provided us with more insight about the possible optimizations.

References

1. Abadi, M., Lomas, T.M.A., Needham, R.: Strengthening passwords. SRC Technical Note **33** (1997)
2. Kelsey, J., Schneier, B., Hall, C., Wagner, D.: Secure applications of low-entropy keys. Information Security (1998) 121–134
3. Kaliski, B.: RFC 2898: PKCS# 5: Password-Based Cryptography Specification Version 2.0. IETF, Sep (2000)
4. Mentens, N., Batina, L., Verbauwhede, I., Preneel, B.: Time-Memory Trade-Off Attack on FPGA Platforms: UNIX Password Cracking. Reconfigurable Computing: Architectures and Applications (2006) 323–334
5. Thompson, C.J., Hahn, S., Oskin, M.: Using modern graphics architectures for general-purpose computing: A framework and analysis. In: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture, IEEE Computer Society Press (2002) 306–317
6. Cook, D.L., Ioannidis, J., Keromytis, A.D., Luck, J.: CryptoGraphics: Secret key cryptography using graphics cards. Topics in Cryptology–CT-RSA 2005 (2005) 334–350
7. Yang, J., Goodman, J.: Symmetric key cryptography on modern graphics hardware. Advances in Cryptology–ASIACRYPT 2007 (2008) 249–264
8. Harrison, O., Waldron, J.: AES encryption implementation and analysis on commodity graphics processing units. Cryptographic Hardware and Embedded Systems-CHES 2007 (2007) 209–226
9. Manavski, S.A.: CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In: Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on, IEEE (2008) 65–68
10. Di Biagio, A., Barenghi, A., Agosta, G., Pelosi, G.: Design of a parallel AES for graphics hardware using the CUDA framework. In: Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, IEEE (2009) 1–8
11. Bos, J.W., Osvik, D.A., Stefan, D.: Fast Implementations of AES on Various Platforms. Technical report, Cryptology ePrint Archive, Report 2009/501, November 2009. http://eprint. iacr. org (2009)
12. Li, C., Wu, H., Chen, S., Li, X., Guo, D.: Efficient implementation for MD5-RC4 encryption using GPU with CUDA. In: Anti-counterfeiting, Security, and Identification in Communication, 2009. ASID 2009. 3rd International Conference on, IEEE (2009) 167–170
13. Hu, G., Ma, J., Huang, B.: High Throughput Implementation of MD5 Algorithm on GPU. In: Ubiquitous Information Technologies & Applications, 2009. ICUT'09. Proceedings of the 4th International Conference on, IEEE (2010) 1–5
14. Mukherjee, R., Rehman, M.S., Kothapalli, K., Narayanan, P., Srinathan, K.: (Presenting new speed records and constant time encryption on the GPU)
15. Schober, M.: (Efficient Password and Key recovery using Graphic Cards)

16. Szerwinski, R., Güneysu, T.: Exploiting the power of GPUs for asymmetric cryptography. Cryptographic Hardware and Embedded Systems–CHES 2008 (2008) 79–99
17. Harrison, O., Waldron, J.: Efficient acceleration of asymmetric cryptography on graphics hardware. In Preneel, B., ed.: AFRICACRYPT. Volume 5580 of Lecture Notes in Computer Science., Springer (2009) 350–367
18. Bernstein, D.J., Chen, H.C., Chen, M.S., Cheng, C.M., Hsiao, C.H., Lange, T., Lin, Z.C., Yang, B.Y.: The billion-mulmod-per-second PC. SHARCS Workshop (2009)
19. Bernstein, D.J., Chen, T.R., Cheng, C.M., Lange, T., Yang, B.Y.: ECM on graphics cards. Advances in Cryptology-EUROCRYPT 2009 **28** (2009)
20. Bernstein, D., Chen, H.C., Cheng, C.M., Lange, T., Niederhagen, R., Schwabe, P., Yang, B.Y.: ECC2K-130 on NVIDIA GPUs. Progress in Cryptology-INDOCRYPT 2010 (2010) 328–346
21. Drepper, U.: Unix crypt using SHA-256 and SHA-512. Technical report, Akkadia (2008)
22. NVIDIA: Compute Unified Device Architecture Programming Guide. Technical report, Nvidia Corporation (2010)
23. NVIDIA: CUDA Best practices guide. Technical report, Nvidia Corporation (2010)
24. Rivest, R.: RFC1321: The MD5 message-digest algorithm. RFC Editor United States (1992)
25. Foster, I.: Designing and building parallel programs: concepts and tools for parallel software engineering. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA (1995)
26. Stevens, M., Sotirov, A., Appelbaum, J., Lenstra, A., Molnar, D., Osvik, D., De Weger, B.: Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. Advances in Cryptology-CRYPTO 2009 (2009) 55–69
27. Designer, S.: John the Ripper password cracker (2011)
28. Harrison, O., Waldron, J.: Practical symmetric key cryptography on modern graphics hardware. In: Proceedings of the 17th conference on Security symposium, USENIX Association (2008) 195–209

Appendix: Design of MD5-crypt

MD5-crypt is the standard password hashing scheme for FreeBSD operating systems, supported by most Linux distributions (in the GNU C library) and implemented in Cisco routers. It was designed by Poul-Henning Kamp in 1994. Figure 7 shows the schematic overview of the MD5-crypt implementation. The figure only shows the most relevant parts and abstracts from initializations. Basically, MD5-crypt applies the MD5-compression function 1002 times:

- In the first application, the concatenation of the password, salt and password again is hashed.
- In the second application, the concatenation of the password, magic string (‘1’), salt and the result of the first application is hashed.
- In the next thousand applications, the concatenation of the password, salt and result of the previous application is hashed based on the round number n. The pseudo code of this application is shown in Algorithm 1.

Algorithm 1 MD5-crypt pseudo code, main loop.

```
1: for i = 0, i < 1000, i++ do
2:     msglen = 0
3:     for j = 0, j < 16, j++ do
4:         buffer[j] = 0
5:     end for
6:     if i & 1 then                                  ▷ Case i is odd
7:         set(buffer,password)
8:         msglen += len(password)
9:     else                                           ▷ Case i is even
10:        set(buffer,final)                          ▷ Add the result of last round
11:        msglen += 16
12:    end if
13:    if i % 3 then                                  ▷ Case i ∤ 3
14:        set(buffer,salt)
15:        msglen += len(salt)
16:    end if
17:    if i % 7 then                                  ▷ Case i ∤ 7
18:        set(buffer,password)
19:        msglen += len(password)
20:    end if
21:    if i & 1 then                                  ▷ Case i is odd
22:        set(buffer,final)                          ▷ Add the result of last round
23:        msglen += 16
24:    else                                           ▷ Case i is even
25:        set(buffer,password)
26:        msglen += len(password)
27:    end if
28:    set(buffer,0x80)                               ▷ Add the binary 1
29:    set(buffer,msglen << 3)                        ▷ Add the message bit length
30:    MD5Compress(final,buffer)                      ▷ Call MD5 and store result in final
31: end for
32: if target == final then                           ▷ Match with target
33:    setFlag(inputPassword)
34: end if
```

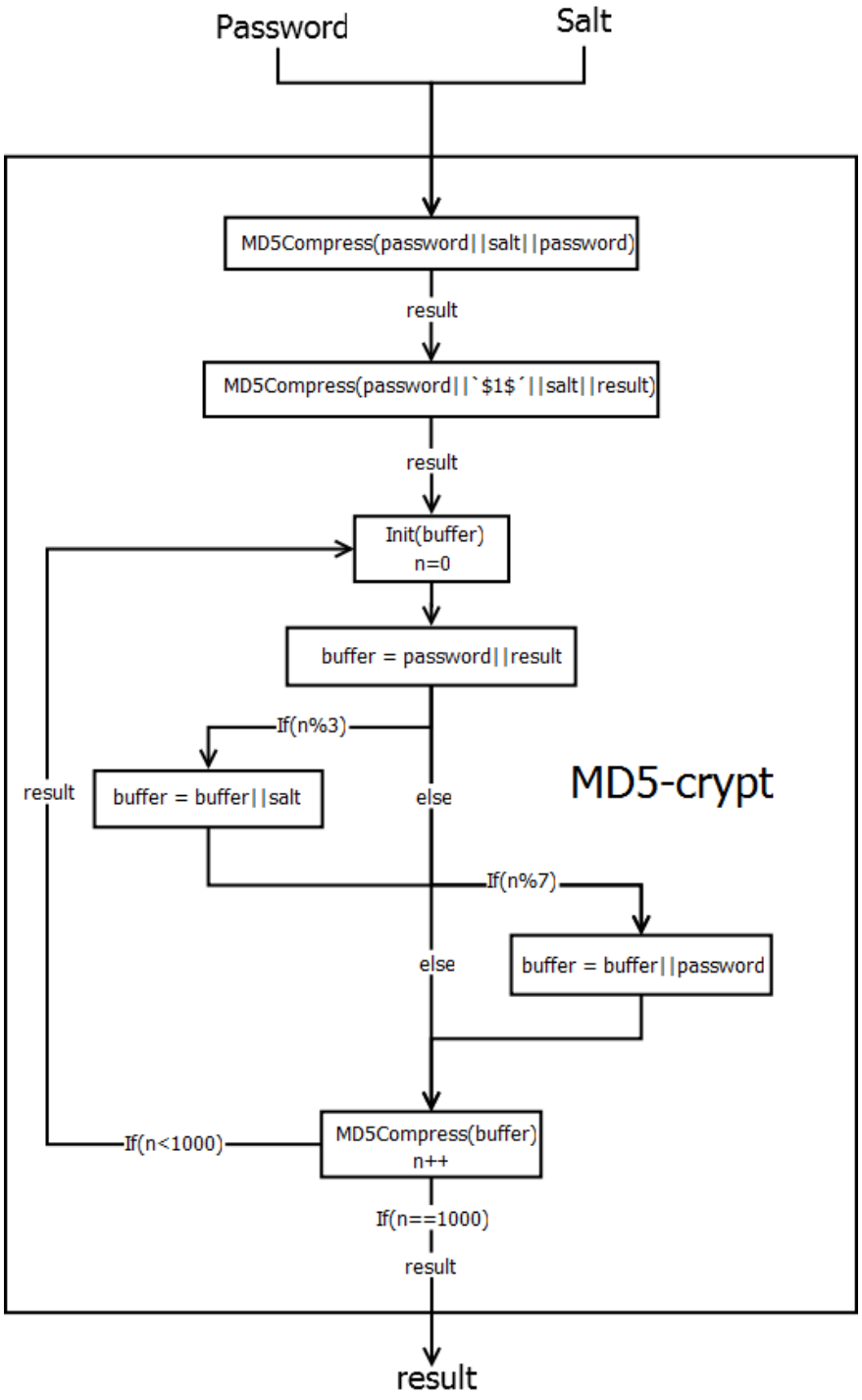


Fig. 7. Schematic overview of MD5-crypt.

Codebreaking with IBM machines in World War II

Stephen Budiansky
spb@budiansky.com
http://budiansky.com

Abstract

Many important enemy code systems were broken by the US Army and Navy during World War II with the help of a variety of special-purpose analytic machinery. Among the most important of these were special adaptations of commercial IBM card machines, developed to automate specific time-consuming tasks both in initially solving enemy cryptosystems and then in routinely deciphering intercepted messages. Other innovative cryptanalytic hardware developed during the war used optical, paper-tape, and other storage devices that pushed electro-mechanical computing to its technological limits in this era just prior to the dawn of the digital revolution.

Cryptol: The Language of ~~Cryptography~~ Cryptanalysis

Sally A. Browning and Joe Hurd

Galois, Inc
{sally, joe}@galois.com
www.galois.com
www.cryptol.net

1 Introduction

Cryptol was designed by Galois, Inc. for the NSA as a domain-specific language for specifying cryptographic algorithms, eliminating the need for separate and voluminous natural language documentation. Cryptol is tailored to the unique needs of cryptography and cryptographic implementations. It is fully executable, allowing cryptographers to experiment with their programs incrementally as their designs evolve, with the compiler checking the consistency of data types and array lengths at every stage. These same attributes make Cryptol a good language for expressing cryptanalysis algorithms, providing a platform to explore different approaches and carry out experiments at low cost. In addition, Cryptol provides a refinement methodology to bridge the conceptual gap between specification and low-level implementation, and can generate both hardware and software implementations from high-level specifications, as well as formal models for verification. For example, Cryptol allows engineers and mathematicians to program cryptographic algorithms on FPGAs as if they were writing software, and the Cryptol verification toolset can show functional equivalence between the specification and the implementation at each stage of the tool-chain. In addition, the Cryptol verification toolset can be usefully applied to the reference specification of cryptographic algorithms. Proving desirable high-level properties of a cryptographic algorithm gives assurance of its robustness, while conversely finding counterexamples of desirable properties may inspire approaches to cryptanalysis.

2 Cryptol Case Studies

2.1 Exploring an algorithm

The AIM crypto-engine engineers at General Dynamics C4 Systems use the Cryptol modeling language as part of the development process. Cryp-

tol provides four basic benefits leading to the certification of crypto equipment. First, Cryptol allows the design engineer to rapidly express an algorithm in a common mathematical notation, which is fully executable on the Cryptol interpreter, providing verification that the algorithm is completely understood. Second, the Cryptol notation for the various components of the algorithm are used to annotate the AIM micro sequencer code which provides much greater readability of that extremely dense assembly language. Third, component testing of AIM code, from small snippets through major subroutines is greatly facilitated with Cryptol generated test vectors derived from end-to-end test vectors provided in algorithm source specifications. Finally, Cryptol models directly support the certification effort.

2.2 Produce and refine a family of designs

A team of developers from Rockwell Collins, Inc. and Galois, Inc. has successfully designed, implemented, simulated, integrated, analyzed, and tested a complex embedded Cryptographic Equipment Application (CEA) in less than 3 months. An algorithm core generated from a Cryptol specification for AES-256 running in Electronic Codebook mode demonstrated throughput in excess of 16 Gbps. These high-speed CEA implementations comprise a mixture of software and VHDL, and target a compact new embedded platform designed by Rockwell Collins. Notably, almost no traditional low-level interface code was required in order to implement these high-performance CEAs. In addition, automated formal methods prove that algorithm implementations faithfully implement their high-level specifications.

2.3 Gaining confidence in an implementation

Van der Waerdens theorem states that for any positive integers r and k there exists a positive integer N such that if the integers 1, 2, ..., N are each colored with one of r different colors, then there are at least k integers in arithmetic progression all of the same color. For any r and k, the smallest such N is the van der Waerden number W(r,k). Van de Waerden numbers are difficult to compute. In 2007, Dr. Michal Kouril of the University of Cincinnati established that W(2,6)=1132 (i.e., 1132 is the smallest integer N such that every 2-coloring of 1, 2, ..., N contains a monochromatic arithmetic progression of length 6). The most recent previous result, W(2,5)=178, was discovered some 30 years earlier. Kouril computed W(2,6) using a special SAT-solver and clever techniques to

bound the search and employed FPGAs to speed up the search. In order to convince himself that the FPGA ensemble was doing what he expected, he wrote a Cryptol specification for the algorithm running in the FPGA ensemble, generated formal models for both the Cryptol specification and the VHDL implementation, and verified that the two were equivalent.

2.4 Gaining confidence in a third-party implementation

Skein is a suite of cryptographic hash algorithms targeted at the NIST SHA-3 competition. At its core, Skein uses a tweakable block cipher named Threefish. The unique block iteration (UBI) chaining mode defines the mode of operation by the repeated application of the block cipher function. Galois has developed and published a Cryptol specification for Skein, and verified two independently developed VHDL implementations of Skein against our specification, finding an ambiguity bug in one of them.

2.5 Building a MILS FPGA

The Xilinx Single Chip Cryptographic (SCC) technology enables Multiple Independent Levels of Security (MILS) on a single FPGA. Galois Cryptol Workbench provides a tool flow that puts FPGA implementation into the hands of mainline developers, improving both productivity and assurance, without sacrificing performance. These two technologies fit seamlessly into a single development flow. The combined solution can address high-grade cryptographic application requirements (redundancy, performance, red/black data, and multiple levels of security on a single chip) as well as high assurance development needs (high-level designs, automatic generation of implementation from design, automatically-generated equivalence evidence), and has the potential to significantly reduce the time of costs of developing Type-1 cryptographic applications.

3 Try Cryptol for your Applications

The Cryptol interpreter is freely available at www.cryptol.net. Documentation and evaluation copies of the full Cryptol toolset are also available at that site.

On the strength comparison of ECC and RSA

Masaya Yasuda, Takeshi Shimoyama, Tetsuya Izu, and Jun Kogure

FUJITSU LABORATORIES LTD.
1-1, Kamikodanaka 4-chome, Nakahara-ku, Kawasaki, 211-8588, Japan

Abstract. At present, the RSA cryptosystem is most widely used in public key cryptography. On the other hand, elliptic curve cryptography (ECC) has recently received much attention since smaller ECC key sizes provide the same security level as RSA. Although there are a lot of previous works that analyze the security of ECC and RSA, the comparison of strengths varies depending on analysis. The aim of this paper is once again to compare the security strengths, considering state-of-art of theory and experiments. In this paper, we compare the computing power required to solve the elliptic curve discrete logarithm problem (ECDLP) and the integer factorization problem (IFP), respectively, and estimate the sizes of the problems that provide the same level of security.

1 Introduction

After Rivest, Shamir, and Adleman proposed the RSA cryptosystem in 1977 [36], Koblitz and Miller independently proposed ECC in 1985 [19, 26, 30]. The security of RSA is closely related to the hardness of the IFP, while the security of ECC is closely related to the hardness of the ECDLP. If we can solve the IFP (or the ECDLP), we can break RSA (or ECC), respectively. Currently, subexponential-time algorithm to solve the IFP are known. On the other hand, the best known algorithm to solve the ECDLP has fully exponential running time. This fact ensures that smaller ECC key sizes provide the same security level as RSA. The advantages of smaller key sizes are very important to use devices with limited processing capacity, storage or power supply, like smart cards. Hence ECC can be used more widely in the future, and it is important to compare the security strengths of ECC and RSA in order to embed ECC into information systems. In this paper, we mainly estimate the computing power required to solve the ECDLP and the IFP in a year, respectively. Using special-purpose hardware for the IFP or the ECDLP is a theme of great interest and there are some previous works, such as [18, 21, 40, 41]. However, these platforms and architectures vary, and it is difficult to make an analysis on the cost performance. Hence we would like to leave it as a future work, and in this paper we focus on the hardness of the IFP and the ECDLP from the view point of software implementation.

Although a number of ways to solve the ECDLP are known, Pollard's rho method [34] is the fastest known algorithm for solving the ECDLP except special cases such as the supersingular cases and the anomalous cases [11, 19, 29, 38, 39, 42]. The rho method works by giving a pseudo-random sequence defined by an iteration function and then detecting a collision in the sequence (see [19] for details of the rho method). The running time of the rho method is determined by the number of iterations before obtaining a collision and the processing performance of iterations. Since the number of iterations heavily depends on iteration functions, we first discuss the choice of iteration functions suitable for solving the ECDLP. Since the rho method is probabilistic, we next estimate the number of iterations required to solve the ECDLP with very high probability based on our experiments of solving

the ECDLP of relatively small parameters (We here consider the success probability of the rho method as 99%). With respect to the processing performance of iterations, we need to evaluate it as strictly as possible. Note that the processing performance of iterations is related with operations on elliptic curves. We implemented operations on elliptic curves by using our library for arbitrary length integers, but our implementation results were far from previously known records (see [2, 4] for example). We then use the previously known records to estimate the processing performance of iterations. Furthermore, we focus on three types in the ECDLP, namely, prime fields, binary fields, and Koblitz curves types.

On the other hand, it is known that the general number field sieve method (GNFS) is the most efficient known algorithm for solving the IFP of large composite integers [27]. CRYPTREC Report 2006 [10] gives the expected computing power required to solve the IFP of N-bit composite integers with $N = 768, 1024, 1592$ and 2048 based on experiments of the GNFS. To estimate the computing power required to solve the IFP, we use the CRYPTREC results and the well-known heuristic complexity of the GNFS.

The outline of this paper is as follows: In Section 2, we review the rho method for solving the ECDLP to fix our notation. In Section 3, we summarize previously known results of the security evaluation of ECC and the strength comparison of ECC and RSA. In Section 4, we discuss on the rho method for solving the ECDLP based on previous and our own experimental results. In Section 5, we estimate the computing power required to solve the ECDLP and the IFP respectively, and calculate the bit sizes of the ECDLP and the IFP that provide the same level of security. Finally in Section 6, we conclude our work.

2 Pollard's rho method for the ECDLP

To fix our notation, we review the rho method for the ECDLP mainly due to [19].

2.1 Basic idea of the rho method

Definition 1 (ECDLP). *Given an elliptic curve E defined over a finite field $\mathbb{F}_q$ with q elements, a point $S \in E(\mathbb{F}_q)$ of prime order n, and a point $T \in \langle S \rangle$, find the integer $k \in [0, n-1]$ with $T = kS$.*

Fix an iteration function $f : \langle S \rangle \to \langle S \rangle$ such that it is easy to compute $X' = f(X)$ and $c', d' \in [0, n-1]$ with $X' = c'S + d'T$ for given $X = cS + dT$. For a starting point $X_0 = c_0 S + d_0 T$ with randomly chosen $c_0, d_0 \in [0, n-1]$, we define a sequence $\{X_i\}_{i \geq 0}$ by $X_{i+1} = f(X_i)$ for $i \geq 0$. It follows from the property of the iteration function f that we can easily compute $c_i, d_i \in [0, n-1]$ with $X_i = c_i S + d_i T$. Since the set $\langle S \rangle$ is finite, the sequence will eventually meet a point that has occurred before, which is called a *collision*. A collision $X_i = X_j$ with $i \neq j$ gives the relation $c_i S + d_i T = c_j S + d_j T$. Since we have $(c_i - c_j)S = (d_j - d_i)T = (d_j - d_i)kS$, we can compute the solution

$$k = (c_i - c_j) \cdot (d_j - d_i)^{-1} \bmod n$$

of the ECDLP if $d_j \not\equiv d_i \bmod n$. This is the basic idea of the rho method for solving the ECDLP (see [19, pp. 157- 158] for details).

Since a collision gives the solution of the ECDLP with very high probability, the number of iterations before obtaining a collision is significant for the running time of the rho method. To solve the ECDLP efficiently, we take an iteration function f with the characteristic of a random function. If f is a random function, the expected number of iterations before obtaining a collision is approximately

$$\sqrt{\pi n/2} \approx 1.2533\sqrt{n}$$

by the birthday paradox.

2.2 Improving the rho method

Parallelized rho method: Van Oorshot and Wiener [45] proposed a variant of the rho method that yields a factor M speed up when M processors are employed. The idea is to allow the sequences $\{X_i^{(j)}\}_{i\geq 0}$ generated by the processors to collide with one another. More precisely, each processor randomly selects its own starting point $X_0^{(j)}$, but all processors use the same iteration function f to compute subsequent points $X_i^{(j)}$.

Collision detection: Floyd's cycle-finding algorithm [25] finds a collision in the sequence generated by a single processor. The following strategy enables efficient finding of a collision in the sequences generated by different processors. An easy testable *distinguishing property* of points is selected. For example, a point may be *distinguished* if the leading t bits of its x-coordinate are zero. Let $0 < \theta < 1$ be the proportion of points in the set $\langle S\rangle$ having this distinguishing property. Whenever a processor encounters a distinguished point, it transmits the point to a central server which store it in a sorted list. When the server receives the same distinguished point for the second time, it computes the desired logarithm and terminates all processors. The expected number of iterations per processor before obtaining a collision is $(\sqrt{\pi n/2})/M$, when M processors are employed. A subsequent distinguished point is expected after $1/\theta$ iterations. Hence the expected number of elliptic curve operations performed by each processor before observing a collision of distinguished points is

$$\frac{1}{M}\sqrt{\frac{\pi n}{2}} + \frac{1}{\theta}.$$

We note that the running time of $1/\theta$ iterations after a collision occurs is negligible for the total running time if we select θ such that $1/\theta$ is small enough compared to the order n of the point S.

Speeding up the rho method using automorphisms: Wiener and Zuccherato [46] and Gallant, Lambert and Vanstone [17] show that we can speed up the rho method using automorphisms. Let $\psi : \langle S\rangle \to \langle S\rangle$ be a group automorphism of order r such that ψ can be computed very efficiently. We define an equivalence relation $\sim$ on the set $\langle S\rangle$ by

$$P \sim Q \Longleftrightarrow P = \psi^j(Q) \text{ for some } j \in [0, r-1].$$

We denote the set of equivalence classes by $\langle S\rangle/\sim$, and let $[P]$ denote the equivalence class containing a point P. The idea of the speed-up using the automorphism ψ is to modify an

iteration function on $\langle S \rangle$ so that it is defined on $\langle S \rangle / \sim$. To achieve this, we can define an iteration function f on $\langle S \rangle / \sim$ by

$$f([P]) := [g(P)]$$

for an iteration function g on $\langle S \rangle$. Since almost all equivalence classes have size r, then the collision search space has size approximately n/r. Hence the expected number of iterations of the rho method sped up by the automorphism ψ is

$$\sqrt{\frac{\pi n}{2r}},$$

which is a speed-up by a factor of $\sqrt{r}$.

Any elliptic curve has the negation map $\psi(P) = -P$ of order 2 as an automorphism. Since the negation map can be computed efficiently, it is useful to use the speed-up of the rho method. Hence the expected number of iterations of the rho method sped up by the negation map is $\frac{\sqrt{\pi n}}{2}$, which is a speed-up by a factor of $\sqrt{2}$. Koblitz curves were first suggested for use in cryptography by Koblitz [26]. The defining equation for a Koblitz curves E is

$$E : y^2 + xy = x^3 + ax^2 + b,$$

where $a, b \in \mathbb{F}_2$ with $b \neq 0$. The *Frobenius map* $\phi : E(\mathbb{F}_{2^m}) \to E(\mathbb{F}_{2^m})$ is defined by

$$\phi : (x, y) \mapsto (x^2, y^2) \text{ and } \phi : \mathcal{O} \mapsto \mathcal{O},$$

where $\mathcal{O}$ is the point of infinity of E. We note that the Frobenius map is a group automorphism of order m on the group $E(\mathbb{F}_{2^m})$ and can be computed efficiently since squaring in $\mathbb{F}_{2^m}$ is relatively inexpensive (see [19] for details). Using both the Frobenius and the negation maps, the rho method on Koblitz curves is sped up. The expected number of iterations of the rho method sped up by both the Frobenius and the negation maps is

$$\frac{1}{2}\sqrt{\frac{\pi n}{m}},$$

which is a speed-up by a factor of $\sqrt{2m}$.

3 Previously known results on the strength comparison

In this section, we summarize previously known results of the security evaluation of ECC and the strength comparison of ECC and RSA: In Table 1, we show the comparable security strengths for the approved algorithms by NIST SP 800-57 [32, Table 2 in p. 63]. We also show the evaluation of solving the ECDLP by ANSI X9.62 [1] in Table 2: For example, the data of Table 2 imply that we need to have a computer with 8.5×10^{11} MIPS to solve the ECDLP of 160-bit in a year. It needs 485 years to solve the ECDLP of 160-bit even if we use a 'Jaguar', which is one of the most powerful computers in the world and has 1.75×10^{15} FLOPS ($\approx 1.75 \times 10^9$ MIPS). Furthermore, we summarize results of the comparable security strengths of ECC and RSA given by certain organizations in Table 3.

Table 1. The comparable security strengths by NIST SP 800-57 [32]

Bits of security	Symmetric key algorithms	FFC (e.g., DSA, D-H)	IFC (e.g., RSA)	ECC (e.g., ECDSA)
80	2TDEA	$L = 1024$, $N = 160$	$k = 1024$	$f = 160 - 223$
112	3TDEA	$L = 2048$, $N = 224$	$k = 2048$	$f = 224 - 255$
128	AES-128	$L = 3072$, $N = 256$	$k = 3072$	$f = 256 - 383$
192	AES-192	$L = 7680$, $N = 384$	$k = 7680$	$f = 384 - 511$
256	AES-256	$L = 15360$, $N = 512$	$k = 15360$	$f = 512+$

L is the size of the public key and N is the size of the private key. The values of k and f are commonly considered to be the key size.

Table 2. The evaluation of solving the ECDLP by ANSI X9.62 [1]

bit size of n	$\sqrt{\pi n/4}$	MIPS year
160	2^{80}	8.5×10^{11}
186	2^{93}	7.0×10^{15}
234	2^{117}	1.2×10^{23}
354	2^{177}	1.3×10^{41}
426	2^{213}	9.2×10^{51}

4 Discussion on the rho method for solving the ECDLP

In this section, we first discuss the choice of iteration functions suitable for solving the ECDLP. We next estimate the number of iterations before obtaining a collision and the processing performance of iterations.

4.1 Choice of suitable iterations

We discuss the choice of iteration functions suitable for solving the ECDLP.

Prime and binary fields cases : A typical iteration function is as follows: Let $\{H_1, H_2, \ldots, H_L\}$ be a random partition of the set $\langle S \rangle$ into L sets of roughly the same size. We call the number L a *partition number*. We write $H(X) = j$ if $X \in H_j$. For $a_j, b_j \in_R [0, n-1], 1 \leq j \leq L$, set $M_j = a_j S + b_j T \in \langle S \rangle$. Then we can define an iteration function by

$$f_{\mathrm{TA}}(X) = X + M_j, \text{ where } j = H(X).$$

For given a point $X = cS + dT$, we can compute $X' = f_{\mathrm{TA}}(X) = c'S + d'T$ with $c' = c + a_j \bmod n$ and $d' = d + b_j \bmod n$. This iteration function is called an *L-adding walk* proposed by Teske (see [43, 44]). Teske also investigated the performance of some iteration functions and showed that the L-adding walk has better performance than the other iteration functions. To analyze the performance of the L-adding walk accurately, we solved the ECDLP over both prime and binary fields of 40 and 50 bits. In the followings, we describe our experiments:

– We used parallelized rho method with $M = 10$ processors and collision detection using distinguished points having $1/\theta$ small enough compared with the order n of the point S.

Table 3. The comparison of security strengths of ECC and RSA with 80-bit security

Report	ECC	RSA
NIST [32]	160	1024
Lenstra [28]	160	1300
RSA Labs. [37]	160	760
NESSIE [31]	160	1536
IETF [33]	-	1228
ECRYPT II [14]	160	1248

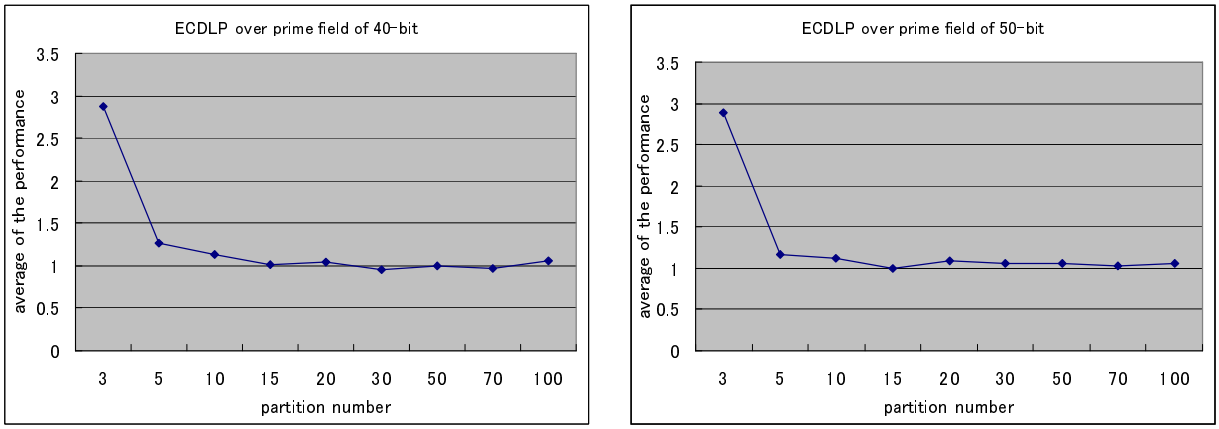


Fig. 1. Experimental results on the average of $\delta(f_{\mathrm{TA}})$ for the ECDLP over prime fields of 40 and 50 bits (Exp $= \sqrt{\pi n/2}$, without the speed-up using the negation map)

– We used the L-adding walk f_{TA} with some $3 \leq L \leq 100$. For each parameter, we solved the ECDLP for 100 times with randomly chosen starting points. Note that we did not use the speed-up with the negation map in our experiments.

To analyze the performance of an iteration function f, we consider the value

$$\delta(f) := (\text{The number of iterations } f \text{ before obtaining a collision}) \,/\mathrm{Exp},$$

where 'Exp' denotes the expected number of iterations (see §2 for details). Note that f has the performance very close to that of random walks and is suitable for solving the ECDLP if $\delta(f)$ is very close to 1. We show our experimental results on the average of $\delta(f_{\mathrm{TA}})$ in Fig. 1 and 2. In [44], Teske analyzed the performance of the L-adding walk by experiments on the ECDLP over prime fields of $5-13$ digits and concluded that the L-adding walk has the performance very close to that of random walks if $L \geq 16$. On the other hand, it follows from our experimental results that the average of $\delta(f_{\mathrm{TA}})$ is very close to 1 in both Fig. 1 and 2 if $L \geq 20$. Hence the L-adding walk with $L \geq 20$ has the performance of very close to that of random walks on average. In using the speed-up with the negation map, we have to deal with the fruitless cycles. Note that choosing larger L decreases the chance of hitting a fruitless cycle, and hence it helps us to reduce the frequency for checking fruitless cycles.

Koblitz curves case : We next consider Koblitz curves case. To solve the ECDLP on Koblitz curves, Gallant, Lambert and Vanstone proposed an iteration function suitable for the rho

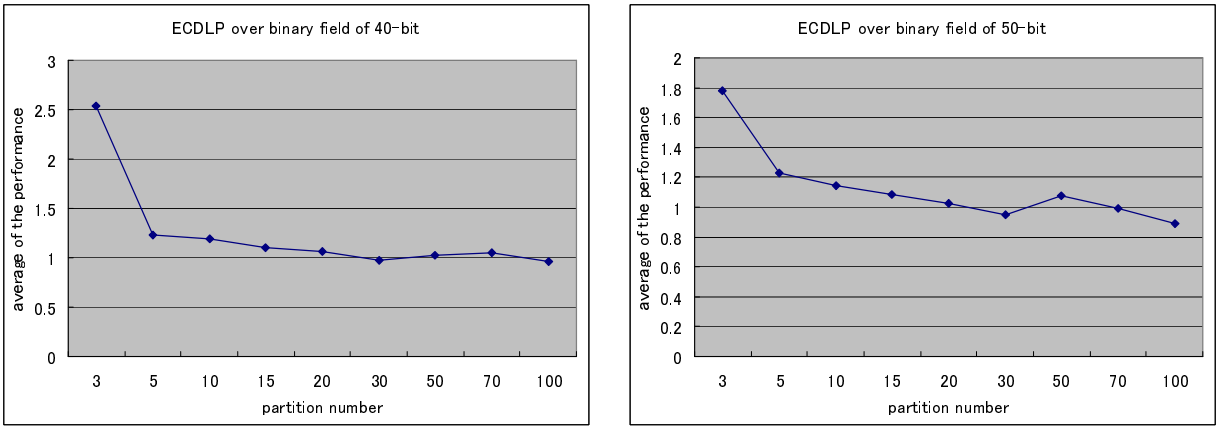


Fig. 2. Experimental results on the average of $\delta(f_{\mathrm{TA}})$ for the ECDLP over binary fields of 40 and 50 bits (Exp $= \sqrt{\pi n/2}$, without the speed-up using the negation map)

method with the speed-up using both the Frobenius and the negation maps as follows [17]: Let E be a Koblitz curve and let $E/\sim$ denote the set of equivalence classes defined by the Frobenius map ϕ and the negation map. We define an iteration function

$$g : X \mapsto X + \phi^j(X), \quad j = \mathrm{hash}_m(\mathcal{L}(X))$$

on the group $E(\mathbb{F}_{2^m})$, where hash_m is a conventional hash function (in the computer science) with range $[0, m-1]$ and $\mathcal{L}$ is a labeling function from the set $E/\sim$ to some set of representatives. For example, the labeling function $\mathcal{L}$ takes the lexicographically least x-coordinate of the elements of the equivalent class. Using the iteration function g, we give an iteration function f_{GLV} on the set $E/\sim$ defined by

$$f_{\mathrm{GLV}}([X]) = [g(X)] \quad \text{for } X \in E,$$

which is the iteration function proposed by Gallant, Lambert and Vanstone (see also §2.2). Note that the iteration function f_{GLV} is a well-defined map on $E/\sim$. In our paper [48], we proposed an iteration function on Koblitz curves which is an extension of f_{GLV} based on the Teske's idea [43]. For $0 \le s \le m$, we define an iteration function on Koblitz curve $E(\mathbb{F}_{2^m})$ given by

$$g_s(X) = \begin{cases} 2X & \text{if } 0 \le j \le s, \\ X + \phi^j(X) & \text{otherwise,} \end{cases}$$

where $j = \mathrm{hash}_m(\mathcal{L}(X))$. As above, we define $f_{\mathrm{GLV},s}$ by $f_{\mathrm{GLV},s}([X]) = [g_s(X)]$ for $X \in E$. Clearly, the iteration function $f_{\mathrm{GLV},s}$ with $s = 0$ is equal to f_{GLV}. We also analyzed the performance of $f_{\mathrm{GLV},s}$ by solving the ECDLP on Koblitz curves $E(\mathbb{F}_{2^m})$ with $m = 41, 53, 83$ and 89 for 100 times with randomly chosen starting points. In Table 4, we give our experimental result on the average of $\delta(f_{\mathrm{GLV},s})$ from [48, Table 3]. From Table 4, we see the followings: The number of iterations $f_{\mathrm{GLV},s}$ before obtaining a collision increases as the parameter s becomes large on average. Furthermore, the iteration function f_{GLV} has the performance very close to that of random walks on average since the average value of $\delta(f_{\mathrm{GLV},s})$ with $s = 0$ is very close to 1.

Table 4. Experimental result on the average of $\delta(f_{\mathrm{GLV},s})$ for $s = 0, m/5, m/3, m/2$ on Koblitz curves $E(\mathbb{F}_{2^m})$ with $m = 41, 53, 83$ and 89 from [48, Table 3] (Exp $= \frac{1}{2}\sqrt{\pi n/m}$)

	$s = 0$	$s = m/5$	$s = m/3$	$s = m/2$
$m = 41$	1.06	1.14	1.29	1.17
$m = 53$	1.10	0.96	1.12	1.26
$m = 83$	1.03	0.99	1.25	1.16
$m = 89$	1.01	1.20	1.08	1.29
average	1.05	1.07	1.18	1.22

Remark 1. For solving the ECC2K-130 which is one of the Certicom ECC challenges on Koblitz curves [13], the authors in [2, 5, 7] proposed an iteration function on $E/\sim$ as follows (see also [20] for the iteration used to solve the ECC2K-95 and the ECC2K-108):

$$f([X]) = [g(X)],\ g(X) = X + \phi^j(X),\ j = ((\mathrm{HW}(x)/2 \bmod 8) + 3), \tag{1}$$

where $\mathrm{HW}(x)$ is the Hamming weight of the x-coordinate of a point $X \in E$. Although this function has the advantage of the fast processing performance, this function might reduce the randomness of the walk due to using $\mathrm{HW}(x)$ (see [7, Appendix B] for details). The authors analyzed the randomness of this function based on a refinement of the heuristic method given by Brent and Pollard [8]. Their analysis shows that the number of iterations with this function is expected to be about 1.07 times of the number of iterations with random walks on average. In order to estimate the number of iterations required to solve the ECDLP with 99% probability, we used f_{GLV} in our analysis. To investigate the randomness of the function defined by (1) in more detail is our future work.

Summary : From the above arguments, the iteration function f_{TA} with $L = 20$ (resp. f_{GLV}) has the performance very close to that of random walks on average in prime and binary cases (resp. in the case of Koblitz curves). For simplicity, we denote by $f_{\mathrm{TA[20]}}$ the L-adding walk with $L = 20$. To evaluate the running time of the rho method, we fix $f_{\mathrm{TA[20]}}$ (resp. f_{GLV}) as an iteration function in the cases of prime and binary fields (resp. in the case of Koblitz curves).

4.2 Number of iterations of the rho method for the ECDLP

We estimate the number of iterations required to solve the ECDLP with very high probability.

Prime and binary fields cases : In Fig. 3 and 4, we give our experimental results on distributions of the number of iterations $f_{\mathrm{TA[20]}}$. Data of Fig. 3 (resp. Fig. 4) are obtained by solving the ECDLP over prime field (resp. binary field) of 40-bit for 10,000 times with randomly chosen starting points (note that we did not use the speed-up with the negation map). As the numbers of iterations before obtaining a collision can be modeled as waiting times, it is reasonable to approximate the graph by Γ-distribution. The theoretic value in Fig. 3 and 4

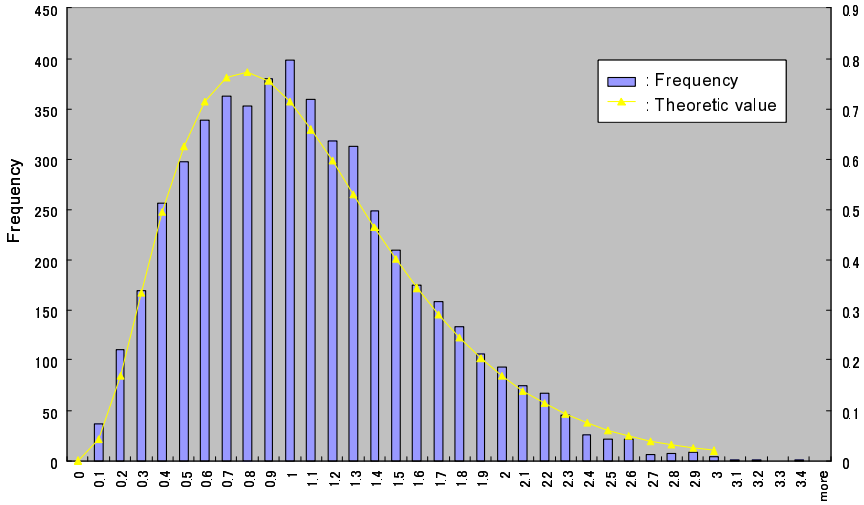


Fig. 3. Distribution of the frequencies of the number of iterations of the rho method for solving the ECDLP over prime field of 40-bit ($1 = \sqrt{\pi n/2}$: the expected number of iterations)

is the curve of Γ-distribution, where we set shape parameter $k = 3.46$ and scale parameter $\theta = 0.317$ in the probability density function

$$f(x; k, \theta) = \frac{1}{\theta^k \Gamma(k)} x^{k-1} e^{-\frac{x}{\theta}}.$$

With this approximation, we see that we can solve the ECDLP with 99% probability if we compute iterations of the rho method by three times of the expected number $\sqrt{\pi n/2}$ (see §2.1 for the expected number of iterations).

It is known that the iteration function with the speed-up using the negation map can fall into a short cycle, which is called "fruitless cycle", and hence the optimal speed-up cannot be expected in general [16, 17]. Recently, Bernstein, Lange and Schwabe in [6] improved the rho method for obtaining a speed-up with the negation map and showed a speed-up very close to $\sqrt{2}$ on hardware with the L-adding walk f_{TA}. To evaluate the running time of the rho method, we consider

$$\frac{3 \cdot \sqrt{\pi n}}{2}$$

as the number of iterations for solving the ECDLP with 99% probability in these cases.

Koblitz curves case : Data of Table 4 shows that the number of iterations f_{GLV} on Koblitz curve $E(\mathbb{F}_{2^m})$ before obtaining a collision is very close to $\frac{1}{2}\sqrt{\pi n/m}$ on average. We also expect that the iteration function f_{GLV} has Γ-distribution same as Fig. 3 and 4. Hence we consider

$$\frac{3}{2} \cdot \sqrt{\frac{\pi n}{m}}$$

as the number of iterations for solving the ECDLP with 99% probability in this case.

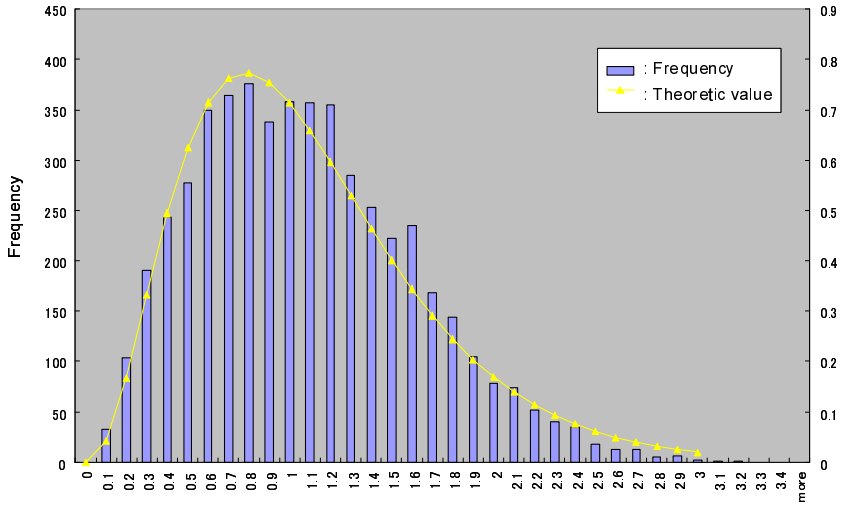


Fig. 4. Distribution of the frequencies of the number of iterations of the rho method for solving the ECDLP over binary field of 40-bit ($1 = \sqrt{\pi n/2}$: the expected number of iterations)

4.3 Processing performance of iterations

We here estimate the processing performance of iterations $f_{\mathrm{TA}[20]}$ and f_{GLV} by using the previously known records. Fix an integer number N. For simplicity, let $t(f)$ denote the processing performance of an iteration function f on an elliptic curve of N-bit.

Prime fields case : The authors in [6] implemented the L-adding walk f_{TA} for solving the ECDLP on an elliptic curve secp112r1 over prime field of 112-bit. They show from their implementation that it needs 306.08 cycles per iteration for their software (CPU: Cell SPE 3GHz). Note that they reported that their software actually took 362 cycles per iteration. There can be a loss in performance if we take so large L that the precomputed points do not fit in cache. Therefore we assume that we take L so that the precomputed points fit in cache and $t(f_{\mathrm{TA}})$ is not affected by the size of L. Moreover, since $t(f_{\mathrm{TA}})$ is approximately equal to the processing performance of a point addition on elliptic curves, we estimate that $t(f_{\mathrm{TA}})$ is proportional to the value $N^{1.585}$ due to the Karatsuba method which is one of the well-known fast multiplication algorithms. From the above arguments, we estimate that we have

$$t(f_{\mathrm{TA}[20]}) = 306.08 \times (N/112)^{1.585} \text{ cycles.}$$

Binary fields case : It is shown in [2] that the processing performance of a point addition of elliptic curves ECC2-131 and NIST 2-131 over binary field of 131-bit is 1047 cycles (CPU: Cell SPE 3GHz). As in the case of prime fields, we estimate that we have

$$t(f_{\mathrm{TA}[20]}) = 1047 \times (N/131)^{1.585} \text{ cycles.}$$

Table 5. The estimated cost of each Step in computing $g(X) = X + \phi^j(X)$

	Step 1	Step 2	Step 3	Step 4	Total
Cost*	0.22	0.25	1.00	0.15	1.62

*We consider the cost of a point addition on E in polynomial-basis as a standard value.

Koblitz curves case : In computing f_{GLV}, the cost of computing $g(X) = X + \phi^j(X)$ is dominant. In our implementation, we take a point $X \in E$ represented by normal-basis as input, get the index $j = \mathrm{hash}_m(\mathcal{L}(X))$ by computing $m-1$ elements $\phi^i(X)$ for $i = 1, \ldots, m-1$ and take $Y = \phi^j(X)$ in normal-basis (Step 1), transform normal-basis to polynomial-basis (Step 2), compute $X + Y$ in polynomial-basis (Step 3), transform bases again (Step 4) and finally output $g(X)$ represented by normal-basis. Note that it needs to map $g(X) \in E$ to $[g(X)] \in E/\sim$ in computing f_{GLV}, but the cost of this mapping is included in Step 1 of the next computation of g. In Table 5, we give the cost of each Step from our implementation (see [47] for details). We estimate from Table 5 that $t(f_{\mathrm{GLV}})$ is 1.62 times of the processing performance of a point addition on E over binary fields in polynomial-basis. Therefore we estimate that we have

$$t(f_{\mathrm{GLV}}) = 1.62 \times 1047 \times (N/131)^{1.585} \text{ cycles.}$$

5 Comparison of the ECDLP and the IFP

In this section, we estimate the computing power required to solve the ECDLP and the IFP, respectively. Furthermore, we calculate the bit sizes of the ECDLP and the IFP that provide the same level of security.

5.1 Computing power required to solve the ECDLP

Since the running time of iterations due to the collision detection of distinguished points is negligible, we see that the running time of the rho method with an iteration function f for solving the ECDLP of N-bit is approximately equal to

$$(\text{the number of iterations}) \times t(f).$$

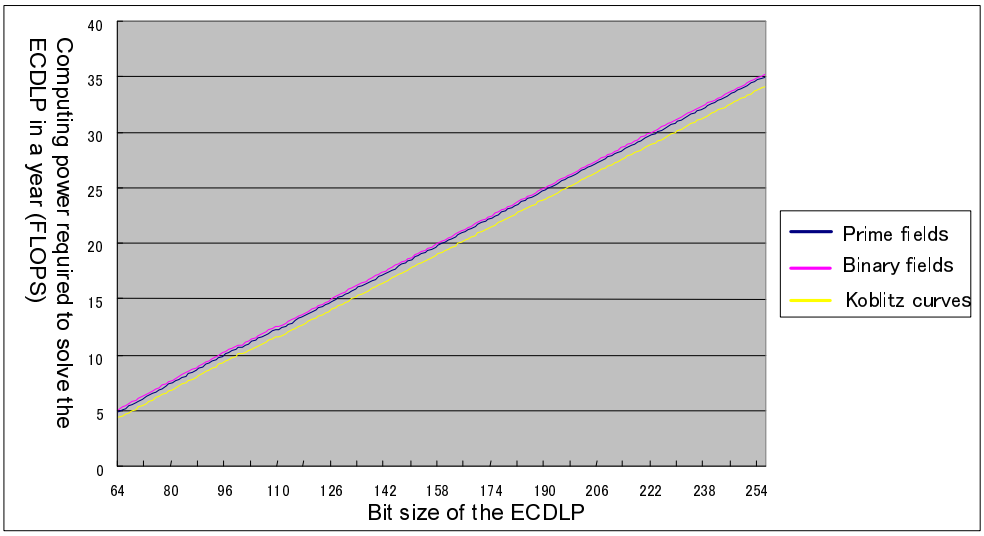
Note that the number of iterations (resp. the value $t(f)$) is discussed in Subsection 4.2 (resp. 4.3). Hence we estimate the computing power T required to solve the ECDLP of N-bit in a year using the rho method as follows (FLOPS is the unit of T):

$$T = \begin{cases} 3 \cdot \sqrt{\pi 2^N}/2 \times 306.08 \times (N/112)^{1.585}/Y & \text{(prime fields case),} \\ 3 \cdot \sqrt{\pi 2^N}/2 \times 1047 \times (N/131)^{1.585}/Y & \text{(binary fields case),} \\ 3 \cdot \sqrt{\pi 2^N/N}/2 \times 1.62 \times 1047 \times (N/131)^{1.585}/Y & \text{(Koblitz curves case).} \end{cases}$$

We set $Y = 365 \cdot 24 \cdot 60 \cdot 60$ (seconds) and $n = 2^N$ as the order of the point S. In Table 6, we give an estimation of the computing power T for each N.

Table 6. Estimation of the computing power T required to solve the ECDLP of N-bit in a year by using the rho method (FLOPS is the unit of T)

Prime fields case		Binary fields case		Koblitz curves case	
N	$\log_{10} T$	N	$\log_{10} T$	N	$\log_{10} T$
86	8.17	84	8.28	88	8.15
107	11.49	105	11.60	110	11.57
115	12.74	112	12.70	117	12.65
123	13.99	120	13.95	125	13.89
138	16.33	136	16.44	141	16.35
154	18.81	151	18.77	157	18.81
176	22.21	173	22.18	179	22.19
207	26.99	205	27.11	211	27.08
247	33.13	245	33.25	251	33.18



Remark 2. In [5], Bernstein et al. measured the computing power to break the Certicom ECC2K-130 challenge based on their extensive experiments. This challenge is to solve the ECDLP on a Koblitz curve E over $\mathbb{F}_{2^{131}}$ with $\sharp E(\mathbb{F}_{2^m}) = 4n$ and $n \approx 2^{129}$. They showed that this challenge would be solved in two years on average using 534 GPUs (1.242 GHz NVIDIA GTX 295, 60 core). On the other hand, we extrapolate by our estimation formula that it needs $10^{14.516}$ FLOPS ($\approx$ 4267 GPUs) to solve the ECDLP of 129-bit in a year with 99% probability. Since the difference is mainly due to the conditions of the solving period (2 years vs 1 year) and the success probability of the rho method (50% vs 99%), our estimation is not so far from their experimental data.

5.2 The computing power required to solve the IFP

We first discuss the complexity of the GNFS to estimate the computing power required to solve the IFP. The GNFS consists of four steps, namely, the polynomial selection step, the

sieving or the relation finding step, the linear algebra step, and the square root step. Among these four steps, the sieving step is dominant procedure theoretically and experimentally. Heuristically, the complexity of the GNFS for factoring a composite integer N is given by

$$L_N\left(\frac{1}{3}, \sqrt[3]{\frac{64}{9}}\right) \text{ as } N \to \infty \tag{2}$$

where

$$L_N(s,c) = \exp((c+o(1))(\log N)^s(\log\log N)^{1-s}).$$

According to [27, 35], the above complexity can be obtained by the following way:

For two positive integers x and z, let $\Phi(x,z)$ denote the probability that an arbitrary integer in the range $[1,x]$ is z-smooth. Note that a positive integer is called *z-smooth* if none of its prime factors is greater than z. It follows from [9] that we have

$$\Phi(x,z) = (\log_z x)^{(-1+o(1))\log_z x} \text{ as } x \to \infty.$$

By using the estimation of $\Phi(x,z)$, we see that the probability that a positive integer at most $L_x(\nu,\lambda)$ is $L_x(\omega,\mu)$-smooth is $L_x(\nu-\omega, -(\nu-\omega)\lambda/\mu)$ as $x \to \infty$. For factoring a composite integer N, take a polynomial

$$f(x) = c_d x^d + c_{d-1}x^{d-1} + \cdots + c_0$$

of degree d and an integer M such that $f(M) \equiv 0 \bmod N$ and $M \approx N^{1/(d+1)}$. Note that the degree d is determined by $d = \lambda^{-1}(\log N/\log\log N)^{1/3}$ and the most suitable value of λ will be determined in the next paragraph. Fix a root θ of $f(x)=0$ and let $K = \mathbb{Q}(\theta)$ denote the number field generated by θ. For a relation (a,b), the absolute value of an algebraic norm

$$\begin{aligned} c_d\mathcal{N}_{K/\mathbb{Q}}(a+b\theta) &= (-b)^d f(-a/b) \\ &= c_d a^d + c_{d-1}a^{d-1}(-b) + \cdots + c_0(-b)^d \end{aligned}$$

is about $N^{1/(d+1)} \cdot \max(a,b)^d$. Take $L_N(1/3, 2\lambda^2)$ as both the upper bound of the factor base and the sieving area of the relations (a,b).

The size of the algebraic norm $c_d\mathcal{N}_{K/\mathbb{Q}}(a+b\theta)$ is approximately equal to

$$\begin{aligned} L_N(1/3,2\lambda^2)^d \cdot N^{1/(d+1)} &= \exp(2\lambda(\log N)^{2/3}(\log\log N)^{1/3}) \cdot \exp(\lambda(\log N)^{2/3}(\log\log N)^{1/3}) \\ &= L_N(2/3, 3\lambda) \end{aligned}$$

and the size of $M = N^{1/(d+1)}$ is approximately equal to

$$N^{1/(d+1)} = \exp(\lambda(\log N)^{2/3}(\log\log N)^{1/3}) = L_N(2/3,\lambda).$$

Then we have the probability that both the above numbers become B-smooth is

$$L_N(1/3, -3(1/3)/(2\lambda)) \cdot L_N(1/3, -(1/3)/(2\lambda)) = L_N(1/3, -2/(3\lambda))$$

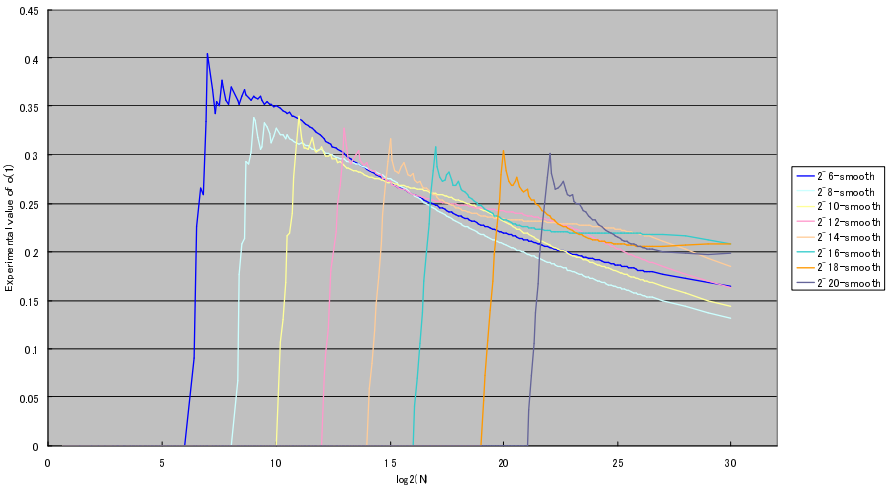


Fig. 5. Experimental result on the $o(1)$-value of $\Phi(x, z)$

Since the number of relations should be more than that of the factor bases, we have

$$L_N(1/3, -2/(3\lambda)) \cdot L_N(1/3, 2\lambda^2)^2 \geq L_N(1/3, 2\lambda^2).$$

We then have $\lambda^3 \geq 1/3$, which shows that $\sqrt[3]{1/3}$ is the most suitable value of λ. Set $\lambda = \sqrt[3]{1/3}$. Since we need more smooth relations than there are elements in the factor base, the complexity of the GNFS is as follows:

$$\begin{aligned}\sharp(\text{the factor base})/(\text{probability of the smoothness}) &= L_N(1/3, 2\lambda^2 + (2/3)/\lambda) \\ &\geq L_N(1/3, (64/9)^{1/3}).\end{aligned}$$

The complexity of the GNFS includes an error value $o(1)$ which is mainly due to the $o(1)$-value of $\Phi(x, z)$. In Figure 5, we give our experimental result on the $o(1)$-value of $\Phi(x, z)$ where x is at most 30-bit. From Figure 5, we expect that the $o(1)$-value is included in the range $[0, 0.2]$ as $x \to \infty$. Since the function $\Phi(x, z)$ is used twice for deriving the complexity of the GNFS, we expect that the error value of the complexity of the GNFS is included in the range $[0, 0.4]$.

The unit of the GNFS complexity is an average cost for checking the smoothness of one element in the sieving area. Practically, such cost depend on various manner, coordinate transformation to special-q lattice, addition of $\log p$ to memory, trial division and pseudo-prime test, cofactorization, etc. Since it seems too complex to evaluate the cost theoretically, we derive the cost from the experimental results of CRYPTREC report 2006. It is shown in CRYPTREC Report 2006 [10, Table 2. 4 and Figure 2.1] that the computing power required to solve the IFP of an integer of each size of RSA is estimated by investigating the computing power of the sieving step based on experiments. Moreover, CRYPTREC Report 2006 [10, Figure 2.2] gives the expected computing power required to solve the IFP of an integer of each size of IFS in a year, which we show in Table 7. We can see that the value of the complexity of the GNFS is very close to each data of Table 8 if we set $o(1) = 0.348172 \in [0, 0.4]$ as the

Table 7. Estimation of factoring [10, 22, 23]

N	(c, d)	#{special-q}	sec/(special-q)	clock/(add. log p)	estimation(PC year)	memory
1024	$2^{16}, 2^{15}$	$1.95 \cdot 10^{12}$	135.0	276.60	$8.4 \cdot 10^{6}$	2GByte
1536	$2^{16}, 2^{15}$	$71.9 \cdot 10^{16}$	207.8	425.76	$4.6 \cdot 10^{12}$	2GByte
2048	$2^{16}, 2^{15}$	$315 \cdot 10^{20}$	245.0	501.98	$25 \cdot 10^{16}$	2GByte

(PC : 2.2 GHz Athlon64 dual core CPU)

Table 8. The expected computing power T required to solve the IFP of N-bit in a year by CRYPTREC report 2006 [10] (FLOPS is the unit of T)

N	768	1024	1536	2048
$\log_{10} T$	12.6879	16.3395	22.2319	27.0413

error value of the complexity of the GNFS and $\ell = 1.0373 \times 10^7$ as the leading coefficient of L_N. In Table 9, we give the estimation of the computing power required to solve the IFP in a year.

5.3 ECDLP vs IFP

In our estimation, we make the following assumptions:

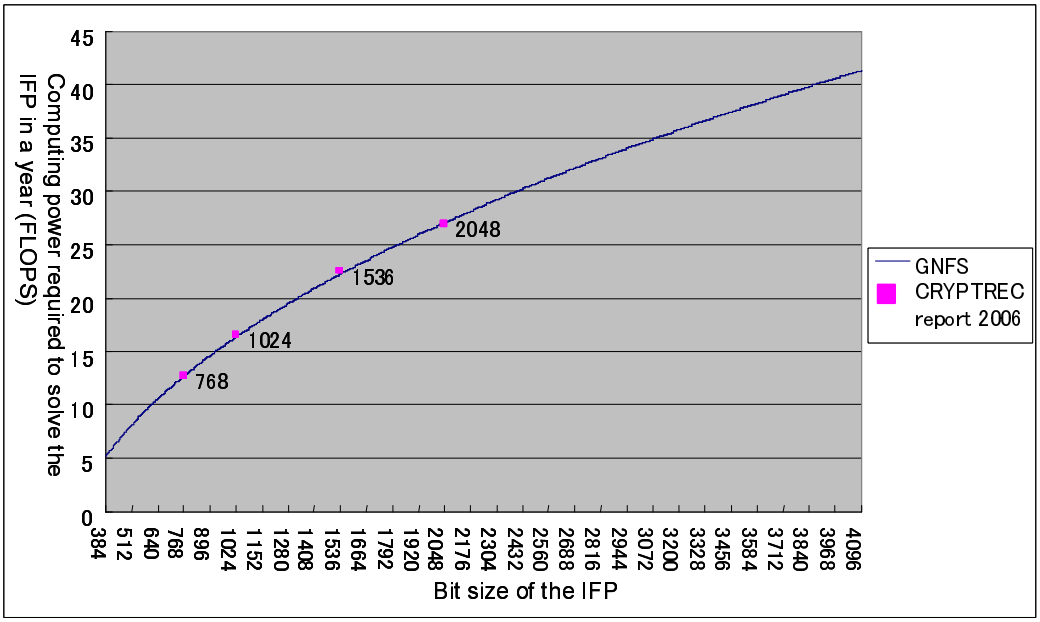
- ECDLP side:
  - The complexity of the iteration function scales proportionally to N^α, where N is the bit length and α is a constant determined by the multiplication method. Figures in Table 6 and 10 are obtained when we use Karatsuba method for multiplications, i.e. $\alpha = 1.585$. If we use ordinary multiplication, figures of the ECDLP bit length in Table 6 and 10 can be reduced at most two bits.
  - The memory requirement of the rho method can be controlled by using the distinguishing points. Therefore the memory requirement of the rho method is negligible.
- IFP side:
  - The complexity of the GNFS is given by the formula (2). Using latest experimental data, we calculated $o(1) = 0.348172$ and $\ell = 1.0373 \times 10^7$. We assume that these values apply for larger integers.
  - We note that the memory requirements of the GNFS against the numbers with the target bit length in Table 7 are the same size (2G Bytes). In [22], there is another estimation of the GNFS of 1536 bits by using 3.5G Byte memory, they estimated its complexity can be reduced from $4.6 \cdot 10^{12}$ to $4.2 \cdot 10^{12}$. Generally speaking, unlimited memory size for executing the GNFS would reduce the computational complexity. The strict evaluation of such effect is our future work.

We then calculate the bit sizes of the ECDLP and the IFP that provide the same level of security. In Table 10, we give an estimation of the strength comparison of the ECDLP and the IFP. In particular, we have the followings from Table 10:

- The security of 768-bit IFP is close to that of 115-bit ECDLP over prime field, 112-bit ECDLP over binary field, or 117-bit ECDLP on Koblitz curves. The world records of

Table 9. The estimation of the computing power T required to solve the IFP of N-bit in a year using GNFS from CRYPTREC Report 2006 [10] (FLOPS is the unit of T)

N	$\log_{10} T$
512	8.21
696	11.53
768	12.68
851	13.94
1024	16.33
1219	18.75
1536	22.23
2048	27.04
2839	33.20



2011 for solving the RSA and ECC are 768-bit RSA in 2010 [24] and 112-bit ECC over prime field in 2009 [15], respectively. Since the times of these two records are close, we consider that these records indicate reasonability of our estimation.

– The security of 1024-bit IFP is close to that of 138-bit ECDLP over prime field, 136-bit ECDLP over binary field, or 141-bit ECDLP on Koblitz curves. Though it is often said that 160-bit ECC corresponds to 1024-bit RSA, our estimation indicates that shorter ECC key sizes provide the same level of security.

6 Conclusions

In this paper, we evaluated the complexity of the rho method for solving the ECDLP based on state-of-the-art theory and experiments, and estimated the computing power required to solve the ECDLP in a year (see Table 6). We also estimated the computing power required

Table 10. Estimation of the bit sizes of the ECDLP and the IFP providing the same level security by comparing Table 6 and 9

Bit size of the IFP	Bit size of the ECDLP		
	Prime fields	Binary fields	Koblitz curves
512	86	84	88
696	107	105	110
768	115	112	117
851	123	120	125
1024	138	136	141
1219	154	151	157
1536	176	173	179
2048	207	205	211
2839	247	245	251

to solve the IFP based on results of CRYPTREC Report 2006 (see Table 9), and gave an estimation of the strength comparison of the ECDLP and the IFP (see Table 10). If we say 160-bit ECC has 80-bit security because the complexity of the rho method is square root, our estimation indicates that 1024-bit RSA does not reach the 80-bit security.

Acknowledgements

A part of this research is financially supported by a contract research with the National Institute of Information and Communications Technology (NICT), Japan.

References

1. ANSI X9.62, Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), (1999).
2. D. Bailey, B. Baldwin, L. Batina, D. Bernstein, P. Birkner, J. Bos, G. van Damme, G. de Meulenaer, J. Fan, T. Güneysu, F. Gurkaynak, T. Kleinjung, T. Lange, N. Mentens, C. Paar, F. Regazzoni, P. Schwabe and L. Uhsadel, "The Certicom Challenges ECC2-X", IACR ePrint Archive, 2009/466, available at `http://eprint.iacr.org/2009/466`, (2009).
3. D. Bernstein, "Curve25519: New Diffie-Hellman Speed Records", *Proceeding of PKC 2006*, LNCS 3958, pp. 207-228, (2006).
4. D. Bernstein, "Speed Reports for Elliptic-Curve Cryptography", available at `http://cr.yp.to/ecdh/reports.html`, (2010).
5. D. Bernstein, H. Chen, C. Cheng, T. Lange, R. Niederhagen, P. Schwabe and B. Yang, "ECC2K-130 on NVIDIA GPUs", *Proceeding of Indocrypt 2010*, LNCS 6498, pp. 328-344, (2010).
6. D. Bernstein, T. Lange and P. Schwabe, "On the Correct Use of the Negation Map in the Pollard rho Method", *Proceeding of PKC 2011*, LNCS 6571, pp. 128-146, (2011).
7. Breaking ECC2K-130, IACR ePrint Achive, 2009/541, available at `http://eprint.iacr.org/2009/541.pdf`.
8. R. Brent and J. Pollard, "Factorization of the eighth Fermat number", Mathematics of Computation 36, pp. 627-630, (1981).
9. E. R. Canfield, P. Erdos and C. Pomerance, "On a problem of Oppenheim concerning Factorisatio Numerorum", J. Number Theory 17, pp. 1-28, (1983).
10. CRYPTREC, CRYPTREC Report 2006, available at `http://www.cryptrec.go.jp/report/c06_wat_final.pdf`, (2006).
11. I. Blake, G. Seroussi and N. Smart, Elliptic Curves in Cryptography, Cambridge University Press, (1999).

12. Certicom, Certicom ECC Challenge, available at http://www.certicom.jp/images/pdfs/cert_ecc_challenge.pdf, (1997).
13. Certicom, Curves List, available at http://www.certicom.jp/index.php/curves-list, (1997).
14. ECRYPT II, "ECRYPT II Report on Key Sizes (2011)", available at http://www.keylength.com/en/3/, (2011).
15. EPFL IC LACAL, "PlayStation 3 computing breaks 2^{60} barrier 112-bit prime ECDLP solved", available at http://lacal.epfl.ch/112bit_prime, (2009).
16. S. D. Galbraith and R. S. Ruprail, "Using Equivalence Classes to Accelerate Solving the Discrete Logarithm Problem in a Short Interval", *Proceeding of PKC 2010*, LNCS 6056, pp. 368-386, (2010).
17. R. Gallant, R. Lambert and S. Vanstone, "Improving the Parallelized Pollard Lambda Search on Binary Anomalous Curves," Mathematics of Computation 69, pp. 1699-1705, (2000).
18. T. G$\acute{\text{u}}$neysu, T. Kasper, M. Novotný, C. Paar and A. Rupp, "Cryptanalysis with COPACOBANA", Transactions on Computers, Nov. 2008, Vol. 57, pp. 1498-1513, (2008).
19. D. Hankerson, A. Menezes and S. Vanstone, Guide to Elliptic Curve Cryptography, Springer Professional Computing, (2004).
20. R. Harley, Elliptic curve discrete logarithms project, available at http://pauillac.inria.fr/~harley/ecdl/.
21. T. Izu, J. Kogure and T. Shimoyama, "CAIRN 2: An FPGA Implementation of the Sieving Step in the Number Field Sieve Method", Cryptographic Hardware and Embedded Systems 2007, pp. 364-377, (2003).
22. T. Kleinjung, "Estimates for factoring 1024-bit integers", Securing Cyberspace: Applications and Foundations of Cryptography and Computer Security, Workshop IV: Special purpose hardware for cryptography: Attacks and Applications, slides are available at http://www.ipam.ucla.edu/schedule.aspx?pc=scws4., (2006).
23. T. Kleinjung, "Evaluation of Complexity of Mathematical Algorithms", CRYPTREC technical report No.0601 in FY2006, available at http://www.cryptrec.jp/estimation.html, (2007).
24. T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev and P. ZImmermann, "Factorization of a 768-bit RSA modulus", *Advances in Cryptology CRYPTO 2010*, LNCS 6223, pp. 333-350, (2010).
25. D. Knuth, The art of computer programming, Seminumerical Algorithms, vol. II, Addison-Wesley. Reading, (1969).
26. N. Koblitz, "Elliptic curve cryptosystems," Mathematics of Computation 48, pp. 203-209, (1987).
27. A. Lenstra, H. Lenstra, M. Manasse and J. Pollard, "The Number Field Sieve", *Symposium on Theory of Computing - STOC '90*, pp. 564-572, ACM, (1990).
28. A. Lenstra and E. Verheul, "Selecting Cryptographic Key Sizes", Journal of Cryptology 14, No. 4, pp. 255-293, (2001).
29. A. Menezes, T. Okamoto and S. Vanstone, "Reducing elliptic curve logarithms to logarithms in a finite field," IEEE Transactions on Information Theory 39, pp. 1639-1646, (1993).
30. V. S. Miller, "Use of elliptic curves in cryptography", *Advances in Cryptology CRYPTO 1985*, LNCS 218, pp. 417-426, (1986).
31. NESSIE, "NESSIE Security Report", Feburary 2003.
32. NIST Special Publication 800-57, available at http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf.
33. H. Orman and P. Hoffman, "Determining Strengths for Public Keys Used for Exchanging Symmetric Keys", IETF RFC 3766/BCP 86, April 2004.
34. J. Pollard, "Monte Carlo methods for index computation mod p", Mathematics of Computation 32, pp. 918-924, (1978).
35. C. Pomerance, "The Number Field Sieve", Proceedings of Symposia in Applied Mathematics 48, pp. 465-480, (1994).
36. R. Rivest, A. Shamir and L. Adelman, "A method for obtaining digital signatures and public-key cyrptosystems", Communications of the ACM 21, pp. 120-126, (1978).
37. RSA Labs., "A Cost-Based Security Analysis of Symmetric and Asymmetric Key Lengths", RSA Labs Bulletin, No. 13, April 2000 (Revised November 2001).
38. T. Satoh and K. Araki, "Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves," Commentarii Mathematici Universitatis Sancti Pauli 47, pp. 81-92, (1998).
39. I. Semaev, "Evaluation of discrete logarithms in a group of p-torsion points of an elliptic curve in characteristic p," Mathematics of Computation 67, pp. 353-356, (1998).
40. A. Shamir, "Factoring Large Numbers with the TWINKLE Device (Extended Abstract)", Cryptographic Hardware and Embedded Systems 1999, LNCS 1717, pp. 2-12, (1999).
41. A. Shamir and E. Tromer, "Factoring Large Numbers with the TWIRL Device", *Advances in Cryptology CRYPTO 2003*, LNCS 2729, pp. 1-26, (2003).
42. N.P. Smart, "The discrete logarithm problem on elliptic curves of trace one," Journal of Cryptology 12, pp. 110-125, (1999).

43. E. Teske, "Speeding up Pollard's rho method for computing discrete logarithms", *Algorithmic Number Theory-ANTS III*, LNCS 1423, pp. 541-554, (1998).
44. E. Teske, "On random walks for Pollard's rho method", Mathematics of Computation 70, pp. 809-825, (2001).
45. P. C. van Oorschot and M. J. Wiener, "Parallel collision search with cryptanalytic applications", Journal of Cryptology 12, pp. 1-28, (1999).
46. M. J. Wiener and R. J. Zuccherato, "Fast attacks on elliptic curve cryptosystems", *Selected Areas in Cryptology-SAC '98*, LNCS 1556, pp. 190-200, (1999).
47. M. Yasuda, T. Izu, T. Shimoyama and J. Kogure "On the attacking evaluation of elliptic curve cryptography on Koblitz curves" SCIS 2010, Pre-proceedings, 1D2-5, (2010).
48. M. Yasuda, T. Izu, T. Shimoyama and J. Kogure, "On random walks of Pollard's rho method for the ECDLP on Koblitz curves", Journal of Math-for-Industry, Vol. 3 (2011B-3), pp. 107-112, (2011).

A Flexible Hardware ECDLP Engine in Bluespec

Lyndon Judge and Patrick Schaumont

Center for Embedded Systems for Critical Applications (CESCA)
Bradley Department of Electrical and Computer Engineering
Virginia Tech, Blacksburg, VA 24061, USA
{lvjudge1,schaum}@vt.edu

Abstract. A parallel hardware implementation of the Pollard rho algorithm is a complex design requiring multiple layers of design hierarchy. In this contribution, we investigate an ECDLP design for a secp112r1 curve and we discuss the design difficulties of a traditional hardware design method based on Verilog. The lack of flexible control structures, and the bottom-up style of hardware design leads to a rigid architecture, incapable of supporting quick architecture changes and design space exploration. We then present the same ECDLP implementation using Bluespec, a rule-based hardware description language. The language features of Bluespec support effective design space exploration. We demonstrate this by comparing several secp112r1 ECDLP variants, with non-trivial modification of parameters. We investigate the speed area tradeoff resulting from various design parameters and demonstrate performance from 53K to 2.87M iterations per second for Xilinx FPGA slice counts varying from 4,407 to 35,232 slices. We emphasize that these numbers were obtained by measuring the performance of a prototype, rather than estimating them from synthesis tool output. We conclude that the design of complex cryptanalytic hardware can greatly benefit from better hardware design methodologies, and we would like to advocate the importance of this aspect.

1 Introduction

Elliptic curve cryptography (ECC) has become a popular choice for public key cryptography implementations due to its smaller key sizes relative to RSA. The security of ECC is due to the difficulty of solving the elliptic curve discrete logarithm problem (ECDLP). Therefore, previous work has been devoted to determining the security of ECC by attempting to solve ECDLP for various key sizes. The parallelized Pollard's Rho method, introduced by van Oorschot and Wiener [14], is the best known algorithm to solve ECDLP. So far, Pollard's Rho method has been implemented on a variety of accelerator platforms including FPGAs, Playstation 3 Cell processors, and GPUs. It's the purpose of this contribution to examine the limitations of hardware design with Verilog and investigate a design methodology using Bluespec, a more recent proposal.

A quick literature review shows that custom architectures in hardware outperform software implementations, due to the computational parallelism offered

by hardware. Table 1 gives an overview of published ECDLP implementations for various curves. Of all proposed systems, Fan's hardware implementation of an attack on ECC2k-130 [8] achieves top performance. Other hardware designs including [1] and [7] claim strong performance relative to competing platforms. In [8], Fan presents a hardware system to solve ECDLP for ECC2k-130 with computation speed of 111M point additions per second per core using Xilinx Spartan-3 FPGAs. This outperforms an attack on the same curve using GPUs that achieves 63M point additions per second [2] and one using Cell CPUs that computes 4.28M point additions per second [4]. These results demonstrate that existing programmable hardware platforms are capable of achieving top performance.

Table 1. Summary of prior ECDLP systems

Platform	Curve	Publication Year	Point Additions per Second
Spartan-3 FPGA [8]	ECC2k-130 130 bit, binary field	2010	111M
Nvidia GTX 295 GPU [2]	ECC2k-130 130 bit, binary field	2010	63M
Spartan-3E FPGA [1]	ECC2k-130 130 bit, binary field	2009	33.67M
Cell Processor [1]	ECC2k-130 130 bit, binary field	2009	27M
Nvidia GTZ295 GPU [1]	ECC2k-130 130 bit, binary field	2009	12.56M
Spartan-3E FPGA [7]	ECC2-131 131 bits, binary field	2007	10.0M
Cell Processor [3]	secp112r1 112 bit, prime field	2010	8.8M
Cell Processor [5]	secp112r1 112 bit, prime field	2009	7M
Cell Processor [4]	ECC2k-130 130 bit, binary field	2010	4.3M
Virtex 5 FPGA [11]	secp112r1 112 bit, prime field	2011	660K
Spartan-3 FPGA [10]	128 bit, prime field	2007	57.8K

However, many of the advanced optimizations used in software are not found in hardware implementations. These optimizations, such as negation maps [3] and tag tracing [6], use complex decision making or complex data structures, features that are difficult to support in classic hardware design. Indeed, the standard hardware design process uses hardware description languages (HDLs) to describe behavior at the register transfer level (RTL). The low abstraction

level of HDLs increases the design cost of hardware systems well beyond that of functionally equivalent software systems.

- In HDLs, a design is expressed as a hierarchy of modules and designers must explicitly manage the interactions between modules. The interface of a module is defined in structural terms, through ports that carry input signals or output signals. Typical module interactions (synchronization, data transfers, and so on) must be expressed in terms of signal transitions on module ports. All these elements encourage a designer to focus on low-level details at the expense of developing the big picture. A designer thinks of design with modules as a bottom-up design process: low-level modules must be completed before proceeding to higher levels.
- HDLs require that control be expressed explicitly. Designers must manually define required control logic using finite state machines. In large systems requiring complex control structures, expressing the control explicitly greatly increases design time. By themselves, finite state machines do not support control hierarchy. This makes hierarchical control concepts, such as subroutines and exceptions, particularly difficult to express using HDLs. Manual design of complex controllers with techniques such as microprogramming, tends to be rigid, with poor support for architectural exploration. Hence, the design of large highly optimized hardware systems is a labor intensive process that is difficult to adapt for reuse or experimentation.

A low abstraction level is not always a disadvantage; a software programmer naturally switches from C to assembly when performance is a concern. But a hardware designer using HDL has no choice: everything is designed at low abstraction level. Low abstraction level also becomes a concern when a designer is working through the initial stages of a design: analyzing the design problem, refining the specification, understanding the implementation cost. Indeed, design space exploration is essential to master large, complicated designs, including cryptanalytic systems. The most successful cryptanalytic systems are those that can be easily adapted for such experimentation. Unfortunately, the bottom up design flow and explicit control definition required by HDLs mean that even minor changes to a hardware architecture may require time consuming design changes and logic verification. As a result, these explorations are often skipped, leaving a possible suboptimal design.

In this paper, we investigate, in the context of an ECDLP design, how changing the hardware description language can improve the design process. We use Bluespec and show how this language overcomes the pitfalls of traditional HDLs. We demonstrate design space exploration using Bluespec by presenting several alternative implementations of a secp112r1 ECDLP engine, with variations in ECDLP batch vector size and number of point addition modules. We did not yet evaluate more advanced Pollard rho optimizations, such as negation maps and tag lists; we defined this as future work. However, we think our current results already provide credible evidence that higher abstraction level will greatly help the hardware design of this problem.

The remainder of this paper is structured as follows. In the next section, we give a brief overview of the Bluespec System Verilog language and Bluespec design flow. In Section 3, we give a high-level specification of the modular multiplier required for the ECDLP system and we explain the design of a modular multiplier for secp112r1 using Bluespec in Section 4. We discuss the complete system architecture and Bluespec implementation in Section 5. We give the implementation results of our design space exploration in Section 6 and finally conclude the paper.

2 Bluespec

Bluespec [13] is an electronic system level (ESL) hardware synthesis toolset that claims faster and more accurate hardware design at a higher abstraction level than HDLs. Bluespec is centered on the Bluespec System Verilog (BSV) hardware programming language. BSV introduces several unique programming constructs to raise the abstraction level of the hardware design process without sacrificing the designer's control over system microarchitecture or resource usage. In addition to BSV, Bluespec provides a compiler and simulator to allow generation of fully synthesizable Verilog and cycle-accurate simulation of BSV designs.

2.1 Modules and Interfaces

Bluespec designs use a module hierarchy similar to that of HDL designs, but they are constructed differently. Each module includes a datapath, behavior, and an interface. The datapath consists of instantiations of lower level components used in the module such as wires, registers, and user-defined modules. The behavior of a module is defined using rules and methods to control data manipulation by datapath elements. The interface encapsulates the input and output behavior of the module by means of methods. Two key elements of Bluespec not found in classic HDL are rules and interface methods.

Interface methods define all input and output operations that can be performed by the module. Methods are classified into three types: Action methods that cause state changes, Value methods that return data values, and ActionValue methods that both cause state changes and return data. Action and ActionValue methods accept zero or more arguments as inputs to the module, while Value and ActionValue methods return a value as module output. The Bluespec compiler determines the implicit execution conditions for each method and adds the appropriate control signals to the design to enforce these conditions. This allows designers to ensure correct behavior without explicitly defining the handshaking logic required to synthesize that behavior in hardware.

Listing 1.1 is an example of a simple 18x16 single cycle multiplier in BSV. The multiplier interface includes two methods; *load*, which provides the input operands and *result*, which returns the product. The interface methods are defined within the multiplier module as shown. Operand multiplication is performed immediately during a load. The result is stored in the *product* register

and returned by the *result* method. Note that, by means of appending a zero-bit at the most-significant-bit, we ensure an unsigned multiplier. Since this is a single cycle design, no additional control is required to define the module behavior. The pragmas *synthesize* and *always_ready* control mapping of the Bluespec design into Verilog. The *synthesize* pragma directs the compiler to generate a separate Verilog module, rather than inlining instantiations of the BSV module. The *always_ready* pragma prevents generation of a ready signal for the specified interface methods.

Listing 1.1. Single cycle 18x16 multiplier

```
interface Multiplier;
  method Action load(Bit#(18) f, Bit#(16) g);
  method Bit#(34) result();
endinterface

(* synthesize *)
(* always_ready = "load,result" *)
module mkMultiplier(Multiplier);
  Reg#(Bit#(34)) product <- mkReg(0);

  method Action load(Bit#(18) f, Bit#(16) g);
    product <= {0, f} * {0, g};
  endmethod

  method Bit#(34) result();
    return product;
  endmethod
endmodule
```

2.2 Control

Behavior of Bluespec modules is expressed using rules. A rule consists of one or more actions guarded by a Boolean condition. Each rule is executed atomically, which guarantees that all actions in the rule either execute simultaneously in parallel or do not execute. An advantage of atomicity is that each rule can be evaluated individually at any given point in time to determine how the state of the module is impacted. Designers can define each rule individually without explicitly specifying the relationships between rules in the module. Designers must ensure that all actions required to execute in the same cycle are included in a single rule and that no rule contains conflicting actions. As long as these conditions are satisfied, the Bluespec compiler will determine a schedule of rule execution that preserves the functional behavior of the design. The compiler also prevents conflicting rules from firing at the same time; the order of execution for conflicting rules can be manually specified by the designer or automatically assigned by the compiler based on overall schedule requirements of the design.

In HDLs, it is common practice to use a ready signal to indicate completion of a multicycle operation. BSV provides the specialized *Maybe* datatype to

encapsulate a ready signal and result value into a single type. Rather than maintaining the control as a separate signal from the data, designers can assign both simultaneously by either tagging the Maybe-typed variable invalid or assigning a data value. To facilitate the use of the Maybe type, the Bluespec library includes methods to read data from and check the validity of Maybe-typed variables.

2.3 Example: Multicycle Multiplier

Listing 1.2 extends the earlier multiplier example to illustrate the use of rules and Maybe datatype in BSV for a multicycle 18x16 multiplier. This multiplier represents each operand as a set of coefficients and computes the product in three clock cycles using conventional schoolbook multiplication. The interface of the multicycle multiplier returns the product as a Maybe type.

The multiplication requires two stages of 9x8 multiplication and two stages of partial product accumulation and alignment. A state register, *mult_state*, is used to track the current stage of the computation and each rule is guarded by a condition that allows it to execute only during the proper stage. The load stage is specified within the load method definition and stores each operand in a register for use during computation. The multiplication stages are controlled by rules *mult_st1* and *mult_st2* in Listing 1.2. The first partial product accumulation stage executes in the same clock cycle as the second multiplication stage and aligns the four partial product into the upper and lower halves of the 34-bit product. This stage is implemented by the rule *do_acc*. Result assignment stage tags the Maybe-typed product as valid and assigns the 34-bit result by aligning its previously computed halves and adding the carry bit from the lower half to the upper half.

The *invalid_product* rule tags the product as invalid. Since both *assign_res* and *invalid_product* drive *product*, the rules are in conflict whenever both are able to fire, specifically when *mult_state* equals three. The functionally correct behavior is for *assign_res*, rather than *invalid_product*, to fire when *mult_state* is three. This is enforced by manually assigning higher priority to the *assign_res* rule using the syntax *(* descending_urgency = "assign_res, invalid_product" *)*.

Listing 1.2. Multicycle 18x16 multiplier

```
interface MultiMultiplier;
  method Action load(Bit#(18) f, Bit#(16) g);
  method Maybe#(Bit#(34)) result();
endinterface

(* synthesize *)
module mkMultiMultiplier(MultiMultiplier);
  Reg#(Bit#(18)) f_in <- mkReg(0);
  Reg#(Bit#(16)) g_in <- mkReg(0);
  Reg#(Bit#(17)) r0 <- mkReg(0);
  Reg#(Bit#(17)) r2 <- mkReg(0);
  Reg#(Bit#(18)) acc1 <- mkReg(0);
  Reg#(Bit#(17)) acc2 <- mkReg(0);
```

```
  Wire#(Bit#(17)) r1 <- mkWire;
  Wire#(Bit#(17)) r3 <- mkWire;
  Reg#(Maybe#(Bit#(34))) product <- mkWire;
  Reg#(Bit#(2)) mult_state <- mkReg(0);

(* descending_urgency = "assign_res, invalid_res" *)
  rule mult_st1(mult_state == 1);
    r0 <= {0,f_in[8:0]} * {0,g_in[7:0]};
    r2 <= {0,f_in[8:0]} * {0,g_in[15:8]};
  endrule

  rule mult_st2(mult_state == 2);
    r1 <= {0,f_in[17:9]} * {0,g_in[7:0]};
    r3 <= {0,f_in[17:9]} * {0,g_in[15:8]};
  endrule

  rule do_acc(mult_state == 2);
    acc1 <= {0,r0} + ({0,r2[8:0]} << 8) + ({0,r1[7:0]} << 9);
    acc2 <= r3 + {0,r2[16:9]} + {0,r1[16:8]};
  endrule

  rule assign_res(mult_state == 3);
    product <= tagged Valid (({0,acc2} << 17) + {0,acc1});
  endrule

  rule invalid_res;
    product <= tagged Invalid;
  endrule

  rule state_ctrl(mult_state != 0);
    mult_state <= mult_state + 1;
  endrule

  method Action load(Bit#(18) f, Bit#(16) g);
    f_in <= f;
    g_in <= g;
    mult_state <= 1;
  endmethod

  method Maybe#(Bit#(34)) result();
    return product;
  endmethod
endmodule
```

With only minor modifications, the multicycle 18x16 multiplier shown in Listing 1.2 can be adapted to support pipelining. A pipelined multiplier would use the same interface and computation stages as the multicycle multiplier. To support pipelining, intermediate results can be represented as Maybe types and their validity can replace the state variable in rule firing conditions. Additionally, *r1* and *r3* must change from wires to registers.

3 Modular Multiplication

We have used Bluespec to design a parallel Pollard's rho engine for secp112r1. This curve uses a prime field $GF(\frac{(2^{128}-3)}{76439})$. We modeled our arithmetic using a redundant 128-bit representation, following ideas from Bernstein et al. in [3] and Bos et al. in [5]. We also applied ideas from Güeneysu in [9] to obtain a parallel implementation of the multiplication. Finally, we optimized the resulting structure by tightly integrating reduction and multiplication, by making use of the properties of the unbalanced modulus $(2^{128} - 3)$. Our design has been presented in [11], but for completeness we recall its key characteristics.

We represent the integers in secp112r1 redundantly, in the ring $R = \mathbb{Z}/q\mathbb{Z}$, where $q = p * 76439$. This allows us to perform arithmetic modulo $q = 2^{128} - 3$ instead of modulo $p = \frac{(2^{128}-3)}{76439}$. This transformation simplifies the reduction operation as follows.

Since $q = 2^{128}-3$ is an unbalanced exponent, a reduction becomes a constant-multiplication and an addition. Indeed, assume $A = A_1.2^{128} + A_0$, then $A \mod (2^{128} - 3) = A_0 + 3.A_1 \ mod \ (2^{128} - 3)$. Furthermore, a redundantly represented integer mod q can be converted into a canonical form by multiplying it with $a = 76439$. If $a \mid q$ and $p = q/a$, then $v \mod p \equiv v.a \mod q$. Therefore, if we start from elements in $GF(p)$, and perform computations with them in R, we can obtain a unique representation of them by multiplying the result in R with $a = 76439$.

We represent a 128-bit number as 8 fields of 16 bit each. To multiply two such numbers F and G, one needs to compute 64 partial products. Our design computes 8 partial products in parallel by multiplying one field of operand G with all fields from operand F. As proposed by Güeneysu [9], placing G in a shift register and F in a rotating register ensures that each multiplier produces aligned results, that can be directly accumulated. We also integrated reduction into the multiplication as follows. When rotating the most significant field from F into the least significant field, we multiply it with 3, thereby obtaining a reduction for $(2^{128} - 3)$.

Fig. 1 shows the block diagram of the modular multiplier. The top of the figure shows the two registers F and G, which are connected to an array of 8 multipliers. Each multiplier is 16-bit by 18-bit, and is able to generate partial products of the form $F_i * G_j$ and $3.F_i * G_j$. The output of the multipliers therefore is 34 bit.

The partial products are accumulated with a layer of 21-bit and 19-bit accumulators. Each accumulator combines two inputs: a 16-bit input from the least-significant part of a multiplier output with the 18-bit most-significant bits of the next-lower multiplier output. The most-significant 18-bit of the most-significant field is reduced, and accumulated at the least significant field.

After accumulation, we obtain a product of overlapping 21-bit and 19-bit fields. These fields are then merged into a representation of non-overlapping 16-bit fields by means of carry propagation. Again, carries at the most-significant side are converted into reduced carries that are added at the least significant

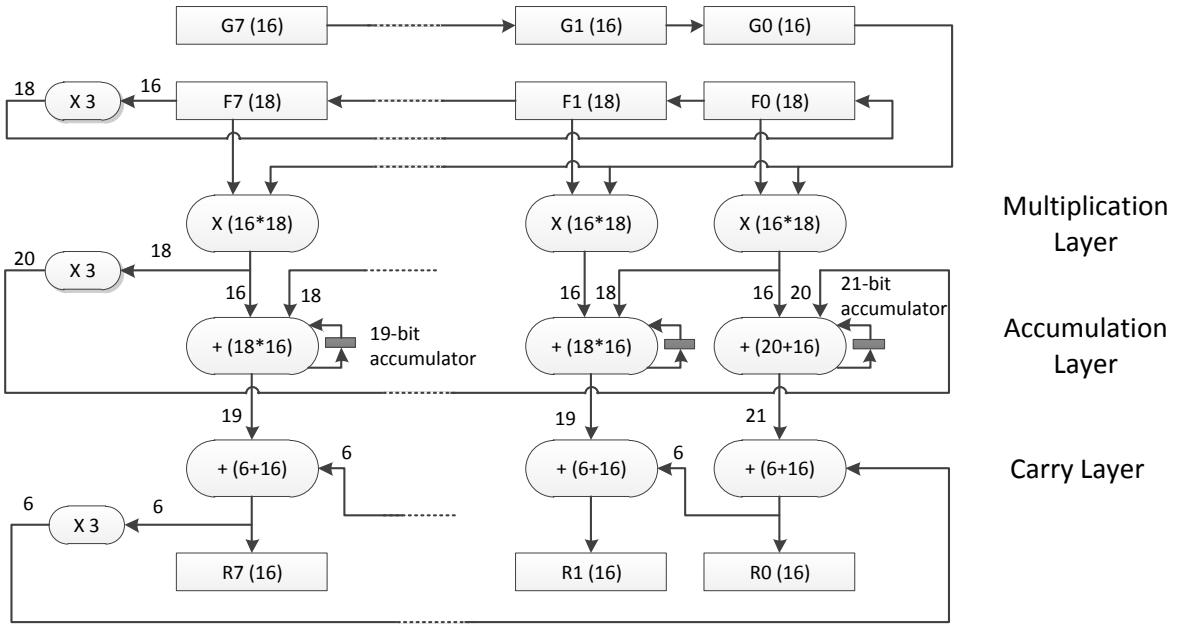


Fig. 1. Block diagram of modular multiplier architecture

side. The carries need to be propagated until the final result R is stable. Through simulation, we experimentally verified that this happens after no more then 14 clock cycles for the worst possible F and G inputs. Compared to a modular multiplier with separate multiplication and reduction, our proposed structure is more compact, and results in a shorter latency. Indeed, a full-multiplication and full-reduction structure would use 20 clock cycles (8 for multiplication, 12 for reduction) as opposed to 14 clock cycles for the merged structure.

In the following section, we will describe the mapping of this design in Bluespec code, and subsequently in FPGA.

4 Designing Modular Multiplier in Bluespec

We use Bluespec to complete a hardware implementation of the modular multiplier for secp112r1 as an example of the concepts and design methodologies that can be used in Bluespec designs. The complete code listing for the modular multiplier is included as Appendix A.

4.1 Design Hierarchy

The hierarchy of the Bluespec design is based on the block diagram in Fig. 1. Low level components required by the modular multiplier are multipliers, accumulators, and adders. Each component receives two input operands and produces a

result after one clock cycle. In BSV, the input operands are provided via an Action method that accepts two arguments and the output is returned via a Value method. Since the result is computed in one clock cycle, no rules or explicit control is required. The low level multipliers and accumulators make use of the high-speed DSP48 blocks in the Virtex-5 FPGA to complete the multiplications and partial product accumulations in a single clock cycle.

4.2 Interface

The interface of the modular multiplier is fairly straightforward; the multiplier receives two 128-bit operand inputs and produces a 128-bit output 14 clock cycles later. The value returned by the result method uses the Maybe datatype. Since modular multiplication is a multiple clock cycle operation, the Maybe-typed return value can only be read when the result is valid, 14 clock cycles after the operands are received.

4.3 Control

Control logic for the modular multiplication is implemented using rules. The atomicity of BSV rules allows design of rules for each computation stage independently, relying on the Bluespec compiler to handle scheduling of and interactions between rules. Computation of the modular multiplication is controlled by rules *do_mult*, *wrap*, *do_acc*, and *do_carry* (lines 47 - 86). These rules load the low level components to perform multiplication, accumulation, and reduction of partial products. None of these rules uses Boolean conditions to guard execution; they execute every clock cycle without any concept of control state.

The validity of the Maybe-typed result is controlled using an enabled counter that assigns the output value as valid after 14 clock cycles. The rule *incr_counter* (lines 90 - 92) controls the cycle counter. The cycle count is incremented every clock cycle until it reaches 15. During the fourteenth clock cycle, the output is tagged valid and assigned the concatenated value of the fully reduced 16-bit coefficients by rule *assign_res* (lines 94 - 100).

The Bluespec compiler derives an execution schedule for all rules in the module based on explicit and implicit conditions of each. The explicit conditions are the Boolean conditions used to guard rule execution. Implicit rule conditions are derived by the Bluespec compiler based on the handshaking conditions and logic required implement interface methods. For example, rule *do_mult* does not have any explicit condition, but it both reads and writes operand registers *f_in* and *g_in*. The load method also writes to *f_in* and *g_in*, storing the input arguments as the operands. This produces an implicit condition on *do_mult*. Implicit conditions can prevent a rule from executing in the same cycle as another rule or method. In general, the Bluespec compiler always assigns higher priority to interface methods than module rules, so rules that conflict with a method will not execute when the method is called, regardless of the explicit rule condition.

5 Microcoded ECDLP Architecture

Using Bluespec, we have implemented a full system to attack secp112r1. We exploit the high abstraction level supported for Bluespec designs to implement a robust system that is easily adaptable to the architectural changes required to explore the design space. Our system includes a complete point addition datapath that can be parallelized to perform multiple point addition steps simultaneously. Each point addition unit supports vectorized inversion with a variable vector size. Bluespec allows us to work at a higher abstraction level to implement our design directly from a natural language interpretation of the specification. The system is designed to work in conjunction with a software driver that provides seed points and performs collision search on distinguished points. The complete code listing for the ECDLP system is located at https://sourceforge.net/p/ecdlpbluespec/.

5.1 Hardware Implementation

Our hardware design is a Bluespec adaptation of the design presented in [11], which is a hardware implementation of [3]. A block diagram of this design is shown in Fig. 2. The design consists of a point addition datapath and microcoded point addition controller. The point addition datapath uses high performance modular arithmetic units for multiplication, addition, subtraction, and inversion mapped onto high-speed DSP48 blocks in the FPGA. The microcoded controller sends control signals to datapath components to implement the sequence of operations required for point addition. The controller supports multiple parallel point addition datapaths executing in SIMD fashion.

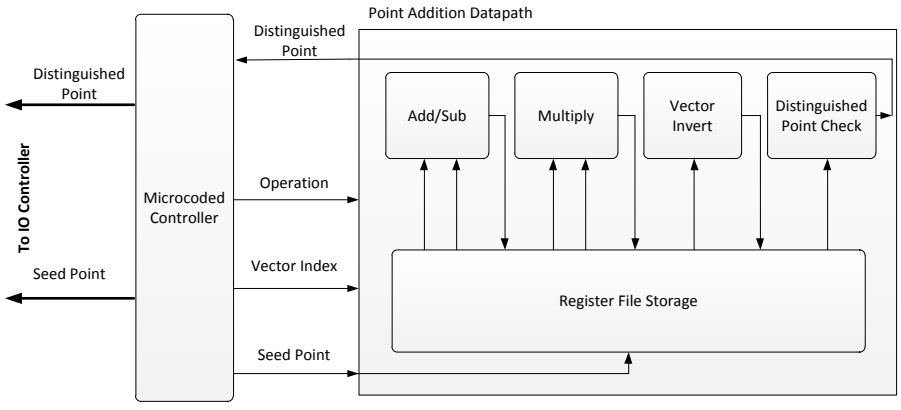


Fig. 2. Block diagram of microcoded ECDLP architecture

5.2 Vectorization

As in [11], we use Mongomery's trick [12] to reduce the cost of inversion. The vectorized point addition datapath computes each operation of the point addition on a vector of points, rather than a single point. The cost of vectorized inversion is shared by all vector elements, whereas other vectorized operations incur a cost equal to the non-vectorized cost times the number of elements. Control signals for the vectorized datapath are generated by the microcoded controller. The controller iterates through all vectors for a given operation before proceeding to the next operation.

We implement the vectorized datapath using register files provided in the Bluespec library to store starting points and temporary results for the point addition. The number of entries in each register file is equal to the vector size. Adjustment of this parameter is supported by the RegFile Bluespec library module. Since all control logic is specified using rules, vector size can be modified without any changes to the control.

5.3 Microcoded Controller in Bluespec

Our implementation is functionally equivalent to a conventional HDL microcoded controller, except that we use Bluespec rules to implement the point addition controller at a higher abstraction level. Each microinstruction is encoded onto a single Bluespec rule. The rule condition maintains the order of operations and rule actions provide control signals to the point addition datapath. An internal program counter is used to control iteration through vector elements and the order of the field operation of a point addition.

Interface The controller interface allows loading seed points and returning distinguished points. The interface, shown in Listing 1.3, provides methods to indicate when a seed point needs to be loaded or a distinguished point has been found. These signals are monitored by an IO module that manages communication with the software driver. In order to avoid stalling or missing points while waiting to send a distinguished point, the controller stores distinguished points in a FIFO. When the IO module reads a distinguished point, the first point is returned and removed from the FIFO. The FIFO is implemented from the Bluespec library and its size can be easily adjusted to accommodate the number of parallel point additions performed by the system.

Listing 1.3. Interface for microcoded controller

```
interface Pactrl;
  method Action loadSeed(Bit#(256) seedpt);
  method Action waitForSeed();
  method Action reset();
  method Action deqDpSp();
  method Bit#(512) retDpSp();
  method Bool loadRq();
```

```
  method Bool dpSendRq();
endinterface
```

Control Our approach to vectorization and microoperation control requires one rule to implement each field operation of point addition. As an example, the rule for the first field operation of a point addition is given in Listing 1.4. For all operations, the rule condition ensures that the rule only executes when the program counter equals the appropriate operation and the previous datapath operation is complete, as indicated by the *fire_cond* signal. The actions within the rule assert control signals for the point addition datapath to perform the operation. The arguments of *loadAddSub* in Listing 1.4 are control signals add/-subtract operation select, operand 1 index, operand 2 index, register file write enable, and vector element index, respectively. Other field operations provide similar control signals. As shown, the controller is configured to support four parallel point addition datapaths. The number of parallel point addition units can be easily modified by instantiating additional datapaths and adding actions to the microoperation rules to assert required control signals.

Listing 1.4. Microcoded control for field operations of a point addition

```
//Counter to iterate through vector element ids
rule vec_cnt_ctrl(incrCond);
  vecCnt <= vecCnt + 1;
endrule

//Set condition to increment vector counter
rule incr_cond_ctrl;
  case (oper) matches
    0: incrCond <= (!ldRqFlg || seedAvail_n);
    1: incrCond <= addSubResVal;
    2: incrCond <= addSubResVal;
    3: incrCond <= (invResVal || vecCnt < 7);
    4: incrCond <= addSubResVal;
    5: incrCond <= multResVal;
    6: incrCond <= multResVal;
    7: incrCond <= addSubResVal;
    8: incrCond <= addSubResVal;
    9: incrCond <= multResVal;
    10: incrCond <= addSubResVal;
    11: incrCond <= chkResVal;
  endcase
endrule

//Register increment condition wire
rule fire_cond_ctrl;
  fire_cond <= incrCond;
endrule

//  1: t1 = Py - Qy
rule op1_ctrl(oper == 4'd1 && fire_cond);
```

```
    pa0.loadAddSub(0, 5, 7, 1, vecCnt);
    pa1.loadAddSub(0, 5, 7, 1, vecCnt);
    pa2.loadAddSub(0, 5, 7, 1, vecCnt);
    pa3.loadAddSub(0, 5, 7, 1, vecCnt);
endrule
```

Internal state management for the microcoded controller uses additional rules to handle iteration through vector elements and incrementation of the operation counter. Vector iteration is controlled using a simple counter that increments whenever the previous datapath operation is complete, as shown in rule *vec_cnt_ctrl* from Listing 1.4. After control signals for the current operation have been asserted for all vector elements, the operation counter increments and proceeds to the next operation. Operation and vector iteration control rely heavily on the use of the Maybe datatype to signal completion of datapath operations.

6 Implementation Results

We demonstrate the suitability of Bluespec for design space exploration in the context of a complex ECDLP machine for secp112r1. We have implemented our design for various vector sizes between 1 and 64 with the number of parallel point addition cores varying between one and four. The high abstraction level of Bluespec allows us to make these changes easily. The performance results for each variation are compared and analyzed to determine the optimum parameters for our ECDLP machine.

For all design variations, our hardware system interacts with a software driver that provides seed points and performs collision search on distinguished points. This is implemented using a Nallatech computing platform that includes a quad-core Xeon processor (E7310, 1.6GHz) and a Virtex-5 (xq5vsx240t) FPGA with 37,440 slices of reconfigurable logic. To demonstrate the correctness of our design, we use a point of low order (2^{50}, specified in [3]), which leads to an expected collision within only 2^{25} point addition steps.

6.1 Design Variations

We evaluate the impact of vector size and number of point addition cores on performance and area of our design by implementing both a single-core and four core version for vector sizes from 1 to 64. Our results are given in Table 2. As shown, best performance is achieved by the designs with the largest vector size. Due to the additional storage required for vectorization, the area required to implement each design also increases with the vector size. Therefore, determination of optimal design parameters must account for the tradeoff between performance and area.

Comparison of results from the one and four core implementations shows that increasing the number of cores produces a linear increase in both performance and area. This is expected because we implement point addition cores as parallel instantiations of the point addition datapath. The number of clock

Table 2. Implementation results for ECDLP with variable vector size and number of cores.

	One Core		Four Core	
Vector Size	Speed $[PA/s]$	Area $[slices]$	Speed $[PA/s]$	Area $[slices]$
1	53K	4407	214K	12,391
8	300K	5977	1.20M	17,705
32	598K	6619	2.38M	29,478
64	717K	9692	2.87M	35,232

cycles required for a point addition step remains constant, but the addition of cores allows that cost to be spread over a larger number of points resulting in a higher number of point additions per second. In addition, our results show that for absolute performance, more cores is preferable over a larger vector size. Consider the four core implementation with vector size 8 and the one core implementation with vector size 32. Both generate 32 points per iteration, but the four core implementation achieves twice as many point additions per second at 2.5 times the area.

6.2 Speed/Area Tradeoff

Determination of the optimal design parameters for our ECDLP system requires analysis of the performance and area of each alternative architecture. Our goal is to find the configuration that will best utilize the limited area of the FPGA to achieve top performance. Thus, we consider not only the speed of each variation, but also the area requirements. Our evaluation of the speed/area tradeoff using the metric point additions per second per slice is shown in Fig. 3. Our design achieves the best results, 90 point additions per second per slice, for vector size of 32. For larger vector sizes, the extra storage required to increase the vector size outweighs the performance benefit of larger vectors, resulting in a degradation of overall performance per unit area. In our system architecture, vectorization is entirely independent of number of cores. Thus, the optimal design can be realized by implementing the maximum number of cores, each with vector size 32.

6.3 Comparison with Prior Work

In terms of absolute performance, our design is slower than previous published solutions. For example, [5] solves ECDLP for secp112r1 using Cell processors to perform point additions in 453 clock cycles, which yields 7M point additions per second. [3] improves this solution, computing point additions in 362 cycles and achieving 8.8M point additions per second. The performance gap between these prior works and our design is due to the limited clock frequency of the

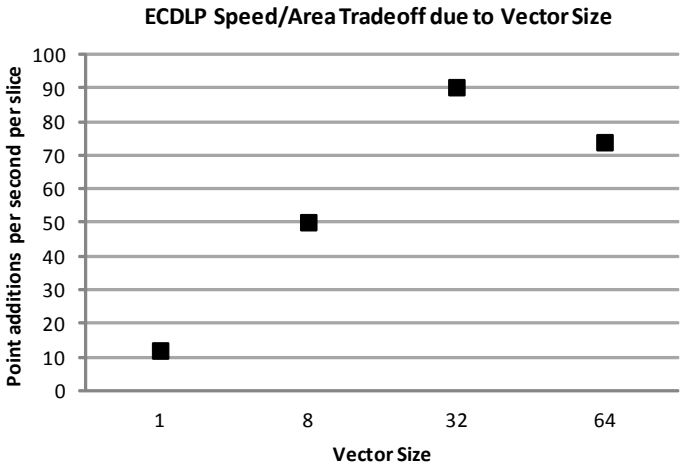


Fig. 3. Evaluation of the performance per unit area

hardware design. Our design runs at 100 MHz, while the Cell processors used by [5] and [3] run at 3.125 GHz. In terms of cycles per point addition, our design is competitive, with the single core 32 element vector implementation requiring 167 clock cycles per point addition. Furthermore, using the full area of the FPGA, we estimate that our design could implement 8 cores with vector size 32 and compute 4.8M point additions per second.

7 Conclusion

HDL coding is very low-level compared to the complexity handled in a typical ECDLP architecture. Their bottom-up design flow, and low-level coding of control does not encourage design-space exploration. We presented our implementation using Bluespec, a hardware description language that claims higher abstraction level for hardware design. Several factors, including the separation of interface from behavior, and the flexible specification of rule-based control, seem to confirm the suitability of Bluespec for complex design tasks. In addition, we demonstrated the feasibility of quick design-space exploration for a Bluespec-based design. Future work includes the evaluation of more complicated Pollard rho enhancements (such as negation maps, and/or tag lists) using Bluespec. Another open question is whether Bluespec is accessible as a parallel design language to software-oriented designers.

References

1. Bailey, D.V., Batina, L., Bernstein, D.J., Birkner, P., Bos, J.W., Chen, H.C., Cheng, C.M., van Damme, G., de Meulenaer, G., Perez, L.J.D., Fan, J., Güneysu, T., Gurkaynak, F., Kleinjung, T., Lange, T., Mentens, N., Niederhagen, R., Paar, C., Regazzoni, F., Schwabe, P., Uhsadel, L., Herrewege, A.V., Yang, B.Y.: Breaking ECC2K-130. IACR Cryptology ePrint Archive, Report 2009: 541 (2009), http://eprint.iacr.org/2009/541
2. Bernstein, D., Chen, H.C., Cheng, C.M., Lange, T., Niederhagen, R., Schwabe, P., Yang, B.Y.: ECC2K-130 on Nvidia GPUs. In: Gong, G., Gupta, K. (eds.) Progress in Cryptology - INDOCRYPT 2010, Lecture Notes in Computer Science, vol. 6498, pp. 328–346. Springer Berlin / Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-17401-8_23
3. Bernstein, D., Lange, T., Schwabe, P.: On the correct use of the negation map in the Pollard rho method. In: Catalano, D., Fazio, N., Gennaro, R., Nicolosi, A. (eds.) Public Key Cryptography - PKC 2011, Lecture Notes in Computer Science, vol. 6571, pp. 128–146. Springer Berlin / Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-19379-8_8
4. Bos, J., Kleinjung, T., Niederhagen, R., Schwabe, P.: ECC2K-130 on Cell CPUs. In: Bernstein, D., Lange, T. (eds.) Progress in Cryptology - AFRICACRYPT 2010, Lecture Notes in Computer Science, vol. 6055, pp. 225–242. Springer Berlin / Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-12678-9_14
5. Bos, J.W., Kaihara, M.E., Kleinjung, T., Lenstra, A.K., Montgomery, P.L.: On the security of 1024-bit RSA and 160-bit elliptic curve cryptography. IACR Cryptology ePrint Archive, Report 2009: 389 (2009), http://eprint.iacr.org/2009/389
6. Cheon, J., Hong, J., Kim, M.: Speeding up the Pollard rho method on prime fields. In: Pieprzyk, J. (ed.) Advances in Cryptology - ASIACRYPT 2008. Lecture Notes in Computer Science, vol. 5350, pp. 471–488. Springer Berlin / Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-89255-7_29
7. Meurice de Dormale, G., Bulens, P., Quisquater, J.J.: Collision search for elliptic curve discrete logarithm over GF(2^m) with FPGA. In: Paillier, P., Verbauwhede, I. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2007, Lecture Notes in Computer Science, vol. 4727, pp. 378–393. Springer Berlin / Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-74735-2_26
8. Fan, J., Bailey, D., Batina, L., Güneysu, T., Paar, C., Verbauwhede, I.: Breaking elliptic curve cryptosystems using reconfigurable hardware. In: Field Programmable Logic and Applications (FPL), 2010 International Conference on. pp. 133–138. IEEE Computer Society (Aug 31, 2010-Sept 2, 2010), http://doi.ieeecomputersociety.org/10.1109/FPL.2010.34
9. Güneysu, T., Paar, C.: Ultra high performance ECC over NIST primes on commercial FPGAs. In: Oswald, E., Rohatgi, P. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2008, Lecture Notes in Computer Science, vol. 5154, pp. 62–78. Springer Berlin / Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-85053-3_5
10. Güneysu, T., Paar, C., Pelzl, J.: Attacking elliptic curve cryptosystems with special-purpose hardware. In: Field programmable gate arrays (FPGA), 2007 ACM/SIGDA 15th international symposium on. pp. 207–215. ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1216919.1216953
11. Mane, S., Judge, L., Schaumont, P.: An integrated prime-field ECDLP hardware accelerator with high-performance modular arithmetic units. In: Athanas,

P.M., Becker, J., Cumplido, R. (eds.) Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on. pp. 198–203. IEEE Computer Society (Nov 30, 2011-Dec 2, 2011), http://doi.ieeecomputersociety.org/10.1109/ReConFig.2011.12

12. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Mathematics of Computation 48(177), 243–264 (1987), http://dx.doi.org/10.2307/2007888
13. Nikhil, R.S.: Bluespec: A general-purpose approach to high-level synthesis based on parallel atomic transactions. In: Coussy, P., Morawiec, A. (eds.) High-Level Synthesis, pp. 129–146. Springer Netherlands (2008), http://dx.doi.org/10.1007/978-1-4020-8588-8_8
14. van Oorschot, P.C., Wiener, M.J.: Parallel collision search with cryptanalytic applications. Journal of Cryptology 12, 1–28 (1999), http://dx.doi.org/10.1007/PL00003816

8 Modmul.bsv : BSV code listing for Modular Multiplier

```
1  interface ModMul;
2    method Action load(Bit#(128) f, Bit#(128)g);
3    method Maybe#(Bit#(128)) result();
4  endinterface
5
6  (* synthesize *)
7  (* always_ready = "load" *)
8  module mkModMul(ModMul);
9    Reg#(Bit#(144)) f_in <- mkReg(0);
10   Reg#(Bit#(128)) g_in <- mkReg(0);
11   Reg#(Bit#(4)) rcnt <- mkReg(15);
12   Reg#(Bit#(128)) prev_res <- mkReg(0);
13   Wire#(Bit#(34)) wrap_around <- mkWire();
14   Wire#(Bit#(34)) carry_wrap <- mkWire();
15   Wire#(Bit#(34)) mod_f <- mkWire();
16   Reg#(Maybe#(Bit#(128))) cout <- mkWire;
17   Multiplier m0 <- mkMultiplier();
18   Multiplier m1 <- mkMultiplier();
19   Multiplier m2 <- mkMultiplier();
20   Multiplier m3 <- mkMultiplier();
21   Multiplier m4 <- mkMultiplier();
22   Multiplier m5 <- mkMultiplier();
23   Multiplier m6 <- mkMultiplier();
24   Multiplier m7 <- mkMultiplier();
25   Accumulator a0 <- mkAccumulator();
26   Accumulator a1 <- mkAccumulator();
27   Accumulator a2 <- mkAccumulator();
28   Accumulator a3 <- mkAccumulator();
29   Accumulator a4 <- mkAccumulator();
```

```
30   Accumulator a5 <- mkAccumulator();
31   Accumulator a6 <- mkAccumulator();
32   Accumulator a7 <- mkAccumulator();
33   Adder c0 <- mkAdder();
34   Adder c1 <- mkAdder();
35   Adder c2 <- mkAdder();
36   Adder c3 <- mkAdder();
37   Adder c4 <- mkAdder();
38   Adder c5 <- mkAdder();
39   Adder c6 <- mkAdder();
40   Adder c7 <- mkAdder();
41
42   rule do_mult;
43     f_in <= {f_in[125:0], mod_f[17:0]};
44     g_in <= g_in >> 16;
45     m0.load(f_in[17:0], g_in[15:0]);
46     m1.load(f_in[35:18], g_in[15:0]);
47     m2.load(f_in[53:36], g_in[15:0]);
48     m3.load(f_in[71:54], g_in[15:0]);
49     m4.load(f_in[89:72], g_in[15:0]);
50     m5.load(f_in[107:90], g_in[15:0]);
51     m6.load(f_in[125:108], g_in[15:0]);
52     m7.load(f_in[143:126], g_in[15:0]);
53   endrule
54
55   rule wrap;
56     wrap_around <= 3 * {0, m7.result()[33:16]};
57     carry_wrap <= 3 * {0, c7.result()[21:16]};
58     mod_f <= 3 * {0, f_in[143:126]};
59   endrule
60
61   rule do_acc;
62     a0.add(wrap_around[19:0], m0.result()[15:0]);
63     a1.add({0, m0.result()[33:16]}, m1.result()[15:0]);
64     a2.add({0, m1.result()[33:16]}, m2.result()[15:0]);
65     a3.add({0, m2.result()[33:16]}, m3.result()[15:0]);
66     a4.add({0, m3.result()[33:16]}, m4.result()[15:0]);
67     a5.add({0, m4.result()[33:16]}, m5.result()[15:0]);
68     a6.add({0, m5.result()[33:16]}, m6.result()[15:0]);
69     a7.add({0, m6.result()[33:16]}, m7.result()[15:0]);
70   endrule
71
72   rule do_carry;
73     c0.add(a0.result(), carry_wrap[5:0]);
74     c1.add(a1.result(), c0.result()[21:16]);
```

```
75      c2.add(a2.result(), c1.result()[21:16]);
76      c3.add(a3.result(), c2.result()[21:16]);
77      c4.add(a4.result(), c3.result()[21:16]);
78      c5.add(a5.result(), c4.result()[21:16]);
79      c6.add(a6.result(), c5.result()[21:16]);
80      c7.add(a7.result(), c6.result()[21:16]);
81    endrule
82
83    rule incr_counter (rcnt < 4'd15);
84      rcnt <= rcnt + 1;
85    endrule
86
87    rule assign_res;
88      if (rcnt == 4'd14)
89        cout <= tagged Valid ({c7.result()[15:0],
                  c6.result()[15:0], c5.result()[15:0],
                  c4.result()[15:0], c3.result()[15:0],
                  c2.result()[15:0], c1.result()[15:0],
                  c0.result()[15:0]});
90      else
91        cout <= tagged Invalid;
92    endrule
93
94    method Action load(Bit#(128) f, Bit#(128) g);
95      f_in <= {2'd0, f[127:112], 2'd0, f[111:96], 2'd0,
             f[95:80], 2'd0, f[79:64], 2'd0, f[63:48], 2'd0,
             f[47:32], 2'd0, f[31:16], 2'd0, f[15:0]};
96      g_in <= g;
97      a0.reset();
98      a1.reset();
99      a2.reset();
100     a3.reset();
101     a4.reset();
102     a5.reset();
103     a6.reset();
104     a7.reset();
105     rcnt <= 0;
106   endmethod
107
108   method Maybe#(Bit#(128)) result();
109     return cout;
110   endmethod
111 endmodule
```

Solving Discrete Logarithms in Smooth-Order Groups with CUDA[1]

Ryan Henry Ian Goldberg

Cheriton School of Computer Science
University of Waterloo
Waterloo ON Canada N2L 3G1

{rhenry,iang}@cs.uwaterloo.ca

Abstract

This paper chronicles our experiences using CUDA to implement a parallelized variant of Pollard's rho algorithm to solve discrete logarithms in groups with cryptographically large moduli but smooth order using commodity GPUs. We first discuss some key design constraints imposed by modern GPU architectures and the CUDA framework, and then explain how we were able to implement efficient arbitrary-precision modular multiplication within these constraints. Our implementation can execute roughly 51.9 million 768-bit modular multiplications per second — or a whopping 840 million 192-bit modular multiplications per second — on a single Nvidia Tesla M2050 GPU card, which is a notable improvement over all previous results on comparable hardware. We leverage this fast modular multiplication in our implementation of the parallel rho algorithm, which can solve discrete logarithms modulo a 1536-bit RSA number with a 2^{55}-smooth totient in less than two minutes. We conclude the paper by discussing implications to discrete logarithm-based cryptosystems, and by pointing out how efficient implementations of parallel rho (or related algorithms) lead to trapdoor discrete logarithm groups; we also point out two potential cryptographic applications for the latter. Our code is written in C for CUDA and PTX; it is open source and freely available for download online.

1 Introduction

Over the past several decades, the algorithms and symbolic computation research communities have made considerable advances with respect to state-of-the-art algorithms for solving many number-theoretic problems of interest. At the same time, Moore's law has ensured steady speed increases in the computing devices on which these algorithms run. The results have been astounding: modern symbolic computation packages, such as Maple[2] and Mathematica[3], accept arbitrary-precision operands and can solve a plethora of useful problems very efficiently. Nonetheless, much work remains; there exist many fundamental problems for which no efficient (i.e., polynomial-time) algorithm is known. While there is considerable interest in expanding the range of problems that a modern symbolic computation toolkit can solve efficiently, it turns out that there are also practical advantages to having some problems remain intractable, specifically when the "inverse problem" is efficient to compute. In particular, such *one-way functions* give rise to public-key cryptosystems, such as the ones that protect our everyday online transactions.

In practice, nearly all public key cryptosystems derive their security guarantees from assumptions about (possibly relaxations of) one of the following three types of presumed 'hard' problems: those related to

[1]The latest version of this paper is available as CACR Tech Report 2012-02, http://cacr.uwaterloo.ca/techreports/2012/cacr2012-02.pdf.

[2]http://www.maplesoft.com/

[3]https://www.wolfram.com/mathematica/

factoring large integers, those related to computing discrete logarithms, or those related to solving certain computational problems on integer lattices. This paper is concerned with a variant of the second problem on this list; i.e., that of computing discrete logarithms when the modulus is large but has smooth totient (more precisely, it focuses on computing discrete logarithms in the multiplicative group of units modulo N when the group order $\varphi(N)$ is B-smooth for some $B \ll N$; that is, when all of the prime factors of $\varphi(N)$ are less than B). In particular, we explore the extent to which one can leverage the massive, yet cost-effective, parallelism provided by modern *general-purpose graphics processing units* (GP GPUs) to solve discrete logarithms (with respect to certain special moduli) that would otherwise be impractical to solve using CPUs alone. We also discuss the implications of being able to solve such discrete logarithms for existing discrete logarithm-based cryptosystems, and demonstrate how this ability gives rise to a useful cryptographic primitive called a *trapdoor discrete logarithm group*.

Outline. Before commencing our foray into parallel programming on GPUs with CUDA and PTX in §3, we first overview related work in the literature in §1.1, and then briefly touch on some mathematical preliminaries, including discrete logarithms and the parallel rho method for computing them, in §2. The key to an efficient realization of parallel rho on GPUs is fast modular multiplication; therefore, we devote much of §4 to the implementation details of efficient arbitrary-precision modular multiplication in CUDA, including various optimizations that lead to dramatic performance improvements over a naive implementation. §4 also discusses how we tailored the parallel rho method to run well within the hardware constraints of CUDA GPUs. A performance evaluation of our implementation appears in §5; based on these empirical observations, as well as some theoretical analysis, §6.1 discusses the implications for deployed discrete-logarithm-based cryptosystems, while §6.2 concludes that GPU-based implementations of parallel rho are sufficient to realize trapdoor discrete logarithm groups, and suggests some possible cryptographic applications. §7 wraps up with a brief summary and a pointer to our open source implementation.

1.1 Related work

In recent years, there has been considerable interest in using commodity graphics processing units (GPUs) to perform highly parallelized computations at a low cost, especially for use in speeding up (and attacking) public key cryptosystems. Because GPUs are particularly well suited to solving systems of linear equations, it should be unsurprising that several high-speed implementations of lattice-based cryptosystems have successfully employed them. For example, Hermans et al. [19] implemented NTRUEncrypt — the encryption function for the NTRU cryptosystem — in CUDA and ran it on an Nvidia GTX 280 graphics card at a record-breaking throughput of 200,000 encryptions per second with a 256-bit security level. Aguilar et al. [1] ported their single-server lattice-based private information retrieval (PIR) scheme to run on GPUs; in all of their experiments, Aguilar et al. observed an 8x–9x improvement in throughput when compared to throughput on a system composed of similarly priced CPUs. Several other research groups have obtained promising results using GPUs to perform modular exponentiations [15, 17, 27, 28], an operation that forms the basis of many number-theoretic public key cryptosystems. Harrison and Waldron [17], for example, report a 4x throughput increase using GPUs instead of comparably priced CPUs for the special case of computing 1024-bit modular exponentiations. More recently, Neves and Araujo [28] obtained similar positive results by implementing arbitrary-precision modular exponentiations in CUDA. The present paper focuses on leveraging GPUs to do the inverse of modular exponentiation; i.e., to solve instances of one variant of the so-called discrete logarithm problem.

Perhaps the most relevant prior work along these lines is that of Bailey et al. [2]; they describe their efforts to use several clusters of conventional computers, PlayStation 3 consoles, powerful graphics cards, and FPGAs to break the Certicom ECC2K-130 challenge [13]. (A more recent paper [4] further elaborates on how the team has optimized their implementation for efficient binary field arithmetic on commodity GPUs.)

As in the present work, Bailey et al. use the parallel rho method to solve discrete logarithms; however, our efforts differ in that the ECC2K-130 challenge involves solving discrete logarithms in an elliptic curve over a binary field, whereas this work considers the problem of solving discrete logarithms in a special class of multiplicative groups. The latter setting is quite different since it involves computing modular arithmetic with a very large modulus, rather than computing binary field elliptic curve arithmetic. At the time of writing, the team's efforts to break ECC2K-130 are still underway; the interested reader should consult `http://ecc-challenge.info/` for nearly real-time progress updates. Other groups [7, 8] have obtained positive results using game consoles — specifically, the Cell-based PlayStation 3 — to solve discrete logarithms over elliptic curves using the parallel rho method, but so far no other group has reported positive results using commodity hardware to solve discrete logarithms in the setting considered in this work.

The current state of the art with respect to fast modular multiplications on GPUs appears to be Bernstein et al.'s work on 'The Billion-Mulmod-Per-Second PC' [5]. The authors of that work managed to obtain an impressive 481 million 192-bit modular multiplications per second on an Nvidia GTX 295 graphics card. The Nvidia GTX 295 has 480 cores that each run at 1.2 GHz; thus, their implementation achieves a per-core throughput of about 1 million 192-bit modular multiplications per second, which is about one modular multiplication per 1200 clock pulses on each core. The experiments considered in this paper use two Nvidia Tesla M2050 cards, which each have 448 cores that run at 1.55 GHz. Our implementation computes roughly 840 million 192-bit modular multiplications per second on each one of these cards — a per-core throughput of about 1.875 million 192-bit modular multiplications per second, which is about one modular multiplication per 830 clock pulses on each core.[4] Several other groups have also implemented efficient modular multiplication in CUDA; unfortunately, the source code for most of these implementations is not publicly available, thus preventing their numerous "speed records" from being independently replicated or verified. To avoid such shortcomings in our own work, all of our source code is open source and freely available for download from `http://crysp.uwaterloo.ca/software/`.

2 Mathematical preliminaries

This section provides a terse overview of the discrete logarithm problem and Pollard's rho method [34] for computing discrete logarithms, as well as van Oorschot and Wiener's approach [40] to parallelizing Pollard's rho. It also briefly discusses Pollard's $p-1$ factoring algorithm [33], which will be relevant to the discussion in §6.2. We begin with a formal statement of the discrete logarithm problem.

Definition 1 (Discrete logarithm problem [25, §3.6]). *Given a finite, cyclic group $\mathbb{G}$ of order n, a generator g of $\mathbb{G}$, and an arbitrary group element $\alpha \in \mathbb{G}$, the **discrete logarithm problem** is to find the unique integer exponent x in the interval $[0, n-1]$ such that $g^x = \alpha$. The exponent x is the **discrete logarithm** of α with respect to g in $\mathbb{G}$.*

Our focus in this work is on solving discrete logarithms in cyclic subgroups $\mathbb{G}$ of the multiplicative group of units modulo N. The parallel rho method has fallen out of fashion for computing discrete logarithms in this setting, with more specialized algorithms such as index calculus being preferred [25, §3.6.5]; nonetheless, in the special case where the group order n has only small factors, Pollard's rho may dramatically outperform

[4] Fundamental differences exist between the Nvidia GTX 295 graphics cards that Bernstein et al. used and the Nvidia Tesla M2050s used in this work. The latter cards are *general-purpose* GPUs, rather than standard graphics cards like those in the GTX series; as such, they possess certain characteristics that make them more suitable for general-purpose computations (for example, computing modular multiplications). Most significantly, the cards in the Tesla series have more memory than those in the GTX series do, and the Tesla M2050 can perform true 32-bit multiplication in hardware, whereas the GTX 295 uses a sequence of 24-bit multiplications to simulate hardware support for 32-bit multiplications. Thus, comparing the relative performance of the two implementations requires a more nuanced approach than simply comparing per-core throughputs; we omit a more meaningful comparison, as such a comparison is not the goal of this work.

index calculus, and it is on this special case that we focus our attention. We do note, however, that Pollard's rho method *is* currently the standard technique for computing discrete logarithms on elliptic curves [4] and many standard texts (e.g. [25, §3.6.3] or [14, §31.9]) therefore cover it in some depth. We briefly describe the algorithm below; however, the interested reader is encouraged to consult one of the aforementioned texts for a more thorough description of the algorithm and analysis of its runtime.

Pollard's rho method. Pollard's rho algorithm is essentially just a clever way to exploit the well-known birthday paradox. In a nutshell, the birthday paradox tells us that, on average, one needs only select about $\sqrt{\pi n/2}$ random elements from a set of n alternatives (with replacement) before encountering a collision (wherein a previously selected element is selected again). This fact allows one to compute the discrete logarithm x of $h \in \mathbb{G}$ with respect to g by repeatedly selecting random exponents $a_i, b_i \in_R [0, n-1]$ to obtain random group elements $g^{a_i}h^{b_i} \in \mathbb{G}$. After sufficiently many such random selections, a collision will occur; in particular, the process eventually yields two triples $(a_{i_1}, b_{i_1}, g^{a_{i_1}}h^{b_{i_1}})$ and $(a_{i_2}, b_{i_2}, g^{a_{i_2}}h^{b_{i_2}})$ such that $g^{a_{i_1}}h^{b_{i_1}} = g^{a_{i_2}}h^{b_{i_2}}$ and $b_{i_1} \not\equiv b_{i_2} \bmod n$, whence it follows that $a_{i_1} + b_{i_1}x \equiv a_{i_2} + b_{i_2}x \bmod n$, and therefore $x = (a_2 - a_1)(b_1 - b_2)^{-1} \bmod n$. The birthday paradox tells us that an expected number of $\sqrt{\pi n/2}$ random selections suffice to find such a collision; thus, the natural algorithmic instantiation of this process solves the discrete logarithm problem with an expected runtime in $\Theta(\sqrt{n})$, and uses $\Theta(\sqrt{n})$ storage.

Pollard's big idea was to modify the above observation to find the collisions without having to store $\Theta(\sqrt{n})$ such triples. To do this, he proposed using a function $f : \mathbb{G} \to \mathbb{G}$, called an *iteration function*, that is chosen so that 1) it is efficient to compute, 2) it behaves heuristically like a randomly selected mapping from $\mathbb{G}$ to itself, and 3) it maps a group element $g^{a_1}h^{b_1}$ to $g^{a_2}h^{b_2}$ in such a way that a_2 and b_2 are easy to compute from a_1 and b_1. The actual instantiation for f that Pollard proposed is

$$f(x) = \begin{cases} h\,x & \text{if } 1 \leq x < \frac{N}{3}, \\ x^2 & \text{if } \frac{N}{3} \leq x < \frac{2N}{3}, \text{ and} \\ g\,x & \text{if } \frac{2N}{3} \leq x < N, \end{cases} \tag{1}$$

which is essentially the same function that we use in our implementation.[5] The algorithm then proceeds by starting with a random group element $g^{a_0}h^{b_0}$ and iteratively applying f to select the subsequent "random" group elements. It is easy to see that when a collision eventually occurs (after an expected $\sqrt{\pi n/2}$ iterations, assuming perfectly random behaviour of f), the subsequent iterations of the process form a cycle, which can be detected using a cycle finding algorithm such as that of Floyd [16] or Brent [10]. Moreover, the group element $g^{a_i}h^{b_i} = f^{(i)}(g^{a_0}h^{b_0})$, where $f^{(i)}(\cdot)$ denotes that f is iteratively applied to the operand i times, is easily expressed in terms of g and h, so once a cycle (hence, collision) is found, one can use it to compute the discrete logarithm as above.

The parallel rho method. Regrettably, Pollard's rho method as presented above does not parallelize well. The reason for this is that iterative application of f is an inherently serial process that each thread of execution must perform independently. It turns out that invoking the procedure Ψ times in parallel yields sublinear expected speedups (in particular, the expected speedup is proportional to $\sqrt{\Psi}$ rather than Ψ). Van Oorschot and Wiener [40] proposed an ingenious way to bypass this limitation using the notion of *distinguished points*. (A distinguished point is simply some group element that has an easily testable property, such as a certain

[5] Teske [39] subsequently proposed a better choice for f, which can reportedly reduce number of iterations by $\approx 20\%$ on average. It is possible that switching to Teske's iteration function would lead to speedups in our implementation, although we suspect that the overhead associated with evaluating the more complicated function on a GPU would increase the cost of an iteration so much as to negate any purported performance increases. Nonetheless, it would be interesting and worthwhile to experiment with Teske's alternative iteration function (or some middle ground between Teske's function and Pollard's function) as a direction for future work.

number of trailing zeros in its binary representation.) Under their regime, one thread acts as a server and Ψ threads act as clients; each client thread starts the iteration process at a different random group element with known representation in terms of g and h and iterates until it hits a distinguished point. When a client thread encounters its first distinguished point $g^{a_i}h^{b_i} = f^{(i)}(g^{a_0}h^{b_0})$, it sends the triple $(a_i, b_i, g^{a_i}h^{b_i})$ to the server thread and starts the iteration process anew with a fresh random group element. Upon receiving a triple $(a_{i_1}, b_{i_1}, g^{a_{i_1}}h^{b_{i_1}})$ from some client such that $g^{a_{i_1}}h^{b_{i_1}} = g^{a_{i_2}}h^{b_{i_2}}$ and $b_{i_1} \not\equiv b_{i_2} \bmod n$ for some previously received triple $(a_{i_2}, b_{i_2}, g^{a_{i_2}}h^{b_{i_2}})$, the server computes the discrete logarithm just as it did before. By the same observation used above, if two threads ever encounter a collision (be it at a distinguished point or not), then all subsequent iterations of those two threads necessarily follow identical trails; thus, the next distinguished point that either thread encounters is also necessarily a collision. In particular, each of the Ψ threads is searching for collisions with any group element encountered by any other thread, and the expected speedup becomes linear in Ψ.

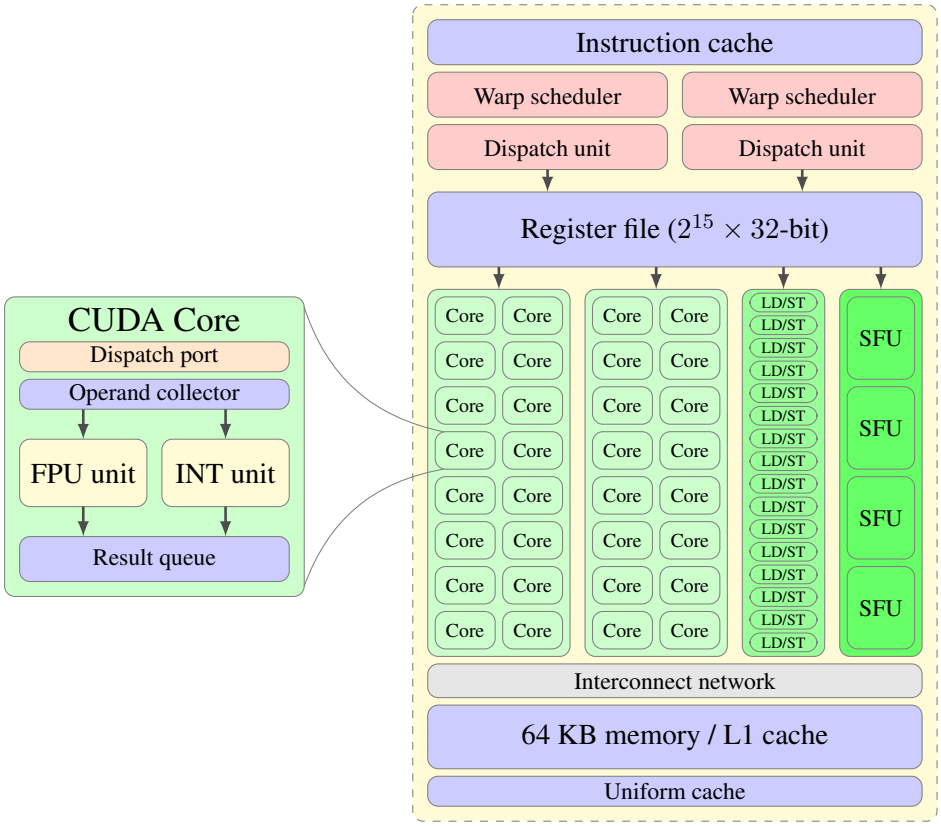
Pollard's $p - 1$ method. Pollard's $p - 1$ factoring algorithm is a special-purpose factoring algorithm that uses Fermat's Little Theorem to find certain, special factors of an integer. The key observation behind the technique is that working in the multiplicative group of units modulo n is equivalent to working in the multiplicative groups of units modulo each of n's prime-power factors. Moreover, Fermat's Little Theorem says that if $\gcd(g, p) = 1$ for a prime p, then $g^{k(p-1)} \equiv 1 \bmod p$ for all k, hence $p \mid \gcd(g^{k(p-1)} - 1, n)$. This suggests the following algorithm for finding prime factors p of n, subject to the condition that all of the prime factors of $p - 1$ are bounded above by some *smoothness* bound B (in such a case, $p - 1$ is called B-smooth): Fix a positive integer B and compute $x = \prod q^{\lfloor \log_q B \rfloor}$, where the product is taken over all primes q less than B, then compute and return $d = \gcd(g^x - 1, n)$. If $1 < d < n$, then d is a product of all prime factors p of n for which $p - 1$ is B-smooth. A simple modification involves progressively increasing B in the computation and computing the gcd at each step to recover the individual prime factors (rather than their product). Note that this procedure requires between $B/\ln 2$ and $1.5B/\ln 2$ modular multiplications, depending on whether modular exponentiations are performed with naive square-and-multiply or a somewhat more efficient algorithm.

3 GPU programming with 'C for CUDA' and PTX

To meet the demands of increasing screen resolutions, frame rates, and scene complexity seen in today's video games, modern graphics cards have evolved into extremely powerful computing platforms that leverage a large degree of parallelism compared to regular CPUs. This has led to much interest in harnessing the power of GPUs as massively parallel co-processors working alongside regular CPUs in applications outside of graphics processing. To facilitate such uses, Nvidia has developed the *Compute Unified Device Architecture* (CUDA) parallel computing platform and programming model and the *Parallel Thread eXecution* (PTX) instruction set architecture, which together form the basis for their GeForce (for consumer PCs), Quadro (for professional workstations), and Tesla (for high-performance general-purpose computing) lines of GPU devices [29].

Nvidia GPUs. The architecture of Nvidia GPU devices exhibit fundamental differences from most CPU-based systems, and effectively utilizing their computational power necessitates an understanding of these differences. §1.1.1 of Nvidia's CUDA C Best Practices Guide [31] describes the most important differences, which mostly pertain to how GPU devices handle threading and memory access; for completeness, we summarize the key architectural features of CUDA GPU devices.

A typical Nvidia GPU contains several *streaming multiprocessors* (SMPs), each of which consists of several *streaming processors* (SPs) and *special function units* (SFUs), an instruction decoder, and some



(a) An exploded view of a single CUDA core.

(b) The internal structure of one SMP in the Fermi architecture.

Figure 1: The internal structure of a streaming multiprocessor (SMP) in the Fermi architecture and an exploded view of a single streaming processor (SP), or "CUDA core". This diagram is adapted from Nvidia's documentation [30].

shared memory. The SPs are known colloquially as *CUDA cores*. The Tesla M2050 cards that we use in our experiments are based on the Nvidia Fermi architecture, which has 32 CUDA cores and 4 SFUs per SMP (the M2050 itself is comprised of 14 SMPs). Each SMP in the GPU is capable of executing a single instruction at a time, which means that the 32 CUDA cores in that SMP must each execute the same instruction simultaneously, albeit on different data (this is called *single instruction/multiple data* or SIMD). Nvidia calls a bundle of 32 threads executing in parallel on an SMP a *warp*, which constitutes the smallest executable unit of parallelism on a CUDA device. All Nvidia GPUs can support at least 24 active warps (768 active threads) per SMP — and some higher-end GPUs can support 32 active warps per SMP — where each warp has its own set of registers; once the GPU allocates registers to a warp, those registers stay allocated to that warp until it finishes execution. This makes threads on the GPU extremely lightweight compared to their counterparts on a regular CPU. An application can queue up thousands of threads and when the GPU must wait on one warp of threads, it simply begins executing work (at the next clock pulse) on another warp of threads, with no intervention from the host device and no swapping of register state. However, getting good performance out of threads in CUDA still requires the software developer to keep several caveats in mind. For example, if there is a conditional branch and some threads in a warp take this branch while others do not (called *warp divergence*), then the other threads will just idle until the branch is complete and they

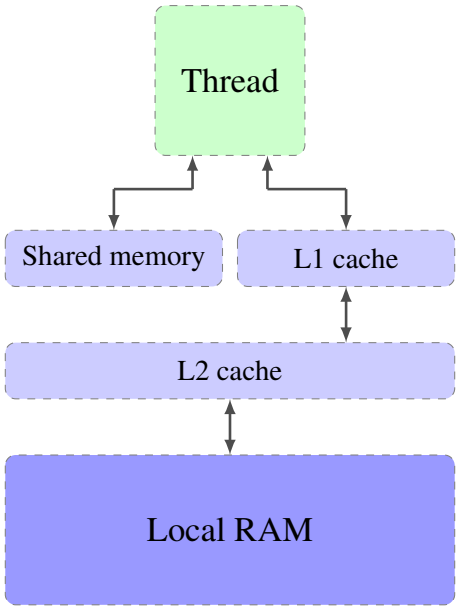


Figure 2: The memory hierarchy in CUDA GPU devices based on the Nvidia Fermi architecture. This diagram is adapted from Nvidia's documentation [30].

all converge back together on a common instruction. The situation is even worse when two or more threads from a warp each take a a different conditional branch: only one branch is executed at a time, and the overall execution time becomes the *sum* of the execution times of each branch taken (rather than the maximum execution time across all branches, as one might intuitively expect).

Figure 1 illustrates the structure of SMPs in the Nvidia Fermi architecture; 1(a) gives an exploded view of a single CUDA core within the SMP, while 1(b) shows the internal structure of the entire SMP. Each CUDA core resembles a regular CPU core but is much simpler, reflecting its heritage as a pixel shader. It has a pipelined floating-point unit (FPU), a pipelined integer unit (INT), some logic for dispatching instructions and operands to these units, and a queue for holding results, but it lacks its own general-purpose register file, L1 cache, function units for each data type, and load/store units for retrieving and saving data.

The other important difference between CUDA GPU devices and CPU-based systems is the memory hierarchy. Memory on the GPU is segmented, both physically and virtually, into several different types, each of which has its own special purpose and performance characteristics. For one thing, the GPU has a very large frame buffer, which Nvidia calls *local RAM*, on which application developers can store their data; the local RAM is further subdivided into read-write *global memory*, read-only *constant memory*, and read-only *texture memory*. There is also a small *shared memory* and some L1 cache that is local to each CUDA core, and an L2 cache that all SMPs on the GPU device share. Figure 2 illustrates the memory hierarchy in CUDA GPU devices based on the Nvidia Fermi architecture. Carefully managing these different types of memory is important, as the latencies experienced when a thread reads from or writes to memory depends on that memory's proximity to the CUDA core running the thread. In the extreme case of fetching data from the host device's memory, these data must travel along the PCIe bus to get to the GPU device, thus incurring extremely high latencies and throughputs that are an order of magnitude or more slower than fetching data from memory on the device. According to Nvidia's documentation [30], access to local RAM requires 200 – 300 clock cycles, while access to on-chip memory (registers, shared memory, L1 cache) requires only one clock cycle. Thus, when reading blocks from (or writing blocks to) local RAM or memory on the host device, it is imperative to use a *coalesced* access pattern, wherein the blocks occupy consecutive memory addresses;

this allows CUDA to batch many small transfers into a single larger transfer. To facilitate effective use of shared memory, cache, and local RAM, CUDA-enabled programming languages such as 'C for CUDA' (see below) include new variable-type qualifiers (`__device__`, `__constant__`, and `__shared__`), allowing programmers to specify where to store the data referenced by a variable.

C for CUDA. Software developers can use CUDA-enabled variants of several industry-standard programming languages to access the virtual instruction set and memory of the parallel computing elements in CUDA GPUs. The most common CUDA-enabled language, and the one used in this work, is a variant of C with some additional Nvidia extensions called 'C for CUDA'. CUDA applications are partitioned into completely encapsulated GPU *kernels* with C statements interleaved; the kernels are executed on the GPU and the C statements on the host CPU. Function-type qualifiers (analogous to the aforementioned variable-type qualifiers) specify where a function should run: the `__host__` function-type qualifier specifies that the host device both invokes and runs the function; the `__global__` function-type qualifier specifies that function is a kernel, meaning that the host device invokes the function, but it runs on the GPU device; and the `__device__` function-type qualifier specifies that code on the GPU device invokes the function and the function runs on the GPU. CUDA imposes a two-tier hierarchical structure on its threads, which the application developer specifies using a new $\lll \cdots, \cdots \ggg$ syntax; groups of threads form *thread blocks*, and groups of thread blocks form a *thread grid*. For example, an invocation of the form `kernel`$\lll$`1, N`$\ggg(\cdots)$; executes `kernel` N times in parallel by N different threads, where each of these threads has a unique *thread ID* that is accessible within the kernel through the built-in `threadIdx` variable. Threads within a block always run on a single CUDA core, which ensures that synchronization and cooperation between threads within a block is inexpensive, whereas different thread blocks may run on different MPs and therefore run independently. This design simplifies scaling, since it enables GPUs with more SMPs to process more blocks in parallel without requiring changes to the program or kernel configuration. Nvidia's `nvcc` compiler translates C for CUDA device source code into device-independent PTX code.

The PTX ISA. PTX is a device-independent pseudo-assembly language for CUDA GPU devices. It provides a means for software developers to make fine-grained optimizations to their code before the `ptxas` compiler converts it into the final device-specific binary file, which is later loaded and executed on the GPU. PTX exposes several useful instructions that the `nvcc` compiler fails to utilize; most relevant to our implementation of modular multiplication are the instructions for add-with-carry-in and optional carry-out (`ADDC`$\{$`.cc`$\}$), subtract-with-borrow-in and optional borrow-out (`SUBC`$\{$`.cc`$\}$), and the fused integer multiply-and-add instruction (`MAD`$\{$`.hi`, `.lo`, `.wide`$\}$). The latter instruction enables our implementation to multiply two 32-bit unsigned integers, and then add a third 64-bit integer, placing the full 64-bit result in a 64-bit register. The PTX ISA also gives developers some control over the allocation and use of registers, which is helpful in minimizing unnecessary copying of register values when the input to one instruction is the output of some earlier instruction.

4 Arbitrary-precision modular multiplication and parallel rho on GPUs

Montgomery multiplication. Our implementation of arbitrary-precision modular multiplication uses the well-known Montgomery multiplication and reduction techniques [26]. We briefly recall how ordinary Montgomery multiplication and reduction work, before discussing the coarsely-integrated operand scanning (CIOS) algorithm for modular Montgomery multiplication, which is the variant that we found to give the best performance in our CUDA implementation.

Let N be a fixed, odd k-bit integer, let $R = 2^k$ (so $2^{k-1} < N < 2^k$ and $R > N$ with $\gcd(N, R) = 1$) and let x and y be two integers in the range $[0, N-1]$. Montgomery multiplication allows for the computation

of $xy \bmod N$ without explicitly carrying out the costly classical modular reduction step. To do this, the multiplicands x and y must first be "Montgomerized" to N-*residues*: $\tilde{x} = xR \bmod N$ and $\tilde{y} = yR \bmod N$. The Montgomery multiplication algorithm computes the N-residue $\tilde{z}$ of $z = xy$ from $\tilde{x}$ and $\tilde{y}$, which turns out to be much faster than computing $xy \bmod N$ directly from x and y (because modular reduction and division by $R = 2^k$ in binary reduces to truncation and rightward bit-shifts). Define R' ($= R^{-1} \bmod N$) and N' to be integers that satisfy Bézout's identity [20, §1.2], $R \cdot R' + N \cdot N' = 1$; these values are easily computed with the extended Euclidean algorithm [25, §2.4]. The Montgomery product of $\tilde{x}$ and $\tilde{y}$ is

$$\begin{aligned}\tilde{z} &= \tilde{x}\tilde{y}R' \bmod N \\ &= (xR)(yR)R^{-1} \bmod N \\ &= (xy)R \bmod N.\end{aligned}$$

The cost savings come from the observation that one can evaluate the above expression using the following procedure. Compute $t = \tilde{x}\tilde{y}$, then $u = (t + (tN' \bmod R)N)/R$; if $u > N$ then return $u - N$, else return u. The desired product z is then obtained by computing $z = \tilde{z}R' \bmod N$. Note that the Montgomery method incurs some overhead in computing R' and N', and that conversion to and from N-residues each require a reduction modulo N. However, if an algorithm computes many modular multiplications with respect to the same modulus to produce only a small set of outputs (such as modular exponentiation, or — in our case — the iterative collision search in parallel rho), the more efficient Montgomery multiplication step results in significant cost savings.

Coarsely-integrated operand scanning. Several alternative algorithms exist for computing the Montgomery multiplication step. In our implementation, we use the *coarsely-integrated operand scanning* (CIOS) method due to Koç et al. [21]. The algorithm is *integrated* because it alternates between multiplication and reduction steps in the computation (that is, it integrates the two procedures into one). The *coarsely-* prefix refers to the frequency with which the algorithm alternates between the two steps; CIOS alternates after processing an array of words, which is in contrast to a *finely-integrated* method, which alternates after processing a single word. Finally, *operand scanning* refers to the fact that the outer loop in the algorithm is over the words of the operands (an alternative approach is *product scanning*, wherein the outer loop is over the words of the product itself). The reader should consult Koç et al.'s paper [21, §5] for full details of the CIOS algorithm.

Interestingly, Bernstein et al. report that "schoolbook" (Montgomery) multiplication gave the best performance in their CUDA implementation of 192-bit modular multiplication [5]; however, our experiments indicate that the CIOS algorithm gives superior performance. This is likely due to its smaller auxiliary storage requirements (the integrated nature of the CIOS method means that it requires just $s+2$ words of auxiliary storage for an s-word modulus, contrasted with $2s + 2$ words for the schoolbook method; this enables more threads to run in parallel on the GPU without exhausting the register pool). Neves and Araujo [28] suggest that the *finely-integrated product scanning* (FIPS) Montgomery multiplication method (also from Koç et al. [21]) yields better performance on GPUs than CIOS does, since each word of the final product can be calculated individually in parallel (whereas the long carry chains in CIOS can make instruction-level parallelism difficult). However, as Bernstein et al. have pointed out [5], using a single thread to compute an entire s-word multiplication leads to improved compute-to-memory-access ratios and eliminates synchronization overhead, thus resulting in better overall performance.

Implementing CIOS with CUDA and PTX. Our implementation of CIOS Montgomery multiplication follows the algorithm given by Koç et al. in §5 of their paper almost exactly, including the suggested improvement for integrating the shifting into the reduction. Our initial implementation looked much like the pseudocode in that paper; however, its performance was underwhelming, and a mysterious bug caused it to

```
// x <- x - y; x and y are both WORDS words long
__device__ void _sub(uint32_t *x, const uint32_t *y)
{
    asm("sub.cc.u32 %0, %1, %2;"
         :  "=r"(x[0]) :  "r"(x[0]), "r"(y[0]));
    for (int i = 1; i < WORDS; i++)
    {
        asm("subc.cc.u32 %0, %1, %2;"
             :  "=r"(x[i]) :  "r"(x[i]), "r"(y[i]));
    }
    asm("subc.u32 %0, %1, %2;"
         :  "=r"(x[WORDS]) : "r"(x[WORDS]), "r"(y[WORDS]));
}
```

Figure 3: PTX-based implementation of arbitrary-precision subtraction.

```
// return x * y + c
static inline __device__ uint64_t mad_u32(
      const uint32_t x, const uint32_t y, const uint64_t c)
{
    uint64_t out;
    asm("mad.wide.u32 %0, %1, %2, %3;"
             :  "=l"(out) :  "r"(x), "r"(y), "l"(c));
    return out;
}
```

Figure 4: PTX-based implementation of fused multiply-and-add.

produce random results in some invocations. (Rather frustratingly, the exact same code *always* succeeded when we ran it in the now deprecated CUDA device emulator.) Eventually, we traced the root of the problem to our use of `memset` to zero the auxiliary array; a race condition was occurring wherein subsequent lines of code sometimes used that memory before the GPU had finished zeroing it. To avoid this, we factored the first iteration out of the outer CIOS loop and modified it to work regardless of the state of the auxiliary array. This modification had the fringe benefit of making the code slightly faster (by entirely avoiding the call to `memset` and several unnecessary additions of 0). Likewise, we factored the last iteration of the outer CIOS loop to place the result directly into the return value, thus avoiding an unnecessary copy at the end of the algorithm.

By far the greatest performance gains occurred when we replaced arithmetic that used standard C-like syntax with inline PTX assembly. For example, rewriting our arbitrary-precision subtraction code to use the subtract-with-borrow-in and optional borrow-out PTX instruction shaved several nanoseconds off the average execution time of each modular multiplication. Figure 3 shows the optimized arbitrary-precision subtraction function; note that in our final implementation, the `for` loop is unrolled to save a few more clock cycles. Similarly, we used PTX's fused multiply-and-add instruction to get significant speedups in the inner CIOS loops (see Figure 4).

Manual loop unrolling proved to be another crucial optimization; at first glance, this optimization appears to be somewhat incompatible with implementing *arbitrary*-precision arithmetic. We solved this by writing a simple Perl script that generates completely unrolled PTX assembly for a given modulus size, which we then link into the binary at compile time. The output of the Perl script also operates entirely on registers instead

of arrays, which is much faster in CUDA, and not possible using an ordinary `for` loop.

Parallel rho with GPUs. We leverage our arbitrary-precision modular multiplication code to implement the parallel rho method. We run the "server thread" on the CPU and each of the "client threads" on one of two Nvidia Tesla M2050 GPU cards (cf. §2). Using Nvidia's CUDA Occupancy Calculator[6] and some experimentation, we found that launching each kernel with 16 warps $= 512$ threads per thread block and 50 total thread blocks per card (which is 25,600 threads per thread grid) provides reasonably good occupancy when our implementation is run to solve discrete logarithms with a 768-bit modulus on our Tesla M2050 cards.

Because there is no lightweight way to interrupt a kernel once it is invoked, we instead have each thread perform some fixed number of iterations before it returns; in particular, each of the 25,600 threads performs 1000 iterations. For the iteration function, we use a "Montgomerized" version of Pollard's original iteration function (Equation (1)), but we note that our use of the iteration function differs somewhat from its regular usage in van Oorschot and Wiener's parallel rho method. When a client thread encounters a distinguished point, it simply outputs the triple $(a_i, b_i, g^{a_i}h^{b_i})$ to the server thread on the host and then continues iterating from that element rather than starting over from a new random group element. We do this to avoid having to initialize a new random element from the server between invocations, and to avoid having all of the threads in the same warp stall for the remainder of the current kernel invocation.[7] If the server does not receive any collisions during a kernel invocation, it simply relaunches the kernel without having to reinitialize or modify any memory on the GPU, and the threads continue iterating from where they left off. This keeps overhead low, but it also means that some client threads could get caught in cycles that do not contain any distinguished points (using a cycle-finding algorithm to detect this would be detrimental to overall performance). However, the expected cycle size is about $\sqrt{n}$, where n is the order of the group $\mathbb{G}$; thus, if the frequency F of distinguished points is such that $F \gg \frac{1}{\sqrt{n}}$, then the probability that this happens is low enough that we can safely ignore it. We define our distinguished points to be elements of $x \in \mathbb{G}$ such that the binary representation of the N-residue of x (i.e., of the Montgomery representation of x) has at least ten trailing zeros, so that about one in every 1024 iterations yields a distinguished point. Since each thread performs 1000 iterations per invocation, the server thread receives about $2^{14.6}$ distinguished points — or one per client thread — each time it invokes the kernel. In the case that $n \leq 2^{20}$ (so that $\frac{1}{1024} \gg \frac{1}{\sqrt{n}}$ does not hold), we change the definition of distinguished points to make them more numerous. In each kernel invocation, the client threads perform a combined total of about $2^{24.6}$ multiplications; thus, for n up to about 2^{50} a single kernel invocation usually suffices to solve the discrete logarithm.

5 Performance evaluation

For our performance benchmarks, we used a server running an Intel Xeon E5620 quad core processor (2.4 GHz) and 2×4 GB of DDR3-1333 RAM, which is equipped with $2 \times$ Tesla M2050 GPU cards. Table 1 below summarizes the per-card performance of our arbitrary-precision modular multiplication implementation; that is, it displays the number of modular multiplications with a k-bit modulus that our implementation can compute on a single Tesla M2050 card for various choices of k, as well as the (amortized) time required

[6] `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`

[7] Of course, this is not the only strategy for efficient handling of distinguished points in a GPU environment. For instance, one of the anonymous SHARCS 2012 reviewers points out that "[there] are good reasons not to continue walking from a distinguished point: One can skip all bookkeeping for counting the a and b and instead store only the starting value together with the distinguished point found. If two points collide the server can redo the computations, this time keeping track of the coefficients. For that to work each walk should be reasonably short. See Bernstein et al. [4] and Bernstein, Lange and Schwabe [6], for details on how to handle this in a SIMD environment".

to do each modular multiplication. We point out that as k increases, the execution time increases not only because it naturally takes longer to multiply larger numbers, but also because larger moduli use more registers, and so each core can compute fewer multiplications in parallel.

Table 1: Number of k-bit modular multiplications per second and (amortized) time required per k-bit modular multiplication on each Tesla M2050 GPU card, for various k.

Bit length of modulus	**Modmults per thread**	**Number of threads**	**Number of trials**	**Time per trial ± std dev**	**Amortized time per modmult**	**Modmults per second**
192	100,000	256,000	100	30.538 s ± 4 ms	1.19 ns	≈ 840,336,000
256	100,000	256,000	100	50.916 s ± 5 ms	1.98 ns	≈ 505,050,000
512	100,000	256,000	100	186.969 s ± 4 ms	7.30 ns	≈ 136,986,000
768	100,000	256,000	100	492.6 s ± 200 ms	19.24 ns	≈ 51,975,000
1024	100,000	256,000	100	2304.5 s ± 300 ms	90.02 ns	≈ 11,108,000

Table 2 shows the average time required to compute a discrete logarithm with a 1536-bit RSA modulus $N = pq$ such that $p - 1$ and $q - 1$ are both 768-bit B-smooth integers, for various choices of B. Our implementation uses the approach of Pohlig and Hellman [32] to solve the discrete logarithm independently modulo p and modulo q using the parallel rho method, and then combines the results via the Chinese Remainder Theorem [25, §2.4.3] to get the final discrete logarithm modulo N. In particular, each of the discrete logarithm computations in Table 2 consists of $2 \cdot \lceil 768/\lg B \rceil$ smaller discrete logarithm computations, each at a cost proportional to $\sqrt{B}$. We therefore expect the total cost of the larger discrete logarithm computation to be proportional to $\sqrt{B}/\lg B = B^{0.5-(\lg\lg B/\lg B)}$. When $B \approx 2^{53}$, as in Table 2, we have that $\lg\lg B/\lg B \approx (\lg 53)/53 \approx 0.108$, so that the total running time should be near $c \cdot B^{0.39}$ for some constant of proportionality c.

Figure 5 plots data from Table 2. The exponent on the trend line is slightly less than the expected value of 0.39 because of overhead that is more significant at lower smoothness bounds. One source of overhead is the Chinese remaindering step that we perform after computing the discrete logarithm modulo p and modulo q. Another source of overhead comes from processing "remainder" submoduli of $p - 1$ and $q - 1$: to generate N we choose all but one prime factor of $p - 1$ and $q - 1$ to be k_B bits long, and the last prime factor is about $768 \bmod k_B$ bits. Since our implementation is not optimized for these smaller submoduli, they tend to introduce some additional, nearly constant overhead to the computation time.

Table 2: Time to compute discrete logarithms modulo a product $N = pq$ of two 768-bit primes, such that $\varphi(N)$ is B-smooth, for various choices of B. The discrete logarithm is solved independently modulo p and q using Pohlig-Hellman [32] and parallel rho, and the results are combined via the Chinese Remainder Theorem [25, §2.4.3]. The runtime reported in the final column is the time required to compute all three steps.

Bit length of modulus	**Smoothness of group order**	**Number of trials**	**Time to compute discrete logarithm**
$1536 = 2 \times 768$	2^{48}	100	23 s ± 1 s
$1536 = 2 \times 768$	2^{50}	100	32 s ± 2 s
$1536 = 2 \times 768$	2^{51}	100	41 s ± 3 s
$1536 = 2 \times 768$	2^{52}	100	49 s ± 4 s
$1536 = 2 \times 768$	2^{53}	100	63 s ± 5 s
$1536 = 2 \times 768$	2^{54}	100	85 s ± 7 s
$1536 = 2 \times 768$	2^{55}	100	110 s ± 10 s
$1536 = 2 \times 768$	2^{56}	100	140 s ± 20 s
$1536 = 2 \times 768$	2^{58}	100	270 s ± 30 s

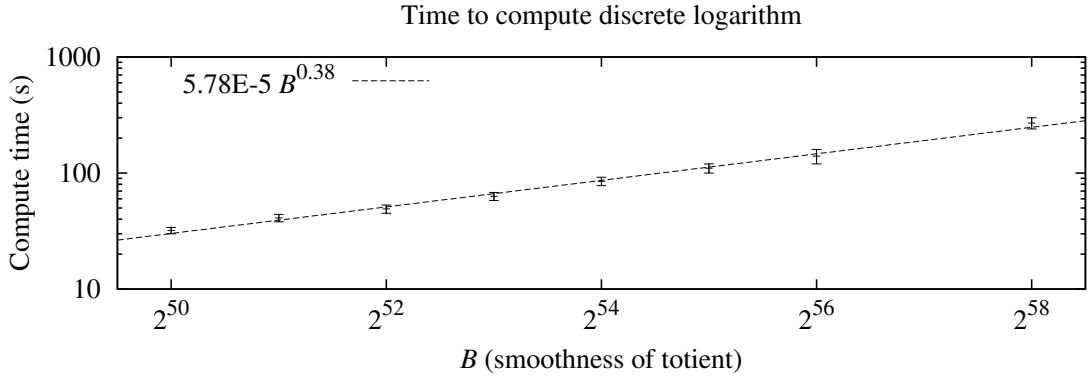


Figure 5: Plot of data from Table 2. The exponent on the trend line is slightly less than 0.39 because of overhead that is more significant at lower smoothness bounds.

By extrapolating to $B = 2^{80}$, we see that it should be feasible to solve discrete logarithms in groups whose order is 2^{80}-smooth in approximately 23 hours.

6 Analysis

6.1 Implications to existing cryptosystems

The idea of exploiting the smoothness of a group's order for attacking cryptosystems whose security relies on the hardness of factoring or computing discrete logarithms is not new. In fact, many cryptographers advocate the use of *safe primes* (primes of the form $p = 2q + 1$ for another prime q) specifically to avoid such attacks (since $p - 1 = 2q$ is not B-smooth for any $B \ll p$). However, the celebrated *elliptic curve factorization method* [23] (ECM) renders these defenses ineffective by considering random elliptic curves over $\mathbb{Z}_p$, which, by Hasse's theorem [38, §V], have orders that are (essentially randomly) distributed between $p + 1 - 2\sqrt{p}$ and $p + 1 + 2\sqrt{p}$. Thus, one can argue that, due to ECM, explicitly choosing safe primes is not helpful, since

to ensure security we must already assume that $p-1$ is non-smooth for a random prime p. Pomerance and Shparlinski [36] study the distribution of smooth integers, and derive rigorous upper bounds on the number of k-bit prime numbers p for which $p-1$ is smooth or has a large smooth factor. Their findings suggest that a *randomly selected*, cryptographically large number is not "sufficiently smooth" to make smoothness-based attacks feasible.

We point out, however, that a bad actor could choose the modulus for a discrete-logarithm-based cryptosystem with malice, inserting a trapdoor for his own use. Our experiments in §5 indicate that computing discrete logarithms in groups of B-smooth order, for B up to at least $\approx 2^{80}$, is entirely realistic with under 24 hours of computation on readily available commodity hardware. For a prime modulus N such that $N-1$ is 2^{80}-smooth, one could use ECM (or some other related technique) to factor $N-1$ and thus learn about its insecurity (although this would require substantial computational effort on the part of the would-be victim). On the other hand, detecting insecure *composite* moduli N is not so simple, since even determining the value of $\varphi(N)$ is equivalent to factoring N [25, §8.2.2]. Our analysis in the next subsection suggests that the massive parallelism afforded by GPUs only helps to widen the gap between the feasibility and detectability of such attacks. Fortunately, most cryptosystems that base their security on the difficulty of computing discrete logarithms work over prime moduli; however, below we point out a concrete — and realistic — attack on zero-knowledge range proofs, which uses a difficult-to-detect composite modulus with smooth totient.

An attack on zero-knowledge 'range proofs'. At Eurocrypt 2000, Boudot proposed a novel zero-knowledge proof that allows a prover to convince a verifier that a committed value is in a specific interval [9]. Boudot's *range proof* relies on Lagrange's four-square theorem [37], which states that an integer can be expressed as a sum of (at most) four squares if and only if it is nonnegative. Suppose that a prover wishes to convince a verifier that a commitment, say $C = g^x \bmod p$, is to a value x in the interval $[a, b]$. To do this, the prover and verifier each compute $C_a = C/g^a = g^{x-a} \bmod p$ and $C_b = g^b/C = g^{b-x} \bmod p$, and then the prover engages the verifier in a zero-knowledge proof of knowledge of two tuples of integers (c, d, e, f) and (h, i, j, k) such that $C_a = g^{c^2+d^2+e^2+f^2} \bmod p$ and $C_b = g^{h^2+i^2+j^2+k^2} \bmod p$. Of course, for soundness the proof assumes that the prover does not know the group order, since otherwise the prover could simply find, say, (c, d, e, f) such that $c^2 + d^2 + e^2 + f^2 = x - a + \varphi(p)$ and thereby fool the prover. Therefore, when p is prime (which it usually is), the verifier chooses a composite modulus N for which the prover does not know the factorization, then the prover commits to x modulo N and proves in zero knowledge that this new commitment is to the same x as the original commitment. Finally, the prover and verifier do the above range proof using the commitment modulo N. If the verifier chooses a modulus N with smooth totient (thus violating one of the assumptions needed to prove computational zero-knowledge), then when the prover commits to his secret x in the group modulo N, the verifier can compute the discrete logarithm to learn x.[8]

6.2 Trapdoor discrete logarithm groups

In earlier work [18], we discussed using a CPU-based implementation of parallel rho to construct *trapdoor discrete logarithm groups*; that is, groups in which computing discrete logarithms is easy for anyone in possession of a special trapdoor key, but cryptographically hard for everybody else. The GPU-based implementation of parallel rho that we consider in this work allows for the same construction, but with a considerably improved margin of security.

Construction. The idea behind the trapdoor discrete logarithm group construction is to work in the multiplicative group of units modulo a k_N-bit RSA modulus $N = pq$ such that $p \approx q$, and both $p-1$ and

[8] Of course, one can thwart this attack by having the verifier prove to the prover that the composite modulus is the product of two safe primes, for example by using the technique of Camenisch and Michels [12].

$q-1$ are products of distinct k_B-bit primes; here k_N and k_B are carefully selected parameters. The public key is N and the private (trapdoor) key is the factorization of $\varphi(N)$ into k_B-bit primes. Computing discrete logarithms with knowledge of the trapdoor key requires $\Theta\left(\frac{k_N}{k_B} \cdot 2^{k_B/2}\right)$ highly parallelizable work, whereas the most efficient way to compute discrete logarithms *without* knowledge of the trapdoor key seems to be factoring N into p and q, and then factoring $p-1$ and $q-1$ to recover the trapdoor key. In our original application [18], we actually wanted trapdoor discrete logarithm groups such that computing the discrete logarithm with the trapdoor key is *tunably costly*, but feasible; other applications might wish to set the cost as low as possible subject to the construction staying secure.

Security analysis. Using the parallel rho method, the expected number of k_N-bit modular multiplications needed to compute the discrete logarithm in such a trapdoor group is $c \cdot \left(\frac{k_N}{k_B}\right) \cdot 2^{k_B/2}$, for some constant of proportionality c. (Note that these multiplications are almost completely parallelizable.) Given a parallelism factor of Ψ cores, this takes about

$$\frac{k_N}{k_B} \cdot \frac{c \cdot 2^{k_B/2}}{\Psi \cdot \mu} \text{ seconds,} \tag{2}$$

where μ is the number of multiplications modulo an $(k_N/2)$-bit modulus that are computable per core-second. Thus, to tune the parameters such that discrete logarithm computations require a specific time Γ on average, we solve for k_B in the expression

$$\frac{2^{k_B/2}}{k_B} \approx \frac{\Gamma \cdot \mu \cdot \Psi}{k_N \cdot c}. \tag{3}$$

Through experimentation, we observe that it takes the "progressively increase B" variant of Pollard's $p-1$ method at least about $\frac{3}{5} \cdot 2^{k_B}$ modular multiplications with a k_N-bit modulus to factor N. (Note that this is fewer multiplications than the estimate given in §2, since we only need to consider those primes q such that $2^{k_B} < q < 2^{k_B+1}$, and since we can assume each prime has multiplicity one in $\varphi(N)$, given our prior knowledge about N.) We stress that — in contrast to the case of *general* exponentiation, which can potentially benefit from some parallelism — the adversary must perform these multiplications in a sequential manner [11]; even with a very large degree of parallelism, only a very small speedup is obtainable.[9] It may be possible to parallelize the "non-progressive" variant of Pollard's $p-1$; however, this variant will output the *product* of all prime factors p of N such that $p-1$ is B-smooth, which in the trapdoor discrete logarithm case is just N itself. Therefore, when $k_B \ll 85$, an adversary requires about $\frac{3}{5} \cdot \frac{2^{k_B}}{\mu}$ seconds to factor N. We ran some experiments on an Intel Q9550 quad core CPU (2.83 GHz) that indicate a value of $\mu = 385{,}000$ $^{\text{mults}}/_{\text{second}}$ on that device (which has considerably faster individual cores than our Tesla machine does). Thus, setting k_B as low as 55 yields over 1500 years of (non-parallelizable) wall-clock time to factor N using the Pollard $p-1$ method on this CPU, while requiring less than two minutes to compute trapdoor discrete logarithms with our two M2050 cards.

Maurer and Yacobi [24, §4] point out that, since the cost of factoring increases with 2^{k_B}, while the cost of computing discrete logarithms increases with $\sqrt{2^{k_B}}$, faster cores and more parallelism only help to increase security. In particular, if μ and Ψ increase by a factor f and g, respectively, then we can revise the parameters such that security increases by a factor of fg^2.

[9] Other factoring algorithms, such as ECM [23] or the quadratic sieve algorithm (QS) [35], are highly parallelizable and can factor a general modulus with *sublinear* asymptotic complexity; however, the linear cost of Pollard's $p-1$ factoring algorithm is by far the most efficient method for factoring N, given its special form and our parameter selection. In other words, while both of the aforementioned algorithms have superior asymptotic complexity to Pollard's $p-1$ factoring algorithm (depending on how one asymptotically relates k_B and k_N), the actual position on the $O(2^{k_B})$ cost curve in our case is much smaller than the corresponding position on the cost curves for these asymptotically faster algorithms. For reference, factoring a 1536-bit RSA modulus using more efficient algorithms requires about 2^{85} (parallelizable) work; thus, the cost curves intersect when $\Psi \cdot 2^{k_B} \approx 2^{85}$ and Pollard's $p-1$ algorithm ceases to be most efficient for larger values of $\Psi \cdot 2^{k_B}$ [22].

Zero-knowledge proofs of costliness. Using a straightforward generalization of the zero-knowledge proof that a number is a product of two safe primes, due to Camenisch and Michels [12], one can prove in zero-knowledge that the prime factors q of $\varphi(N)$ each satisfy $2^{k_B} < q < 2^{k_B+1}$. Given some assumptions about available computing power, Equation (2) lets us estimate how long an average trapdoor discrete logarithm computation takes. While the validity of this proof relies on assumptions about the available computational capacity, it *does* give a reliable estimate of the computational — and thus economic — cost of being able to compute discrete logarithms, which is useful in applications such as partial key escrow [3] or our own anonymous blacklisting [18].

7 Conclusion

In this paper, we discussed our experiences with using GPUs and Nvidia's CUDA framework to accelerate the computation of discrete logarithms with respect to a special class of moduli. In particular, we presented our approach to implementing fast, arbitrary-precision modular multiplication on GPUs using C for CUDA and the PTX instruction set architecture, and then described how we were able to leverage this modular multiplication to implement a parallel version of Pollard's rho algorithm. We also examined the implications to existing cryptosystems whose security is based on the presumed intractability of computing discrete logarithms, and pointed out that efficient implementations of Pollard's rho for groups with smooth order enables the construction of cryptographically secure trapdoor discrete logarithm groups. All of our source code is open source and is freely available online from the CrySP group's website at `http://crysp.uwaterloo.ca/software/`.

Acknowledgements. We thank the anonymous reviewers for their thoughtful and constructive comments. Funding for this research was provided in part by NSERC, Mprime NCE (formerly MITACS) and the Ontario Research Fund. The first author is supported by an NSERC Vanier Canada Graduate Scholarship and a Cheriton Graduate Scholarship.

References

[1] Carlos Aguilar Melchor, Benoit Gaborit, Philippe Gaborit, Vincent Jolivet, and Pierre Rousseau. High-speed Private Information Retrieval Computation on GPU. In *Proceedings of SECURWARE 2008*, pages 263–272, Cap Esterel, France, August 2008.

[2] Daniel V. Bailey, Lejla Batina, Daniel J. Bernstein, Peter Birkner, Joppe W. Bos, Hsieh-Chung Chen, Chen-Mou Cheng, Gauthier van Damme, Giacomo de Meulenaer, Luis Julian Dominguez Perez, Junfeng Fan, Tim Güneysu, Frank Gurkaynak, Thorsten Kleinjung, Tanja Lange, Nele Mentens, Ruben Niederhagen, Christof Paar, Francesco Regazzoni, Peter Schwabe, Leif Uhsadel, Anthony Van Herrewege, and Bo-Yin Yang. Breaking ECC2K-130. Cryptology ePrint Archive, Report 2009/541, 2009. Available online: `http://eprint.iacr.org/2009/541.pdf`.

[3] Mihir Bellare and Shafi Goldwasser. Verifiable Partial Key Escrow. In *Proceedings of CCS 1997*, pages 78–91, Zurich, Switzerland, April 1997.

[4] Daniel J. Bernstein, Hsieh-Chung Chen, Chen-Mou Cheng, Tanja Lange, Ruben Niederhagen, Peter Schwabe, and Bo-Yin Yang. ECC2K-130 on NVIDIA GPUs. In *Proceedings of INDOCRYPT 2010*, volume 6498 of *LNCS*, pages 328–346, Hyderabad, India, December 2010.

[5] Daniel J. Bernstein, Hsueh-Chung Chen, Ming-Shing Chen, Chen-Mou Cheng, Chun-Hung Hsiao, Tanja Lange, Zong-Cing Lin, and Bo-Yin Yang. The Billion-Mulmod-Per-Second PC. In *Workshop record of SHARCS'09: Special-purpose Hardware for Attacking Cryptographic Systems*, Lausanne, Switzerland, September 2009.

[6] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. On the Correct Use of the Negation Map in the Pollard rho Method. In *Proceedings of PKC 2011*, volume 6571 of *LNCS*, pages 128–146, March 2011.

[7] Joppe W. Bos, Marcelo E. Kaihara, Thorsten Kleinjung, Arjen K. Lenstra, and Peter L. Montgomery. Solving a 112-bit Prime Elliptic Curve Discrete Logarithm Problem on Game Consoles using Sloppy Reduction. *International Journal of Applied Cryptography*, 2011.

[8] Joppe W. Bos, Marcelo E. Kaihara, and Peter L. Montgomery. Pollard Rho on the PlayStation 3. In *Workshop record of SHARCS'09: Special-purpose Hardware for Attacking Cryptographic Systems*, Lausanne, Switzerland, September 2009.

[9] Fabrice Boudot. Efficient Proofs that a Committed Number Lies in an Interval. In *Proceedings of EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 431–444, Bruges, Belgium, May 2000.

[10] Richard P. Brent. An Improved Monte Carlo Factorization Algorithm. *BIT*, 20:176–184, 1980.

[11] Richard P. Brent. Parallel Algorithms for Integer Factorisation. *Number Theory and Cryptography*, pages 26–37, 1990.

[12] Jan Camenisch and Markus Michels. Proving in Zero-Knowledge that a Number Is the Product of Two Safe Primes. In *Proceedings of EUROCRYPT 1999*, volume 1592 of *LNCS*, pages 107–122, Prague, Czech Republic, May 1999.

[13] Certicom Corporation. Certicom ECC Challenge. Available online: `http://www.certicom.com/images/pdfs/cert_ecc_challenge.pdf`.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[15] Sebastian Fleissner. GPU-Accelerated Montgomery Exponentiation. In *Proceedings of ICCS 2007*, Beijing, China, May 2007.

[16] Robert W. Floyd. Nondeterministic Algorithms. *Journal of the ACM*, 14(4):635–644, October 1967.

[17] Owen Harrison and John Waldron. Public Key Cryptography on Modern GPU Hardware. In *Booklet of accepted posters for Eurocrypt 2009*, Cologne, Germany, April 2009.

[18] Ryan Henry, Kevin Henry, and Ian Goldberg. Making a Nymbler Nymble Using VERBS. In *Proceedings of PETS 2010*, volume 6205 of *LNCS*, pages 111–129, Berlin, Germany, July 2010.

[19] Jens Hermans, Frederik Vercauteren, and Bart Preneel. Speed Records for NTRU. In *Proceedings of CT-RSA 2010*, volume 5985 of *LNCS*, pages 73–88, San Francisco, California, March 2010.

[20] Gareth A. Jones and Josephine M. Jones. *Elementary Number Theory*. Springer-Verlag, 1998.

[21] Çetin Kaya Koç, Tolga Acar, and Burton S. Kaliski Jr. Analyzing and Comparing Montgomery Multiplication Algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[22] Arjen K. Lenstra. Key Lengths. In Hossein Bidgoli, editor, *Handbook of Information Security*, volume II, pages 617–635. Wiley, December 2005.

[23] Hendrik W. Lenstra Jr. Factoring Integers With Elliptic Curves. *Annals of Mathematics*, 126(3):649–673, 1987.

[24] Ueli M. Maurer and Yacov Yacobi. A Non-interactive Public-Key Distribution System. *Designs, Codes and Cryptography*, 9(3):305–316, 1996.

[25] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[26] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[27] Andrew Moss, Dan Page, and Nigel P. Smart. Toward Acceleration of RSA Using 3D Graphics Hardware. In *Proceedings of the IMA International Conference on Cryptography and Coding 2007*, Cirencester, UK, December 2007.

[28] Samuel Neves and Filipe Araujo. On the Performance of GPU Public-Key Cryptography. In *Proceedings of ASAP 2011*, pages 133–140, Santa Monica, California, September 2011.

[29] NVIDIA Corporation. Why Choose Tesla. Available online: `http://www.nvidia.com/object/why-choose-tesla.html`. Retrieved 18-Jan-2012.

[30] NVIDIA Corporation. NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi, February 2010. v1.1. Available online: `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf`.

[31] NVIDIA Corporation. CUDA C Best Practices Guide, May 2011. DG-05603-001_v4.0. Available online: `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf`.

[32] Stephen Pohlig and Martin Hellman. An Improved Algorithm for Computing Logarithms Over GF(p) and its Cryptographic Significance. *IEEE Transactions on Information Theory*, IT-24(1):106–110, January 1978.

[33] John M. Pollard. Theorems of Factorization and Primality Testing. *Proceedings of the Cambridge Philosophical Society*, 76(3):521–528, 1974.

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

[35] Carl Pomerance. Analysis and Comparison of Some Integer Factoring Algorithms. In *Computational Methods in Number Theory, Part I*, pages 89–139, Amsterdam, 1982.

[36] Carl Pomerance and Igor Shparlinski. Smooth Orders and Cryptographic Applications. In *Proceedings of ANTS 2002*, volume 2369 of *LNCS*, pages 338–348, Sydney, Australia, July 2002.

[37] Michael O. Rabin and Jeffrey O. Shallit. Randomized Algorithms in Number Theory. *Communications on Pure and Applied Mathematics*, 39(S1):S239–S256, 1986.

[38] Joseph A. Silverman. *The Arithmetic of Elliptic Curves*. Graduate Texts in Mathematics. Springer-Verlag, 1994.

[39] Edlyn Teske. Speeding Up Pollard's Rho Method for Computing Discrete Logarithms. In *Proceedings of ANTS 1998*, volume 1423 of *LNCS*, pages 541–554, Portland, Oregon, June 1998.

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

Cryptanalysis of MD5 and SHA-1

Marc Stevens
Centrum voor Wiskunde en Informatica (CWI)
marc@marc-stevens.nl
http://marc-stevens.nl/research/

Abstract

In this talk, I will review our most recent cryptanalytic methods on MD5 and SHA-1 and discuss implementation issues. In particular I will focus on a new exact disturbance vector analysis for SHA-1 that in contrast to current literature takes into account the dependence of local collisions. Furthermore, we show how it can be used to implement both an identical-prefix and a chosen-prefix collision attack on SHA-1 which improve on the respective best known attacks

Solving Quadratic Equations with XL on Parallel Architectures

Chen-Mou Cheng[†‡], Tung Chou[†], Ruben Niederhagen[†*], Bo-Yin Yang[†]
†Academia Sinica, Taipei, Taiwan
‡National Taiwan University, Taipei, Taiwan
∗Technical University Eindhoven, Eindhoven, the Netherlands

Abstract

Solving a system of multivariate quadratic equations (MQ) is a hard problem whose complexity estimates are relevant to many cryptographic scenarios. In some cases it is required in the best known attack; sometimes it is a generic attack (such as for the multivariate PKCs), and some of the time it determines a provable level of security (such as for the QUAD stream ciphers).

Under some reasonable-looking assumptions, the best way to solve generic MQ systems is the XL algorithm implemented with a sparse matrix solver such as Wiedemann. Knowing how fast one can implement this attack gives us a good idea of how future cryptosystems related to MQ can be broken, similar to how implementations of General Number Field Sieve that factors smaller RSA numbers gives us more insight into the security of actual RSA-based cryptosystems.

This paper describes such an implementation of XL with Block Wiedemann. We are able to solve in 2.5 days, on a US$6000 computer, a system with 30 variables and 60 equations over $\mathbb{F}_{16}$ (a computation of about 2^{57} $\mathbb{F}_{16}$-multiplications). This is something that we do not expect that F_4/F_5 would accomplish due to its much higher space usage. The software can be easily adapted to other small fields including $\mathbb{F}_2$. More importantly, it scales nicely for small clusters, NUMA machines, and a combination of both. The software is expected to go into SAGE or other open-source projects.

keywords: XL, Gröbner Basis, Block Wiedemann, sparse solver, multivariate quadratic systems

1 Introduction

Some cryptographic systems can be attacked by solving a system of multivariate quadratic equations. For example the symmetric block cipher AES can be attacked by solving a system of 8000 quadratic equations with 1600 variables over $\mathbb{F}_2$ as shown by Courtois and Pieprzyk in [CP02] or by solving a system of 840 sparse quadratic equations and 1408 linear equations over 3968 variables of $\mathbb{F}_{256}$ as shown by Murphy and Robshaw in [MR02] (see also remarks by Murphy and Robshaw in [MR03]). Multivariate cryptographic systems can be attacked naturally by solving their multivariate quadratic system; see for example the analysis of the QUAD stream cipher by Yang, Chen, Bernstein, and Chen in [YCBC07].

We describe a parallel implementation of an algorithm for solving quadratic systems that was first suggested by Lazard in [Laz83]. Later it was reinvented by Courtois, Klimov, Patarin, and Shamir and published in [CKPS00]; they call the algorithm *XL* as an acronym for *extended linearization*: XL *extends* a quadratic system by multiplying with appropriate monomials and *linearizes* it by treating each monomial as an independent variable. Due to this extended linearization, the problem of solving a quadratic system turns into a problem of linear algebra.

XL is a special case of Gröbner basis algorithms (shown by Ars, Faugère, Imai, Kawazoe, and Sugita in [AFI+04]) and can be used as an alternative to other Gröbner basis solvers like Faugère's F_4 and F_5

algorithms (introduced in [Fau99] and [Fau02]). An enhanced version of F_4 is implemented for example by the computer algebra system Magma.

There is an ongoing discussion on whether XL-based algorithms or algorithms of the F_4/F_5-family are more efficient in terms of runtime complexity and memory complexity. One school of thought advocates that for large enough systems XL with a sparse matrix solver is the best generic attack to be accounted for cryptographers (unless certain special structures are present).

To achieve a better understanding of the practical behaviour of XL, we describe herein a parallel implementation of the XL algorithm for shared memory systems, for small computer clusters, and for a combination of both. Measurements of the efficiency of the parallelization have been taken at small size clusters of up to 4 nodes and shared memory systems of up to 48 cores.

This paper is structured as follows: The XL algorithm is introduced in Section 2. Section 3 explains Coppersmith's block Wiedemann algorithm which is used for solving the linearized system. Sections 4 and 5 introduce variations of the Berlekamp–Massey algorithm that are used as building block for Coppersmith's block Wiedemann algorithm: Sections 4 describes Coppersmith's version and section 5 introduces an alternative algorithm invented by Thomé. An implementation of XL using the block Wiedemann algorithm is described in Section 6. Section 7 gives runtime measurements and performance values that are achieved by this implementation for a set of parameters on several parallel systems.

Notations: In this paper a subscript is usually used to denote a row in a matrix, e.g., A_i means the i-th row of matrix A. The entry at the i-th row and j-th column of the matrix A is denoted by $A_{i,j}$. A sequence is denoted as $\{s^{(i)}\}_{i=0}^{\infty}$. The coefficient of the degree-i term in the expansion of a polynomial $f(\lambda)$ is denoted as $f[i]$, e.g., $(\lambda+1)^3[2] = (\lambda^3+3\lambda^2+3\lambda+1)[2] = 3$. The cost (number of field operations) to perform a matrix multiplication AB of matrices $A \in K^{a\times b}$ and $B \in K^{b\times c}$ is denoted as $\mathrm{Mul}(a,b,c)$. The asymptotic time complexity of such a matrix multiplication depends on the size of the matrices, on the field K, and on the algorithm that is used for the computation. Therefore the complexity analyses in this paper use the bound for simple matrix multiplication $O(a \cdot b \cdot c)$ as upper bound for the asymptotic time complexity of matrix multiplications.

2 The XL algorithm

The original description of XL for multivariate quadratic systems can be found in [CKPS00]; a more general definition of XL for systems of higher degree is given in [Cou03]. The following gives a brief introduction of the XL algorithm for quadratic systems; the notation is adapted from [YCC04]:

Consider a finite field $K = \mathbb{F}_q$ and a system $\mathcal{A}$ of m multivariate quadratic equations $\ell_1 = \ell_2 = \cdots = \ell_m = 0$ for $\ell_i \in K[x_1, x_2, \ldots, x_n]$. For $b \in \mathbb{N}^n$ denote by x^b the monomial $x_1^{b_1}x_2^{b_2}\ldots x_n^{b_n}$ and by $|b| = b_1 + b_2 + \cdots + b_n$ the total degree of x^b.

XL first chooses a $D \in \mathbb{N}$ called operational degree and extends the system $\mathcal{A}$ to the system $\mathcal{R}^{(D)} = \{x^b\ell_i = 0 : |b| \leq D-2, \ell_i \in \mathcal{A}\}$ of maximum degree D by multiplying each equation of $\mathcal{A}$ by all monomials of degree less than or equal to $D-2$. Now each monomial $x^d, d \leq D$ is considered a new variable to obtain a linear system $\mathcal{M}$. Note that the system $\mathcal{M}$ is sparse since each equation has the same number of non-zero coefficients as the corresponding equation of the quadratic system $\mathcal{A}$. Finally the linear system $\mathcal{M}$ is solved. If the operational degree D was well chosen, the linear system contains sufficient information about the quadratic equations so that the solution for $x_1, x_2, \ldots x_n$ of the linearized system of $\mathcal{R}^{(D)}$ is also a solution for $\mathcal{A}$; this can easily be checked. Otherwise, the algorithm is repeated with a larger D.

Let $\mathcal{T}^{(D-2)} = \{x^b : |b| \leq D-2\}$ be the set of all monomials with total degree less than or equal to $D-2$. The number $|\mathcal{T}^{(D-2)}|$ of all these monomials is $\binom{n+(D-2)}{n}$ for large fields, and smaller for finite fields GF [YC04, YC05]. Therefore the size of $\mathcal{R}^{(D)}$ grows exponentially with the operational degree D. Consequently, the choice of D should not be larger than the minimum degree that is necessary to find a solution. On the other hand, starting with a small operational degree may result in several repetitions of the XL algorithm and therefore would take more computation than necessary.

In general we get around this dilemma using a heuristic formula given by Yang and Chen in [YC04]. This formula is proved for the large-field case for generic systems by Diem in [Die04], assuming the "maximal rank conjecture" of Ralf Fröberg. In this case, where $q > D$ (see also Moh in [Moh01]), the minimum degree D_0 required for the reliable termination of XL is given by $D_0 := \min\{D : ((1-\lambda)^{m-n-1}(1+\lambda)^m)[D] \leq 0\}$.

3 The block Wiedemann algorithm

The computationally most expensive task in XL is to find a solution for a sparse linear system of equations over a finite field. There are two popular algorithms for that task, the block Lanczos algorithm and the block Wiedemann algorithm. However, the block Lanczos algorithm [Cop93] is not reliable for computations on fields with a characteristic other than 0. Therefore the block Wiedemann algorithm is used for XL. This algorithm was proposed by Coppersmith in 1994 [Cop94] and is a generalization of the original Wiedemann algorithm [Wie86].

The block Wiedemann algorithm has several features that make it powerful for computation in XL. From the original Wiedemann algorithm it inherits the property that the runtime is directly proportional to the weight of the input matrix. Therefore this algorithm is suitable for solving sparse matrices, which is exactly the case for XL. Furthermore big parts of the block Wiedemann algorithm can be parallelized on several types of parallel architectures.

This section describes the implementation of the block Wiedemann algorithm. Although this algorithm is used as a subroutine of XL, the contents in this section are suitable for other applications since they are independent of the shape or data structure of the input matrix.

The block Wiedemann algorithm is a probabilistic algorithm. It solves a linear system $\mathcal{M}$ by computing kernel vectors of a corresponding matrix B in three steps which are called BW1, BW2, and BW3 for the remainder of this paper. The following paragraphs give a review of these three steps on an operational level; for more details please refer to [Cop94].

BW1: Given an input matrix $B \in K^{N\times N}$ and $m, n \in \mathbb{N}$ with $m \geq n$, the first step BW1 computes the first $\frac{N}{m} + \frac{N}{n} + O(1)$ elements of a sequence $\{a^{(i)}\}_{i=0}^{\infty}$ of matrices $a^{(i)} \in K^{n\times m}$ using random matrices $x \in K^{m\times N}$ and $z \in K^{N\times n}$ such that

$$a^{(i)} = (xB^i y)^T, \quad \text{for } y = Bz.$$

The parameters m and n are chosen such that operations on vectors K^m and K^n can be computed efficiently on the target computing architecture. In the following we treat the quotient $\lceil m/n \rceil$ as a constant for convenience. In practice each $a^{(i)}$ can be efficiently computed using two matrix multiplications with the help of a sequence $\{t^{(i)}\}_{i=0}^{\infty}$ of matrices $t^{(i)} \in K^{N\times n}$ defined as

$$t^{(i)} = \begin{cases} y = Bz & \text{for } i = 0 \\ Bt^{(i-1)} & \text{for } i > 0. \end{cases}$$

Thus, $a^{(i)}$ can be computed as

$$a^{(i)} = (xt^{(i)})^T.$$

Therefore, the asymptotic time complexity of BW1 can be written as

$$O\left(\left(\frac{N}{m} + \frac{N}{n}\right)(Nw_B n + mNn)\right) = O\left((w_B + m)N^2\right),$$

where w_B is the average number of nonzero entries per row of B.

BW2: Coppersmith uses an algorithm for this step that is a generalization of the Berlekamp–Massey algorithm given in [Ber66, Mas69]. Literature calls Coppersmith's modified version of the Berlekamp–Massey algorithm "block Berlekamp–Massey" algorithm in analogy to the name "block Wiedemann" or "matrix Berlekamp–Massey" algorithm.

The block Berlekamp–Massey algorithm is an iterative algorithm. It takes the sequence $\{a^{(i)}\}_{i=0}^{\infty}$ from BW1 as input and defines the polynomial $a(\lambda)$ of degree $\frac{N}{m} + \frac{N}{n} + O(1)$ with coefficients in $K^{n\times m}$ as

$$a(\lambda) = \sum_i a^{(i)}\lambda^i.$$

The j-th iteration step receives two inputs from the previous iteration: One input is an $(m+n)$-tuple of polynomials $(f_1^{(j)}(\lambda), \ldots, f_{m+n}^{(j)}(\lambda))$ with coefficients in $K^{1\times n}$; these polynomials are jointly written as $f^{(j)}(\lambda)$ with coefficients in $K^{(m+n)\times n}$ such that $(f^{(j)}[k])_i = f_i^{(j)}[k]$. The other input is an $(m+n)$-tuple $d^{(j)}$of *nominal degrees* $(d_1^{(j)}, \ldots, d_{m+n}^{(j)})$; each nominal degree $d_k^{(j)}$ is an upper bound of $\deg(f_k^{(j)})$.

An initialization step generates $f^{(j_0)}$ for $j_0 = \lceil m/n \rceil$ as follows: Set the polynomials $f_{m+i}^{(j_0)}, 1 \le i \le n$, to the polynomial of degree j_0 where coefficient $f_{m+i}^{(j_0)}[j_0] = e_i$ is the i-th unit vector and with all other coefficients $f_{m+i}^{(j_0)}[k] = 0,\ k \ne j_0$. Try choosing the polynomials $f_1^{(j_0)}, \ldots, f_m^{(j_0)}$ randomly with degree $j_0 - 1$ until $H^{(j_0)} = (f^{(j_0)}a)[j_0]$ has rank m. Finally set $d_i^{(j_0)} = j_0,$ for $0 \le i \le (m+n)$.

After $f^{(j_0)}$ and $d^{(j_0)}$ have been initialized, iterations are carried out until $f^{(\deg(a))}$ is computed as follows: In the j-th iteration, a Gaussian elimination according to Algorithm 1 is performed on the matrix

$$H^{(j)} = (f^{(j)}a)[j] \in K^{(m+n)\times m}.$$

Note that the algorithm first sorts the rows of the input matrix by their corresponding nominal degree in decreasing order. This ensures that during the Gaussian elimination no rows of higher nominal degree are subtracted from a row with lower nominal degree. The Gaussian elimination finds a nonsingular matrix $P^{(j)} \in K^{(m+n)\times(m+n)}$ such that the first n rows of $P^{(j)}H^{(j)}$ are all zeros and a permutation matrix $E^{(j)} \in K^{(m+n)\times(m+n)}$ corresponding to a permutation $\phi^{(j)}$. Using $P^{(j)}$, the polynomial $f^{(j+1)}$ of the next iteration step is computed as

$$f^{(j+1)} = QP^{(j)}f^{(j)}, \quad \text{for } Q = \begin{pmatrix} I_n & 0 \\ 0 & \lambda \cdot I_m \end{pmatrix}.$$

The nominal degrees $d_i^{(j+1)}$ are computed corresponding to the multiplication by Q and the permutation $\phi^{(j)}$ as

$$d_i^{(j+1)} = \begin{cases} d_{\phi_i^{(j)}}^{(j)} & \text{for } 1 \le i \le n, \\ d_{\phi_i^{(j)}}^{(j)} + 1 & \text{for } n < i \le (n+m). \end{cases}$$

The major tasks in each iteration are:

1. The computation of $H^{(j)}$, which takes
$$\deg(f^{(j)})\mathrm{Mul}(m+n,n,m) = O(\deg(f^{(j)}) \cdot n^3);$$
note that only the coefficient of λ^j of $f^{(j)}(\lambda)a(\lambda)$ needs to be computed.
2. The Gaussian elimination, which takes $O(n^3)$.
3. The multiplication $P^{(j)}f^{(j)}$, which takes
$$\deg(f^{(j)})\mathrm{Mul}(m+n,m+n,n) = O(\deg(f^{(j)}) \cdot n^3).$$

In fact $\deg(f^{(j)})$ is always bounded by j since $\max(d^{(j)})$ is at most increased by one in each round. Therefore, the total asymptotic time complexity of Berlekamp–Massey is
$$\sum_{j=j_0}^{N/m+N/n+O(1)} O(j \cdot n^3) = O\left(N^2 \cdot n\right).$$

For the output of BW2, the last m rows of $f^{(\deg(a))}$ are discarded; the output is an n-tuple of polynomials $(f_1(\lambda), \ldots f_n(\lambda))$ with coefficients in $K^{1\times n}$ and an n-tuple $d = (d_1, \ldots, d_n)$ of nominal degrees such that
$$f_k = f_k^{(\deg(a))}$$
and
$$d_k = d_k^{(\deg(a))},$$
for $1 \le k \le n$, where $\max(d) \approx N/n$.

BW3: This step receives an n-tuple of polynomials $(f_1(\lambda), \ldots f_n(\lambda))$ with coefficients in $K^{1\times n}$ and an n-tuple $d = (d_1, \ldots, d_n)$ as input from BW2. For each $f_i(\lambda),\ 1 \le i \le n$, compute $w_i \in K^N$ as
$$\begin{aligned} w_i &= z(f_i[\deg(f_i)])^T + B^1 z(f_i[\deg(f_i)-1])^T + \ldots + B^{\deg(f_i)} z(f_i[0])^T \\ &= \sum_{j=0}^{\deg(f_i)} B^j z(f_i[\deg(f_i)-j])^T. \end{aligned}$$

Note that this corresponds to an evaluation of the reverse of f_i. To obtain a kernel vector of B, multiply w_i by B until $B^{(k_i+1)}w_i = 0,\ 0 \le k_i \le (d_i - \deg(f_i))$. Thus, $B^{k_i}w_i$ is a kernel vector of B.

The block Wiedemann algorithm is a probabilistic algorithm. Therefore, it is possible that this computation does not find a kernel vector for some $f_i(\lambda)$. For a probabilistic analysis of Coppersmith's block Wiedemann algorithm see [Kal95, Vil97a, Vil97b].

In practice, the kernel vectors can be computed efficiently by operating on all polynomials $f_i(\lambda)$ together. As in step BW2, all $f_i(\lambda)$ are written jointly as $f(\lambda)$ with coefficients in $K^{n\times n}$ such that $(f[k])_i = f_i[k]$. By applying Horner's scheme, the kernel vectors can be computed iteratively with the help of a sequence $\{W^{(j)}\}_{j=0}^{\max(d)},\ W^{(j)} \in K^{N\times n}$ using up to two matrix multiplications for each iteration as follows:
$$W^{(j)} = \begin{cases} z \cdot (f[0])^T & \text{for } j = 0, \\ z \cdot (f[j])^T + B \cdot W^{(j-1)} & \text{for } 0 < j \le \deg(f), \\ B \cdot W^{(j-1)} & \text{for } \deg(f) < j \le \max(d). \end{cases}$$

The kernel vectors of B are found during the iterative computation of $W^{(\max(d))}$ by checking whether an individual column $i \in \{1, \ldots, n\}$ is nonzero in iteration k but becomes zero in iteration $k+1$. Therefore, column i of matrix $W^{(k)}$ is a kernel vector of B.

Each iteration step has a asymptotically time complexity of

$$O\left(Nn^2 + Nw_B n\right) = O\left(N \cdot (n + w_B) \cdot n\right).$$

Therefore, $W^{(\max(d))}$ for $\max(d) \approx N/n$ can be computed with the asymptotic time complexity

$$O\left(N^2 \cdot (w_B + n)\right).$$

The output of BW3 and of the whole block Wiedemann algorithm are up to n kernel vectors of B.

4 The block Berlekamp–Massey algorithm

This section first introduces a tweak that allows to speed up computations of Coppersmith's variant of the Berlekamp–Massey algorithm. Later the parallelization of the algorithm is described.

4.1 Reducing the cost of the block Berlekamp–Massey algorithm

The j-th iteration of Coppersmith's Berlekamp–Massey algorithm requires a matrix $P^{(j)} \in K^{(m+n)\times(m+n)}$ such that the first n rows of $P^{(j)}H^{(j)}$ are all zeros. The main idea of this tweak is to make $P^{(j)}$ have the form

$$P^{(j)} = \begin{pmatrix} I_n & * \\ 0 & I_m \end{pmatrix} E^{(j)},$$

where $E^{(j)}$ is a permutation matrix corresponding to a permutation $\phi^{(j)}$ (the superscript will be omitted in this section). Therefore, the multiplication $P^{(j)}f^{(j)}$ takes only $\deg(f^{(j)}) \cdot \mathrm{Mul}(n, m, n)$ field operations (for the upper right submatrix in $P^{(j)}$).

The special form of $P^{(j)}$ also makes the computation of $H^{(j)}$ more efficient: The bottom m rows of each coefficient are simply permuted due to the multiplication by $P^{(j)}$, thus

$$(P^{(j)}f^{(j)}[k])_i = (f^{(j)}[k])_{\phi(i)},$$

for $n < i \leq m + n,\ 0 < k \leq \deg(f^{(j)})$. Since multiplication by Q corresponds to a multiplication of the bottom m rows by λ, it does not modify the upper n rows of the coefficients. Therefore, the bottom m rows of the coefficients of $f^{(j+1)}$ can be obtained from $f^{(j)}$ as

$$(f^{(j+1)}[k])_i = (QP^{(j)}f^{(j)}[k-1])_i = (f^{(j)}[k-1])_{\phi(i)},$$

for $n < i \leq m + n,\ 0 < k \leq deg(f^{(j)})$. Since the bottom right corner of $P^{(j)}$ is the identity matrix of size m, this also holds for

$$((f^{(j+1)}a)[j+1])_i = ((QP^{(j)}f^{(j)}a)[j+1])_i = ((f^{(j)}a)[j])_{\phi(i)}.$$

Thus, $H_i^{(j+1)}$ for $n < i \leq m + n$ can be computed as

$$H_i^{(j+1)} = ((f^{(j+1)}a)[j+1])_i = ((QP^{(j)}f^{(j)}a)[j+1])_i = ((f^{(j)}a)[j])_{\phi(i)} = H_{\phi(i)}^{(j)}.$$

This means the last m rows of $H^{(j+1)}$ can actually be copied from $H^{(j)}$; only the first n rows of $H^{(j+1)}$ need to be computed. Therefore the cost of computing any $H^{(j>j_0)}$ is reduced to $\deg(f^{(j)}) \cdot \mathrm{Mul}(n, n, m)$.

The matrix $P^{(j)}$ can be assembled as follows: The matrix $P^{(j)}$ is computed using Algorithm 1. In this algorithm a sequence of row operations is applied to $M := H^{(j)}$. The matrix $H^{(j)}$ has rank m for all $j \geq j_0$. Therefore in the end the first n rows of M are all zeros. The composition of all the operations is P; some of these operations are permutations of rows. The composition of these permutations is E:

$$P^{(j)}(E^{(j)})^{-1} = \begin{pmatrix} I_n & * \\ 0 & F^{(j)} \end{pmatrix} \iff P^{(j)} = \begin{pmatrix} I_n & * \\ 0 & F^{(j)} \end{pmatrix} E^{(j)}.$$

The algorithm by Coppersmith requires that the first n rows of $P^{(j)}H^{(j)}$ are all zero (see [Cop94, 7]); there is no condition for the bottom m rows. However, the first n rows of $P^{(j)}H^{(j)}$ are all zero independently of the value of $F^{(j)}$. Thus, $F^{(j)}$ can be replaced by I_m without harming this requirement.

4.2 Parallelization of the block Berlekamp–Massey algorithm

The parallel implementation of the block Berlekamp–Massey algorithm on c nodes works as follows: In each iteration step, the coefficients of $f^{(j)}(\lambda)$ are equally distributed over the computing nodes; for $0 \leq i < c$, let $S_i^{(j)}$ be the set containing all indices of coefficients stored by node i during the j-th iteration. Each node stores a copy of all coefficients of $a(\lambda)$.

Due to the distribution of the coefficients, the computation of

$$H^{(j)} = (f^{(j)}a)[j] = \sum_{l=0}^{j} f^{(j)}[l]a[j-l]$$

requires communication: Each node i first locally computes a part of the sum using only its own coefficients $S_i^{(j)}$ of $f^{(j)}$. The matrix $H^{(j)}$ is the sum of all these intermediate results. Therefore, all nodes broadcast their intermediate results to the other nodes. Each node computes $H^{(j)}$ locally; Gaussian elimination is performed on every node locally and is not parallelized over the nodes. Since only small matrices are handled, this sequential overhead is negligibly small.

Also the computation of $f^{(j+1)}$ requires communication. Recall that

$$f^{(j+1)} = QP^{(j)}f^{(j)}, \quad \text{for } Q = \begin{pmatrix} I_n & 0 \\ 0 & \lambda \cdot I_m \end{pmatrix}.$$

Therefore each coefficient k is computed row-wise as

$$(f^{(j+1)}[k])_l = \begin{cases} ((P^{(j)}f^{(j)})[k])_l, & \text{for } 0 < l \leq n, \\ ((P^{(j)}f^{(j)})[k-1])_l, & \text{for } n < l \leq m+n. \end{cases}$$

Computation of $f^{(j+1)}[k]$ requires access to both coefficients k and $(k+1)$ of $f^{(j)}$. Therefore, communication cost is reduced by distributing the coefficients equally over the nodes such that each node stores a continuous range of coefficients of $f^{(j)}$ and such that the indices in $S_{i+1}^{(j)}$ always are larger than those in $S_i^{(j)}$.

Due to the multiplication by Q, the degree of $f^{(j)}$ is increased at most by one in each iteration. Therefore at most one more coefficient must be stored. The new coefficient obviously is the coefficient with highest

degree and therefore must be stored on node $(c-1)$. To maintain load balancing, one node $i^{(j)}$ is chosen in a round-robin fashion to receive one additional coefficient; coefficients are exchanged between neighbouring nodes to maintain an ordered distribution of the coefficients.

Observe, that only node $(c-1)$ can check whether the degree has increased, i.e. whether $\deg(f^{(j+1)}) = \deg(f^{(j)}) + 1$, and whether coefficients need to be redistributed; this information needs to be communicated to the other nodes. To avoid this communication, the maximum nominal degree $\max(d^{(j)})$ is used to approximating $\deg(f^{(j)})$. Note that in each iteration all nodes can update a local list of the nominal degrees. Therefore, all nodes decide locally without communication whether coefficients need to be reassigned: If $\max(d^{(j+1)}) = \max(d^{(j)}) + 1$, node $i^{(j)}$ is computed as

$$i^{(j)} = \max(d^{(j+1)}) \bmod c.$$

Node $i^{(j)}$ is chosen to store one additional coefficient, the coefficients of nodes i, for $i \geq i^{(j)}$, are redistributed accordingly.

Table 1 illustrates the distribution strategy for 4 nodes. For example in iteration 3, node 1 has been chosen to store one more coefficient. Therefore it receives one coefficient from node 2. Another coefficient is moved from node 3 to node 2. The new coefficient is assigned to node 3.

This distribution scheme does not avoid all communication for the computation of $f^{(j+1)}$: First all nodes compute $P^{(j)}f^{(j)}$ locally. After that, the coefficients are multiplied by Q. For almost all coefficients of $f^{(j)}$, both coefficients k and $(k-1)$ of $P^{(j)}f^{(j)}$ are stored on the same node, i.e. $k \in S^{(j)}_{(i)}$ and $(k-1) \in S^{(j)}_{(i)}$. Thus, $f^{(j+1)}[k]$ can be computed locally without communication. In the example in Figure 2, this is the case for $k \in \{0, 1, 2, 4, 5, 7, 9, 10\}$. Note that the bottom m rows of $f^{(j+1)}[0]$ and the top n rows of $f^{(j+1)}[\max(d^{(j+1)})]$ are 0.

Communication is necessary if coefficients k and $(k-1)$ of $P^{(j)}f^{(j)}$ are not on the same node. There are two cases:

- In case $k - 1 = \max(S^{(j+1)}_{i-1}) = \max(S^{(j)}_{i-1}),\ i \neq 1$, the bottom m rows of $(P^{(j)}f^{(j)})[k-1]$ are sent from node $i-1$ to node i. This is the case for $k \in \{6, 3\}$ in Figure 2. This case occurs if in iteration $j+1$ no coefficient is reassigned to node $i-1$ due to load balancing.

- In case $k = \min(S^{(j)}_{i}) = \max(S^{(j+1)}_{i-1}),\ i \neq 1$, the top n rows of $(P^{(j)}f^{(j)})[k]$ are sent from node i to node $i-1$. The example in Figure 2 has only one such case, namely for coefficient $k = 8$. This happens, if coefficient k got reassigned from node i to node $i-1$ in iteration $j+1$.

If $\max(d^{(j+1)}) = \max(d^{(j)})$, i.e. the maximum nominal degree is not increased during iteration step j, only the first case occurs since no coefficient is added and therefore reassignment of coefficients is not necessary.

The implementation of this parallelization scheme uses the Message Passing Interface (MPI) for computer clusters and OpenMP for multi-core architectures. For OpenMP, each core is treated as one node in the parallelization scheme. Note that the communication for the parallelization with OpenMP is not programmed explicitly since all cores have access to all coefficients; however, the workload distribution is performed as described above. For the cluster implementation, each cluster node is used as one node in the parallelization scheme. Broadcast communication for the computation of $H^{(j)}$ is implemented using a call to the `MPI_Allreduce` function. One-to-one communication during the multiplication by Q is performed with the non-blocking primitives `MPI_Isend` and `MPI_Irecv` to avoid deadlocks during communication.

Both OpenMP and MPI can be used together for clusters of multi-core architectures. For NUMA systems the best performance is achieved when one MPI process is used for each NUMA node since this prevents expensive remote-memory accesses during computation.

The communication overhead of this parallelization scheme is very small. In each iteration, each node only needs to receive and/or send data of total size $O(n^2)$. Expensive broadcast communication is only required rarely compared to the time spent for computation. Therefore this parallelization of Coppersmith's Berlekamp–Massey algorithm scales well on a large number of nodes. Furthermore, since $f^{(j)}$ is distributed over the nodes, the memory requirement is distributed over the nodes as well.

5 Thomé's subquadratic version of the block Berlekamp–Massey algorithm

In 2002 Thomé presented an improved version of Coppersmith's variation of the Berlekamp–Massey algorithm [Tho02]. Thomé's version is asymptotically faster: It reduces the complexity from $O(N^2)$ to $O(N \log^2(N))$ (assuming that m and n are constants). The subquadratic complexity is achieved by converting the block Berlekamp–Massey algorithm into a recursive divide-and-conquer process. Thomé's version builds the output polynomial $f(\lambda)$ of BW2 using a binary product tree; therefore, the main operations in the algorithm are multiplications of matrix polynomials. The implementation of Coppersmith's version of the algorithm is used to handle bottom levels of the recursion in Thomé's algorithm, as suggested in [Tho02, Section 4.1].

The main computations in Thomé's version of the Berlekamp–Massey algorithm are multiplications of matrix polynomials. The first part of this section will take a brief look how to implement these efficiently. The second part gives an overview of the approach for the parallelization of Thomé's Berlekamp–Massey algorithm.

5.1 Matrix polynomial multiplications

In order to support multiplication of matrix polynomials with various operand sizes in Thomé's Berlekamp–Massey algorithm, several implementations of multiplication algorithms are used including Karatsuba, Toom–Cook, and FFT-based multiplications. FFT-based multiplications are the most important ones because they are used to deal with computationally expensive multiplications of operands with large degrees.

Different kinds of FFT-based multiplications are used for different fields: The field $\mathbb{F}_2$ uses the radix-3 FFT multiplication presented in [Sch77]. For $\mathbb{F}_{16}$ the operands are transformed into polynomials over $\mathbb{F}_{16^9}$ by packing groups of 5 coefficients together. Then a mixed-radix FFT is applied using a primitive r-th root of unity in $\mathbb{F}_{16^9}$. In order to accelerate FFTs, it is ensured that r is a number without large (≥ 50) prime factors.

$\mathbb{F}_{16^9}$ is chosen because it has several advantages. First, by exploiting the Toom-Cook multiplication, a multiplication in $\mathbb{F}_{16^9}$ takes only $9^{\log_3 5} = 25$ multiplications in $\mathbb{F}_{16}$. Moreover, by setting $\mathbb{F}_{16} = \mathbb{F}_2[x]/(x^4 + x + 1)$ and $\mathbb{F}_{16^9} = \mathbb{F}_{16}[y]/(y^9 + x)$, reductions after multiplications can be performed efficiently because of the simple form of $y^9 + x$. Finally, $16^9 - 1$ has many small prime factors and thus there are plenty of choices of r to cover various sizes of operands.

5.2 Parallelization of Thomé's Berlekamp–Massey algorithm

Thomé's Berlekamp–Massey algorithm uses matrix polynomial multiplications and Coppersmith's Berlekamp–Massey algorithm as building blocks. The parallelization of Coppersmith's version has already been ex-

plained. Here the parallelization of the matrix polynomial multiplications is described on the example of the FFT-based multiplication.

The FFT-based multiplication is mainly composed of 3 stages: forward FFTs, point-wise multiplications, and the reverse FFT. Let f, g be the inputs of forward FFTs and f', g' be the corresponding outputs; the point-wise multiplications take f', g' as operands and give h' as output; finally, the reverse FFT takes h' as input and generates h.

For this implementation, the parallelization scheme for Thomé's Berlekamp–Massey algorithm is quite different from that for Coppersmith's: Each node deals with a certain range of rows. In the forward and reverse FFTs the rows of f, g, and h' are independent. Therefore, each FFT can be carried out in a distributed manner without communication. The problem is that the point-wise multiplications require partial f' but full g'. To solve this each node collects the missing rows of g' from the other nodes. This is done by using the function `MPI_Allgather`. Karatsuba and Toom-Cook multiplication are parallelized in a similar way.

One drawback of this scheme is that the number of nodes is limited by the number of rows of the operands. However, when B is very big, the runtime of BW2 should be very small compared to BW1 and BW3 since it is subquadratic in N. In this case using a different, smaller cluster or a powerful multi-core machine for BW2 might give a sufficient performance as suggested in [KAF+10]. Another drawback is, that the divide-and-conquer approach and the recursive algorithms for polynomial multiplication require much more memory than Coppersmith's version of the Berlekamp–Massey algorithm. Thus Coppersmith's version might be a better choice on memory-restricted architectures or for very large systems.

6 Implementation of XL

This section gives an overview of the implementation of XL. Section 6.1 describes some tweaks that are used to reduce the computational cost of the steps BW1 and BW2. This is followed by a description of the building block for these two steps. The building blocks are explained bottom up: Section 6.2 describes the field arithmetic on vectors of $\mathbb{F}_q$; although the implementation offers several fields ($\mathbb{F}_2$, $\mathbb{F}_{16}$, and $\mathbb{F}_{31}$), $\mathbb{F}_{16}$ is chosen as a representative for the discussion in this section. The modularity of the source code allows to easily extend the implementation to arbitrary fields. Section 6.3 describes an efficient approach for storing the Macaulay matrix that takes its special structure into account. This approach reduces the memory demand significantly compared to standard data formats for sparse matrices. Section 6.4 details how the Macaulay matrix multiplication in the stages BW1 and BW3 is performed efficiently, Section 6.5 explains how the multiplication is performed in parallel on a cluster using MPI and on a multi-core system using OpenMP. Both techniques for parallelization can be combined on clusters of multi-core systems.

6.1 Reducing the computational cost of BW1 and BW3

To accelerate BW1, Coppersmith suggests in [Cop94] to use $x = (I_m|0)$ instead of making x a random matrix. However, for the implementation described in this thesis, using $x = (I_m|0)$ turned out to drastically reduce the probability of finding kernel vectors. Instead, a random sparse matrix is used for x with row weight w_x. This reduces the complexity of BW1 from $O(N^2(w_B + m))$ to $O(N^2 w_B + N m w_x)$.

A similar tweak can be used in BW3: Recall that the computations in BW3 can be performed iteratively such that each iteration requires two multiplications $z \cdot (f[k])^T$ and $B \cdot W^{(k-1)}$. However, z is also a randomly generated matrix, so it is deliberately made sparse to have row weight $w_z < n$. This tweak reduces the complexity of BW3 from $O\left(N^2 \cdot (w_B + n)\right)$ to $O\left(N^2 \cdot (w_B + w_z)\right)$.

In this implementation $w_x = w_z = 32$ is used in all cases.

Notes. The tweaks for BW1 and BW3, though useful in practice, actually reduce the entropy of x and z. Therefore, theoretical analyses of [Kal95, Vil97a, Vil97b] do no longer apply.

6.2 SIMD vector operations in $\mathbb{F}_{16}$

In this implementation, field elements of $\mathbb{F}_{16}$ are represented as polynomials over $\mathbb{F}_2$ with arithmetic modulo the irreducible polynomial $x^4 + x + 1$. Therefore one field element is stored using 4 bits $e_0, \ldots, e_3 \in \{0, 1\}$ where each field element $b \in \mathbb{F}_{16}$ is represented as $b = \sum_{i=0}^{3} e_i x^i$. To save memory and fully exploit memory throughput, two field elements are packed into one byte. Therefore the 128-bit SSE vector registers are able to compute on 32 field elements in parallel. To fully exploit SSE registers, vector sizes of a multiple of 32 elements are chosen whenever possible. In the following only vectors of length 32 are considered; operations on longer vectors can be accomplished piecewise on 32 elements at a time.

Additions of two $\mathbb{F}_{16}$ vectors of 32 elements can be easily accomplished by using a single `XOR` instruction of the SSE instruction set.

Scalar multiplications are more expensive. Depending on the microarchitecture, two different implementations are used: processors which offer the SSSE3 extension can profit from the advanced `PSHUFB` instruction; on all other SSE architectures a slightly slower version is used which is based on bitshift operations and logical operations.

General (non-`PSHUFB`) scalar multiplication: Scalar multiplication by x, x^2 and x^3 can be implemented using a small number of bit-operations, e.g. multiplication by x can be performed as:

$$\begin{aligned}(a_3x^3 + a_2x^2 + a_1x + a_0) \cdot x &= a_3(x+1) + a_2x^3 + a_1x^2 + a_0x \\ &= a_2x^3 + a_1x^2 + (a_0 + a_3)x + a_3\end{aligned}$$

Seen from the bit-representation, multiplication by x results in shifting bits 0,1, and 2 by one position to the left and adding (`XOR`) bit 3 to positions 0 and 1. This sequence of operations can be executed on 32 values in an SSE vector register in parallel using 7 bit-operations as shown in figure 3. Similar computations give the multiplication by x^2 and x^3 respectively.

Therefore multiplying a vector $a \in \mathbb{F}_{16}^{32}$ by an arbitrary scalar value $b \in \mathbb{F}_{16}$ can be decomposed to adding up the results of $a \cdot x^i, i \in [0, 3]$ for all bits i of b that are set to 1. The number of bit-operations varies with the actual value of b.

Scalar multiplication using `PSHUFB`: The `PSHUFB` (Packed Shuffle Bytes) instruction was introduced by Intel with the SSSE3 instruction set extension in 2006. The instruction takes two byte vectors $A = (a_0, a_1, \ldots, a_{15})$ and $B = (b_0, b_1, \ldots, b_{15})$ as input and returns $C = (a_{b_0}, a_{b_1}, \ldots, a_{b_{15}})$. In case the top bit of b_i is set, c_i is set to zero. With this instructions the scalar multiplication can be implemented using a lookup table as follows: For $\mathbb{F}_{16}$ the lookup table L contains 16 entries of 128-bit vectors $L_i = (0 \cdot i, 1 \cdot i, x \cdot i, (x+1) \cdot i, \ldots), i \in \mathbb{F}_{16}$. Given a vector register A that contains 16 $\mathbb{F}_{16}$ elements, one in each byte slot, the scalar multiplication $A \cdot b, b \in \mathbb{F}_{16}$ is computed as $A \cdot b = \texttt{PSHUFB}(L_b, A)$.

Since in the implementation each input vector register contains 32 packed elements, two `PSHUFB` instructions have to be used and the inputs need to be unpacked using shift and mask operations accordingly as shown in figure 4. Using the `PSHUFB` instruction, the scalar multiplication needs 7 operations for any input value b with the extra cost of accessing the lookup table.

6.3 Exploiting the structure of the Macaulay matrix

Recall that in XL a system $\mathcal{A}$ of m quadratic equations in n variables over a field $\mathbb{F}_q$ is linearized by multiplying the equations by each of the $T = |\mathcal{T}^{(D-2)}|$ monomials with degree smaller than or equal to $D-2$. The resulting system $\mathcal{R}^{(D)}$ is treated as a linear system using the monomials as independent variables. This linear system is represented by a sparse matrix that consists of T *row blocks* of m rows where each row in a row block is associated with one row in the underlying system $\mathcal{A}$. The entries in these blocks have the same value as the entries in $\mathcal{A}$ and the column positions of the entries are all the same for any line in one row block.

The resulting matrix has the structure of a Macaulay matrix. Since the matrix does not have a square structure as demanded by the Wiedemann algorithm, rows are dropped randomly from the matrix until the resulting matrix has a square shape. Let each equation in $\mathcal{A}$ have w coefficients. Therefore each row in the Macaulay matrix has a weight of at most w.

The Macaulay matrix can be stored in a general sparse-matrix format in memory. Usually for each row in a sparse matrix the non-zero entries are stored alongside with their column position. In a field $\mathbb{F}_q$ a Macaulay matrix with a row-weight of at most w has about $w\frac{q-1}{q}$ non-zero entries per row. For a Macaulay matrix of N rows such a format would need at least $N \cdot w\frac{q-1}{q} \cdot (b_{value} + b_{index})$ bits, b_{value} and b_{index} denoting the number of bits necessary to store the actual value and the column index respectively.

Nevertheless, in a Macaulay matrix all entries are picked from the same underlying quadratic system. Furthermore, the column indices in each row repeat for the up to m consecutive rows spanning over one row block.

Therefore, memory can be saved by storing the values only once as a dense matrix according to the underlying quadratic system. This needs $m \cdot w \cdot b_{value}$ bits of memory. Furthermore, for each row block the column positions of the entries need to be stored. This takes $T \cdot w \cdot b_{index}$ bits. Furthermore, it must be stored to which row of the quadratic system each row in the square Macaulay matrix is referring to—since a bunch of rows has been dropped to make the get a square matrix. Therefore an index to each row of the quadratic system is stored for each row of the Macaulay matrix. This takes $N \cdot b_{sys-index}$ bits of memory.

All in all, the memory demand of the sparse Macaulay matrix can be compressed to $m \cdot w \cdot b_{value} + T \cdot w \cdot b_{index} + N \cdot b_{sys-index}$ bits which reduces the memory demand compared to a sparse matrix storage format significantly. The disadvantage of this storage format is that entries of the underlying quadratic system $\mathcal{A}$ that are known to be zero cannot be skipped as it would be possible when using a standard sparse-matrix format. This may give some computational overhead during the matrix operations. However, this allows to assume that the Macaulay matrix has the fixed row-weight w for each row regardless of the actual values of the coefficients in $\mathcal{A}$ for the reminder of this paper.

Figure 5 shows a graph of the memory demand for several arbitrary system sizes over $\mathbb{F}_{16}$. Given a certain memory size, e.g. 16 GB, systems of about two more equations can be computed in RAM by using the compact storage format.

Note that the column positions can also be recomputed dynamically for each row block instead of storing them explicitly. However, recomputation increases the computational cost significantly while only a relatively small memory amount is necessary to store precomputed column positions. Since this algorithm is rather bound by computation than by memory, it is more efficient to store the values instead of recomputing them on demand.

6.4 Macaulay matrix multiplication in XL

Recall that in XL stage BW1 of the block Wiedemann algorithm is an iterative computation of

$$a[i] = (x^T \cdot (B \cdot B^i z))^T, \quad 0 \leq i \leq \frac{N}{m} + \frac{N}{n} + O(1)$$

and stage BW3 iteratively computes

$$W^{(j)} = z \cdot (f[j])^T + B \cdot W^{(j-1)}$$

where B is a Macaulay matrix, x and z are sparse matrices, and $a[i], f[k]$ are dense matrices.

Figures 6 and 7 show pseudo-code for the iteration loops. The most expensive part in the computation of stages BW1 and BW3 of XL is a repetitive multiplication of the shape

$$t_{new} = B \cdot t_{old}$$

where $t_{new}, t_{old} \in K^{N \times n}$ are dense matrices and $B \in K^{N \times N}$ is a sparse Macaulay matrix of row weight w.

Due to the row-block structure of the Macaulay matrix, there is a guaranteed number of entries per row (i.e. the row weight w) but a varying number of entires per column, ranging from just a few to more than $2w$. Therefore the multiplication is computed in row-order in a big loop over all row indices.

For $\mathbb{F}_{16}$ the field size is significantly smaller than the row weight. Therefore, the number of actual multiplications for a row r can be reduced by summing up all row-vectors of t_{old} which are to be multiplied by the same field element and perform the multiplication on all of them together. A temporary buffer $b_i, i \in \mathbb{F}_{16}$ of vectors of length n is used to collect the sum of row vectors that are multiplied by i. For all entries $B_{r,c}$ row c of t_{old} is added to $b_{B_{r,c}}$. Finally b is reduced by computing $\sum i \cdot b_i, i \neq 0, i \in \mathbb{F}_{16}$, which gives the result for row r of t_{new}.

With the strategy explained so far, computing the result for one row of B takes $w + 14$ additions and 14 scalar multiplications (there is no need for the multiplication of 0 and 1, see [Sch11, Statement 8], and for the addition of 0, see [Pet11, Statement 10]). This can be further reduced by decomposing each scalar factor into the components of the polynomial that represents the field element. Summing up the entries in b_i according to the non-zero coefficients of i's polynomial results in 4 buckets which need to be multiplied by 1, x, x^2, and x^3 (multiplying by 1 can be omitted once more). This reduces 14 scalar multiplications from before to only 3 multiplications on the cost of 22 more additions. All in all the computation on one row of B (row weight w) on $\mathbb{F}_{p^n}$ costs $w + 2(p^n - n - 1) + (n - 1)$ additions and $n - 1$ scalar multiplications (by $x, x^2, \ldots, x^{n-1}$). For $\mathbb{F}_{16}$ this results in $w + 25$ additions and 3 multiplications per row.

In general multiplications are more expensive on architectures which do not support the `PSHUFB`-instruction than on those which do. Observe that in this case the non-`PSHUFB` multiplications are about as cheep (rather slightly cheaper) since the coefficients already have been decomposed to the polynomial components which gives low-cost SIMD multiplication code in either case.

6.5 Parallel Macaulay matrix multiplication in XL

The Parallelization of the Macaulay matrix multiplication of stages BW1 and BW3 is implemented in two ways: On multi-core architectures OpenMP is used to keep all cores busy, on cluster architectures MPI is used to communicate between the cluster nodes. Both approaches can be combined for clusters of multi-core nodes.

The efficiency of a parallelization of the Macaulay matrix multiplication depends on two factors: The workload must be balanced over all computing units (cores and/or nodes respectively) to fully exploit all available processor cores and the communication overhead must be small.

The strategy of the workload distribution is similar for both OpenMP and MPI. Figure 8 shows an example of a Macaulay matrix. Recall that each row has the same number of entries from the original quadratic system. Due to the structure of the matrix and the low row weight a splitting into column blocks would reduce load balancing and performance drastically. Therefore the workload is distributed by assigning blocks of rows of the Macaulay matrix to the computing units.

If the matrix is split into blocks of equal size, every unit has to compute the same number of multiplications. Nevertheless, due to the structure of the Macaulay matrix the runtime of the computing units may vary slightly: in the bottom of the Macaulay matrix it is more likely that neighbouring row blocks have non-zero entries in the same column. Therefore it is more likely to find the corresponding row of t_{old} in the caches and the computations can be finished faster than in the top of the Macaulay matrix. This imbalance may be addressed by dynamically assigning row ranges depending on the actual computing time of each block.

6.5.1 Parallelization for shared-memory systems: OpenMP

OpenMP offers a straightforward way to parallelize data-independent loops by adding an OpenMP compiler directive in front of a loop. This allows to easily assign blocks of rows of the Macaulay matrix to the processor cores: The outer loop which is iterating over the rows of the Macaulay matrix is parallelized using the directive "`#pragma omp parallel for`". This automatically assigns a subset of rows to each OpenMP thread.

It is not easy to overcome the above mentioned workload imbalance induced by caching effects since OpenMP does not allow to split row ranges into a fixed number of blocks of different sizes. The scheduling directive "`schedule(guided)`" gives a fair workload distribution; however, each processor core obtains several row ranges which are not spanning over consecutive rows. The outcome of this is a loss in cache locality. Thus the workload is fairly distributed but full performance is not achieved due to an increased number of cache misses. In fact, using "`schedule(guided)`" does not result in better performance than "`schedule(static)`". To achieve best performance the row ranges would need to be distributed according to the runtime of earlier iterations; however, it is not possible to express this with OpenMP in a straightforward way. Experiments showed that this results in a loss in performance of up to 5%.

Running one thread per virtual core on SMT architectures might increase the ALU exploitation but puts a higher pressure on the processor caches. Whether the higher efficiency outweighs the higher pressure on the caches needs to be tried out by experiments on each computer architecture for each problem size. Running two threads per physical core, i.e. one thread per virtual core, on an Intel Nehalem CPU increased performance by about 10% for medium sized systems. However, this advantage decreases for larger systems due to the higher cache pressure.

An OpenMP parallelization on UMA systems encounters no additional communication cost although the pressure on shared caches may be increased. On NUMA systems data must be distributed over the NUMA nodes in a way that takes the higher cost of remote memory access into account. Each row of the target matrix t_{new} is touched only once while the rows in the source matrix t_{old} may be touched several times. Therefore on NUMA systems the rows of t_{old} and t_{new} are placed on the NUMA node which accesses them first during the computation. This gives reasonable memory locality and also distributes memory accesses fairly between the memory controllers.

6.5.2 Parallelization for cluster systems: MPI

On an MPI cluster the workload is distributed similar to OpenMP by splitting the Macaulay matrix into blocks of rows. Since the computation on one row of the Macaulay matrix might depend on any of the rows of matrix t_{old}, the full matrix t_{old} needs to be available on all MPI nodes. This can be achieved by a blocking all-to-all communication after each iteration step of stages BW1 and BW3.

If B were a dense matrix, such a communication would take only a small portion of the overall runtime. But since B is a sparse Macaulay matrix which has a very low row weight, the computation time for one single row of B takes only a small amount of time. In fact this time is in the order of magnitude of the time that is necessary to send one row of t_{new} to all other nodes during the communication phase. Therefore this simple workload-distribution pattern gives a large communication overhead.

This overhead is hidden when communication is performed in parallel to computation. Todays high-performance network interconnects are able to transfer data via direct memory access (DMA) without interaction of the CPU. The computation of t_{new} can be split into several parts; during computation on one part, previously computed results are distributed to the other nodes and therefore are available at the next iteration step. Under the condition that computation takes more time than communication, the communication overhead can almost entirely be hidden. Otherwise speedup and therefore efficiency of cluster parallelization is bound by communication cost.

The version 2.2 of the MPI standard offers non-blocking communication primitives for point-to-point communication and gives easy access to the DMA capabilities of high-performance interconnects. Unfortunately there is no support for non-blocking collectives. Therefore a non-blocking `MPI_Iallgather` function was implemented that uses several non-blocking `MPI_Isend` and `MPI_Irecv` instructions. To ensure progress of the non-blocking communication, the MPI function `MPI_Test` must be called periodically for each transfer.

Apart from hiding the communication overhead it is also possible to totally avoid all communication by splitting t_{new} into independent column blocks. Therefore three communication strategies have been implemented which allow to either avoid all communication during stages BW1 and BW3 or to do computation and communication in parallel. All three approaches have certain advantages and disadvantages which make them suitable for different scenarios. The following paragraphs explain the approaches in detail:

1. **Operating on one shared column block of t_{old} and t_{new}:**

   For this approach the Macaulay matrix is split into blocks of rows in the same way as for the OpenMP parallelization. Each row of t_{new} is only touched once per iteration. Therefore each row can be send to the other cluster nodes immediately after the computation on it has finished.

   However, sending many small data packets has a higher overhead than sending few big packets. Therefore, the results of several consecutive rows are computed and sent together in an all-to-all communication: First the result of k rows is computed. Then a non-blocking communication for these k rows is initiated. While data is transfered, the next k rows are computed. At the end of each iteration step the nodes have to wait for the transfer of the last k rows to be finished; the last communication step is blocking.

   Finding the ideal number of rows in one packet for best performance poses a dilemma: On the one hand if k is too small, the communication overhead is increased since many communication phases need to be performed. On the other hand since the last communication step is blocking, large packages result in a long waiting time at the end of each iteration. Finding the best choice of the package size can be achieved by benchmarking the target hardware with the actual program code.

2. **Operating on two shared column blocks of t_{old} and t_{new}:**

   Both small packet size and blocking communication steps can be avoided by splitting the matrices t_{old} and t_{new} into two column blocks $t_{old,0}$ and $t_{old,1}$ as well as $t_{new,0}$ and $t_{new,1}$. The workload is distributed over the nodes row-wise as before. First each node computes the results of its row range for column block $t_{new,0}$ using rows from block $t_{old,0}$. Then a non-blocking all-to-all communication is initiated which distributes the results of block $t_{new,0}$ over all nodes. While the communication is going on, the nodes compute the results of block $t_{new,1}$ using data from block $t_{old,1}$. After computation on $t_{new,1}$ is finished, the nodes wait until the data transfer of block $t_{new,0}$ has been accomplished. Ideally communication of block $t_{new,0}$ is finished earlier than the computation of block $t_{new,1}$ so that the results of block $t_{new,1}$ can be distributed without waiting time while the computation on block $t_{new,0}$ goes on with the next iteration step.

   One disadvantage of this approach is that the entries of the Macaulay matrix need to be loaded twice per iteration, once for each block. This gives a higher memory contention and more cache misses than a single column block version. However, these memory accesses are sequential. Therefore it is likely that the access pattern can be detected by the memory interface and that the data is prefetched into the caches.

   Furthermore the width of the matrices t_{old} and t_{new} has an impact on the performance of the whole block Wiedemann algorithm. For BW1 and BW3 there is no big impact on the number of field-element multiplications which need to be performed since the number of iterations is decreased while the block width is increased; but altering the block size has an effect on memory efficiency due to cache effects. For the Berlekamp–Massey algorithm in step BW2 the width directly influences the number of multiplications, increasing the block width also increases the computation time.

   Therefore computing on two column blocks of t_{old} and t_{new} forces to either compute on a smaller block size (since t_{old} and t_{new} are split) or to increase the total matrix width; a combination of both is possible as well. Reducing the block size might impact the efficiency due to memory effects; enlarging the total matrix width increases the runtime of the Berlekamp–Massey algorithm. The best choice for the block size and therefore the matrix width must be determined by benchmarking.

3. **Operating on independent column blocks of t_{old} and t_{new}:**

   Any communication during stages BW1 and BW3 can be avoided by splitting the matrices t_{old} and t_{new} into independent column blocks for each cluster node. The nodes compute over the whole Macaulay matrix on a column stripe of t_{old} and t_{new}. All computation can be accomplished locally; the results are collected at the end of the computation of these stages.

   Although this is the most efficient parallelization approach when looking at communication cost, the per-node efficiency drops drastically with higher node count: For a high node count, the impact of the width of the column stripes of t_{old} and t_{new} becomes even stronger than for the previous approach. Therefore this approach only scales well for small clusters. For a large number of nodes, the efficiency of the parallelization declines significantly.

   Another disadvantage of this approach is that all nodes must store the whole Macaulay matrix in their memory. For large systems this is may not be feasible.

All three parallelization approaches have advantages and disadvantages; the ideal approach can only be found by testing each approach on the target hardware. For small clusters approach 3 might be the most efficient one although it looses efficiency due to the effect of the width of t_{old} and t_{new}. The performance of

approach 1 and approach 2 depends heavily on the network configuration and the ratio between computation time and communication time.

MPI does not offer the right communication primitives for the demands of a parallel implementation of XL. The blocking collective operations are fine for computation-bound applications; communication-bound sparse Macaulay matrix multiplication requires highly efficient non-blocking collective operations.

The structure of the Macaulay matrix accounts for further loss in parallelization efficiency: As described earlier even though the number of entries is equal in each row of the Macaulay matrix, due to memory caching effects the runtime might be different in different areas of the matrix. Runtime differences between cluster nodes can be straightened out by assigning a different amount of rows to each cluster node. Nevertheless, the parallelization approach 1 must initiate communication several times within one iteration step. Also approaches 2 and 3 have to call the `MPI_Test` function every once in a while during computation to guarantee communication progress. Due to the structure of the Macaulay matrix, these calls to the MPI-API occur out of sync between the nodes which might result in performance penalty.

Furthermore in case the cluster nodes are multi-core systems and OpenMP is used for local parallelization, interrupting the computation loop and resuming computation in another row range will have an impact on caching effects. In case the cluster nodes are NUMA systems, reducing NUMA effects becomes more challenging. Nevertheless, this can be avoided by running several MPI processes on each cluster node, one for each NUMA node; this in turn breaks the non-blocking communication scheme due to the lack of free background communication on most shared memory systems.

All parallelization approaches stated above are based on the memory-efficient Macaulay matrix representation described in Section 6.3. Alternatively the compact data format can be dropped for the favor of a standard sparse matrix data format. This gives the opportunity to optimize the structure of the Macaulay matrix for cluster computation. For example, a Macaulay matrix can be partitioned for parallel sparse matrix-vector multiplication using the *Mondriaan partitioner* of Bisseling, Fagginger Auer, Yzelman, and Vastenhouw available at [BFYV10]. Due to the low row-density, a repartitioning of the Macaulay matrix provides a better communication scheme: The data amount for communication is reduced compared to the current approaches and the costly all-to-all communication can be replaced by a few one-to-one communication paths. Even though this approach does not have an impact on small cluster sizes where efficiency and scalability are already very high, it becomes very important for large cluster sizes when the current communication scheme reaches its limits. This optimization for the communication scheme is not yet implemented; currently this parallel implementation of XL focuses on computational efficiency and memory efficiency and delivers a high performance on small cluster sizes. Optimizing the communication scheme and evaluating the trade-off between communication cost, memory demand, and computational efficiency is a major topic of future research.

All in all there are many pitfalls which reduce the efficiency of XL for a parallel implementation. The next section presents the performance numbers of the current implementation and gives an insight on how much these factors impact performance, scalability, and efficiency of a parallel XL computation.

7 Experimental results

This section gives an overview of the performance and the scalability of the XL implementation described in the previous sections. Experiments have been carried out on two computer systems: on a 48-core NUMA system and on a four node Infiniband cluster. Table 2 lists the key features of these systems.

To reduce the parameter space of the experiments, m was restricted to the smallest value allowed depending on n, thus $m = n$. On the one hand, the choice of m has an impact on the number of iterations of

BW1: a larger m reduces the number of iterations. On the other hand, a larger m increases the amount of computations and thus the runtime of BW2. Therefore, fixing m to $m = n$ does not result in the shortest overall runtime of all three steps of the block Wiedemann algorithm.

Three different experiments were executed: First a quadratic system of a moderate size with 16 equations and 14 variables was used to show the impact of block sizes n and $m = n$ on the block Wiedemann algorithm. The same system was then used to measure the performance of the three parallelization strategies for the large matrix multiplication in the steps BW1 and BW3. The third experiment used a second quadratic system with 18 equations and 16 variables to measure the performance of the parallelization on the cluster system with a varying number of cluster nodes and on the NUMA system with a varying number of NUMA nodes. The following paragraphs give the details of these experiments.

7.1 Impact of the block size

The impact of the block size of the block Wiedemann algorithm on the performance of the implementation was measured using a quadratic system with 16 equations and 14 variables over $\mathbb{F}_{16}$. In this case, the degree D_0 for the linearization is 9. The input for the algorithm is a square Macaulay matrix B with $N = 817190$ rows (and columns) and row weight $w_B = 120$.

Given the fixed size of the Macaulay matrix and $m = n$, the number of field operations for BW1 and BW2 is roughly the same for different choices of the block size n since the number of iterations is proportional to $1/n$ and the number of field operations per iteration is roughly proportional to n. However, the efficiency of the computation varies depending on n. The following paragraphs investigate the impact of the choice of n on each part of the algorithm.

Figure 9 shows the runtime for block sizes 32, 64, 128, 256, 512, and 1024. During the j-th iteration step of BW1 and BW3, the Macaulay matrix is multiplied with a matrix $t^{(j-1)} \in \mathbb{F}_{16}^{N \times n}$. For $\mathbb{F}_{16}$ each row of $t^{(j-1)}$ requires $n/2$ bytes of memory. In the cases $m = n = 32$ and $m = n = 64$ each row thus occupies less than one cache line of 64 bytes. This explains why the best performance in BW1 and BW3 is achieved for larger values of n. The runtime of BW1 and BW3 is minimal for block sizes $m = n = 256$. In this case one row of $t^{(j-1)}$ occupies two cache lines. The reason why this case gives a better performance than $m = n = 128$ might be that the memory controller is able to prefetch the second cache line. For larger values of m and n the performance declines probably due to cache saturation.

According to the asymptotic time complexity of Coppersmith's and Thomé's versions of the Berlekamp–Massey algorithm, the runtime of BW2 should be proportional to m and n. However, this turns out to be the case only for moderate sizes of m and n; note the different scale of the graph in Figure 9 for a runtime of more than 2000 seconds. For $m = n = 256$ the runtime of Coppersmith's version of BW2 is already larger than that of BW1 and BW3, for $m = n = 512$ and $m = m = 1024$ both versions of BW2 dominate the total runtime of the computation. Thomé's version is faster than Coppersmith's version for small and moderate block sizes. However, by doubling the block size, the memory demand of BW2 roughly doubles as well; Figure 10 shows the memory demand of both variants for this experiment. Due to the memory–time trade-off of Thomé's BW2, the memory demand exceeds the available RAM for a block size of $m = n = 512$ and more. Therefore memory pages are swapped out of RAM onto hard disk which makes the runtime of Thomé's BW2 longer than that of Coppersmith's BW2.

7.2 Performance of the Macaulay matrix multiplication

This experiment investigates which of the three different approaches for the cluster parallelization of Macaulay matrix multiplication gives the best performance. Figure 11 shows the runtime of each of the three ap-

proaches for BW1 and BW3 for an execution on four cluster nodes; the same system size was used as in the previous experiment. Since this experiment is only concerned about the parallelization of BW1 and BW3 and not about the performance of BW2, a block size of $m = n = 256$ was used.

Recall that in each iteration step j, the first approach distributes the workload for the Macaulay matrix multiplication row-wise over the cluster nodes and sends the results of the multiplication in the background of ongoing computation. This approach is called "1 block" in Figure 11. The second approach splits the workload similarly to the first approach but also splits $t^{(j-1)}$ and $t^{(j)}$ into two column blocks. The data of one column block is sent in the background of the computation of the other column block. Since $n = 256$, each of the two column blocks has a width of 128 elements. This approach is called "2 blocks" in the figure. The last approach splits the matrices $t^{(j-1)}$ and $t^{(j)}$ into as many column blocks as there are cluster nodes; each node computes independently on its own column block. The results are collected when the computations are finished. In this case the width of the column blocks is only 64 for 4 cluster nodes due to the fixed $n = 256$. This approach is called "independent blocks" in the figure.

The first approach requires blocking communication to send the last packet of rows. Therefore this approach has a longer runtime than the other two approaches. The approaches "2 blocks" and "independent blocks" have a similar runtime; the runtime of "independent blocks" is slightly longer since it uses a smaller block size for each of the column blocks. Note that the performance of this approach would be better for larger block sizes; however, this requires a larger (total) block size n and thus the benefits are outweighed by a longer runtime of BW2.

The approach "2 blocks" was used for the scalability tests that are described in the next paragraphs since it has a good performance for up to 4 cluster nodes and uses a fixed number of column blocks independent of the cluster size. Furthermore it uses a larger column-block size for a given n than the third approach with several independent column blocks.

7.3 Scalability experiments

The scalability was measured using a quadratic system with 18 equations and 16 variables over $\mathbb{F}_{16}$. The operational degree D_0 for this system is 10. The square Macaulay matrix B has a size of $N = 5311735$ rows and columns; the row weight is $w_B = 153$.

For this experiment, the implementation of the block Wiedemann algorithm ran on 1, 2, and 4 nodes of the cluster and on 1 to 8 CPUs of the NUMA system. Figure 12 gives an overview of the runtime of each step of the algorithm. Since this experiment is not concerned about peak performance but about scalability, a block size of $m = n = 256$ is used. The runtime on one cluster node is shorter than on one NUMA node since each cluster has more computing power than one NUMA node (see Table 2). At a first glance the implementation scales nicely: doubling of the core count roughly halves the runtime.

Given the runtime T_1 for one computing node and T_p for p computing nodes, the parallel efficiency E_p on the p nodes is defined as $E_p = T_1/pT_p$. Figures 13 and 14 give a closer look on the parallel speedup and the parallel efficiency of BW1 and BW3; the performance of BW3 behaves very similarly to BW1 and thus is not depicted in detail. These figures show that BW1 and Coppersmith's BW2 indeed have a nice speedup and efficiency on 2 and 4 cluster nodes. The parallelization of Thomé's BW2 however has only a moderate efficiency. In particular the polynomial multiplications require a more efficient parallelization approach.

On the NUMA system, BW1 achieves a good efficiency on up to 8 NUMA nodes. The workload was distributed such that each CPU socket was filled up with OpenMP threads as much as possible. Therefore in the case of two NUMA nodes (12 threads) the implementation achieves a high efficiency since a memory controller on the same socket is used for remote memory access and the remote memory access has only

moderate cost. For three and more NUMA nodes, the efficiency declines to around 80% due to the higher cost of remote memory access between different sockets.

Also on the NUMA system the parallelization of Thomé's BW2 achieves only a moderate efficiency. The parallelization scheme used for OpenMP does not scale well for a large number of threads.

The parallelization of Coppersmith's version of BW2 scales almost perfectly on the NUMA system. The experiment with this version of BW2 is performed using hybrid parallelization by running one MPI process per NUMA node and one OpenMP thread per core. The blocking MPI communication happens that rarely that it does not have much impact on the efficiency of up to 8 NUMA nodes.

8 Some Current Results and Summary

We tabulate results for some examples; so far the scaling is in accordance with our estimate that XL with Wiedemann is the best way to solve large generic systems. Specifically, we compare time and memory consumption between XL and F4 using Figure 15. We use data of our 48-core machine for XL, so the F4-data of time is divided by 48. Although we do not have F4-data for $n > 25$, it seems that XL will stably outperform F4 in both time and memory for $n > 23$.

Running time in seconds for XL with block Wiedemann for $\mathbb{F}_{16}$, $m/n = 2$

			colossus2				giants			
m	n	D_0	bw1	bw2	bw3	total	bw1	bw2	bw3	total
42	21	5	3	13	2	19	4	5	2	11
44	22	6	116	96	68	282	120	90	77	290
46	23	6	196	115	113	427	202	112	120	438
48	24	6	321	165	184	674	334	144	197	679
50	25	6	538	202	306	1050	546	189	315	1054
52	26	6	853	294	476	1628	864	232	494	1595
54	27	6	1336	333	758	2434	1372	305	779	2463
56	28	6	2086	486	1166	3745	2117	390	1188	3704
58	29	6	3197	523	1785	5516	3229	508	1803	5550
60	30	7	143325	4638	80198	228229	142239	3625	78034	223965
			colossus3							
42	21	5	3	12	2	17				
44	22	6	102	87	54	244				
46	23	6	172	101	92	367				
48	24	6	289	139	156	586				
50	25	6	475	176	251	904				
52	26	6	756	202	408	1378				
54	27	6	1193	271	629	2096				
56	28	6	1866	330	984	3183				
58	29	6	2836	373	1506	4719				
60	30	7	124589	2746	67594	194984				
62	31	7	145693	7640	105084	258518				

Running time in seconds for XL with block Wiedemann for $\mathbb{F}_2$, $m = n$

			colossus2				giants			
m	n	D_0	bw1	bw2	bw3	total	bw1	bw2	bw3	total
25	25	6	15	40	9	66	26	45	22	98
26	26	6	25	60	14	102	38	74	29	146
27	27	6	41	95	23	161	71	95	44	217
28	28	6	66	137	37	244	110	145	73	337
29	29	6	107	197	59	368	163	214	107	495
30	30	7	2395	3215	1285	6918	3529	3826	1911	9314
31	31	7	4124	5168	2161	11481	5987	6132	3201	15383
32	32	7	6864	8129	3610	18641	10214	9761	5412	25469
33	33	7	11182	12594	5969	29792				
34	34	7	18195	19286	9644	47184				
			colossus3							
29	29	6	89	148	50	293				
30	30	7	2013	1913	1109	5062				
31	31	7	3428	2998	1839	8300				
32	32	7	5753	4657	3136	13590				
33	33	7	9482	7076	5077	21692				
34	34	7	15494	10853	8282	34699				
35	35	7	24793	7430	13214	45571				

Running time in seconds for XL with block Wiedemann for $\mathbb{F}_{16}$, $m - n = 2$

			colossus2				giants			
m	n	D_0	bw1	bw2	bw3	total	bw1	bw2	bw3	total
15	13	8	21	45	12	78	19	44	13	78
16	14	9	374	244	219	839	338	215	204	761
17	15	9	1038	457	599	2096	931	340	548	1824
18	16	10	19809	2426	11071	33316	17188	1753	9953	28919
19	17	10	53543	4177	30682	88418	47627	2807	26573	77051
			colossus3							
15	13	8	18	45	10	73				
16	14	9	289	210	161	663				
17	15	9	798	330	441	1572				
18	16	10	15432	1485	8570	25501				
19	17	10	41618	2518	22838	66995				

Remarks: The machine colossus2 is a machine with 4 Opteron 6174 CPUs (48 cores) and 256GB of RAM; with a similar machine with only 64GB of RAM, which a local vendor just sent a quote to our institute for US$6,000, we could compute a solution from 30 variables in 60 equations in $\mathbb{F}_{16}$ in 2.6 days (using about 2^{57} field multiplications). The giants cluster is an infiniband-connected small cluster with 4 dual Xeon E5620 (2 quad-core 2.4GHz Westmeres) and 36GB of RAM each. The "colossus3" is the same machine as colossus2 with the CPUs replaced by the newer Bulldozer Opteron 6276 CPUs (32 pairs of 2.3GHz sesqui-cores, pseudo-cores sharing vector units).

Future Work: The programs we have run on other fields with various degree of optimization and speed. We expect to push these results to larger clusters, larger examples to verify the general validity of our approach, and contribute this work to a free software library when we have a sufficient coverage of parameters and base field arithmetics.

References

[AFI+04] Gwénolé Ars, Jean-Charles Faugère, Hideki Imai, Mitsuru Kawazoe, and Makoto Sugita. Comparison between XL and Gröbner basis algorithms. In Pil Lee, editor, *Advances in Cryptology – ASIACRYPT 2004*, volume 3329 of *Lecture Notes in Computer Science*, pages 157–167. Springer-Verlag Berlin Heidelberg, 2004.

[Ber66] Elwyn R. Berlekamp. Nonbinary BCH decoding, 12 1966.

[BFYV10] Rob H. Bisseling, Bas Fagginger Auer, Albert-Jan Yzelman, and Brendan Vastenhouw, 2 2010. `http://www.staff.science.uu.nl/~bisse101/Mondriaan`.

[CKPS00] Nicolas Courtois, Alexnader Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In Bart Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407. Springer-Verlag Berlin Heidelberg, 2000.

[Cop93] Don Coppersmith. Solving linear equations over GF(2): block Lanczos algorithm. *Linear algebra and its applications*, 192:33–60, 1993.

[Cop94] Don Coppersmith. Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm. *Mathematics of Computation*, 62(205):333–350, 1994.

[Cou03] Nicolas T. Courtois. Higher order correlation attacks, XL algorithm and cryptanalysis of Toyocrypt. In Pil Lee and Chae Lim, editors, *Information Security and Cryptology – ICISC 2002*, volume 2587 of *Lecture Notes in Computer Science*, pages 182–199. Springer-Verlag Berlin Heidelberg, 2003.

[CP02] Nicolas T. Courtois and Josef Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In Yuliang Zheng, editor, *Advances in Cryptology – ASIACRYPT 2002*, volume 2501 of *Lecture Notes in Computer Science*, pages 267–287. Springer-Verlag Berlin Heidelberg, 2002.

[Die04] Claus Diem. The XL-algorithm and a conjecture from commutative algebra. In Pil Lee, editor, *Advances in Cryptology – ASIACRYPT 2004*, volume 3329 of *Lecture Notes in Computer Science*, pages 146–159. Springer-Verlag Berlin Heidelberg, 2004.

[Fau99] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases (F_4). *Journal of Pure and Applied Algebra*, 139(1–3):61–88, 6 1999.

[Fau02] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases without reduction to zero (F_5). In *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation – ISSAC 2002*, pages 75–83, New York, NY, USA, 2002. ACM.

[KAF+10] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen Lenstra, Emmanuel Thomé, Joppe Bos, Pierrick Gaudry, Alexander Kruppa, Peter Montgomery, Dag Arne Osvik, Herman te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus. *Advances in Cryptology – CRYPTO 2010*, pages 333–350, 2010.

[Kal95] Erich Kaltofen. Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. *Mathematics of Computation*, 64(210):777–806, 1995.

[Laz83] Daniel Lazard. Gröbner-bases, Gaussian elimination and resolution of systems of algebraic equations. In J. A. van Hulzen, editor, *Proceedings of the European Computer Algebra Conference on Computer Algebra – EUROCAL '83*, volume 162 of *Lecture Notes in Computer Science*, pages 146–156. Springer-Verlag Berlin Heidelberg, 1983.

[Mas69] James L. Massey. Shift-register synthesis and BCH decoding. *IEEE Transactions on Information Theory*, 15(1):122–127, 1969.

[Moh01] Tzuong-Tsieng Moh. On the method of XL and its inefficiency to TTM, 1 2001.

[MR02] Sean Murphy and Matthew J. B. Robshaw. Essential algebraic structure within the AES. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag Berlin Heidelberg, 2002.

[MR03] Sean Murphy and Matthew J. B. Robshaw. Remarks on security of AES and XSL technique. *Electronics Letters*, 39(1):36–38, 1 2003.

[Pet11] Christiane Peters. Stellingen behorend bij het proefschrift *Curves, Codes, and Cryptography*, 2011.

[Sch77] Arnold Schönhage. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Informatica*, 7(4):395–398, 1977.

[Sch11] Peter Schwabe. Theses accompanying the dissertation *High-Speed Cryptography and Cryptanalysis*, 2011.

[Tho02] Emmanuel Thomé. Subquadratic computation of vector generating polynomials and improvement of the block Wiedemann algorithm. *Journal of Symbolic Computation*, 33(5):757–775, 5 2002.

[Vil97a] Gilles Villard. Further analysis of Coppersmith's block Wiedemann algorithm for the solution of sparse linear systems. In *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 32–39. ACM, 1997.

[Vil97b] Gilles Villard. A study of Coppersmith's block Wiedemann algorithm using matrix polynomials. Technical Report 975 IM, LMC-IMAG, 1997.

[Wie86] Douglas Wiedemann. Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory*, 32(1):54–62, 1986.

[YC04] Bo-Yin Yang and Jiun-Ming Chen. Theoretical analysis of XL over small fields. In *Information Security and Privacy – ACISP 2004*, volume 3108 of *Lecture Notes in Computer Science*, pages 277–288. Springer-Verlag Berlin Heidelberg, 2004.

[YC05] Bo-Yin Yang and Jiun-Ming Chen. All in the XL family: Theory and practice. In Choonsik Park and Seongtaek Chee, editors, *Information Security and Cryptology – ICISC 2004*, volume 3506 of *Lecture Notes in Computer Science*, pages 32–35. Springer-Verlag Berlin Heidelberg, 2005.

[YCBC07] Bo-Yin Yang, Owen Chia-Hsin Chen, Daniel J. Bernstein, and Jiun-Ming Chen. Analysis of QUAD. In Alex Biryukov, editor, *Fast Software Encryption*, volume 4593 of *Lecture Notes in Computer Science*, pages 290–308. Springer-Verlag Berlin Heidelberg, 2007.

[YCC04] Bo-Yin Yang, Jiun-Ming Chen, and Nicolas Courtois. On asymptotic security estimates in XL and Gröbner bases-related algebraic cryptanalysis. In Javier Lopez, Sihan Qing, and Eiji Okamoto, editors, *Information and Communications Security*, volume 3269 of *Lecture Notes in Computer Science*, pages 281–286. Springer-Verlag Berlin Heidelberg, 2004.

input : $H^{(j)} \in K^{(m+n)\times m}$
a list of nominal degrees $d^{(j)}$
output: $P^{(j)} \in K^{(m+n)\times(m+n)}$
$E^{(j)} \in K^{(m+n)\times(m+n)}$

1 $M \leftarrow H^{(j)}$, $P \leftarrow I_{m+n}$, $E \leftarrow I_{m+n}$;
2 sort the rows of M by the nominal degrees in decreasing order and apply the same permutation to $P^{(j)}$ and $E^{(j)}$;
3 **for** $k = 1 \rightarrow m$ **do**
4 **for** $i = (m + n + 1 - k)$ **downto** 1 **do**
5 **if** $M_{i,k} \neq 0$ **then**
6 $v_{(M)} \leftarrow M_i$, $v_{(P)} \leftarrow P_i$, $v_{(E)} \leftarrow E_i$;
7 **break**;
8 **for** $l = i + 1$ **to** $(m + n + 1 - k)$ **do**
9 $M_{l-1} \leftarrow M_l$, $P_{l-1} \leftarrow P_l$, $E_{l-1} \leftarrow E_l$;
10 $M_{(m+n+1-k)} \leftarrow v_{(M)}$, $P_{(m+n+1-k)} \leftarrow v_{(P)}$, $E_{(m+n+1-k)} \leftarrow v_{(E)}$;
11 **for** $l = 1$ **to** $(m + n - k)$ **do**
12 **if** $M_{l,k} \neq 0$ **then**
13 $M_l \leftarrow M_l - v_{(M)} \cdot (M_{l,k}/v_{(M)k})$;
14 $P_l \leftarrow P_l - v_{(P)} \cdot (M_{l,k}/v_{(M)k})$;
15 $P^{(j)} \leftarrow P$;
16 $E^{(j)} \leftarrow E$;

Figure 1: Gaussian Elimination in Coppersmith's Berlekamp–Massey algorithm

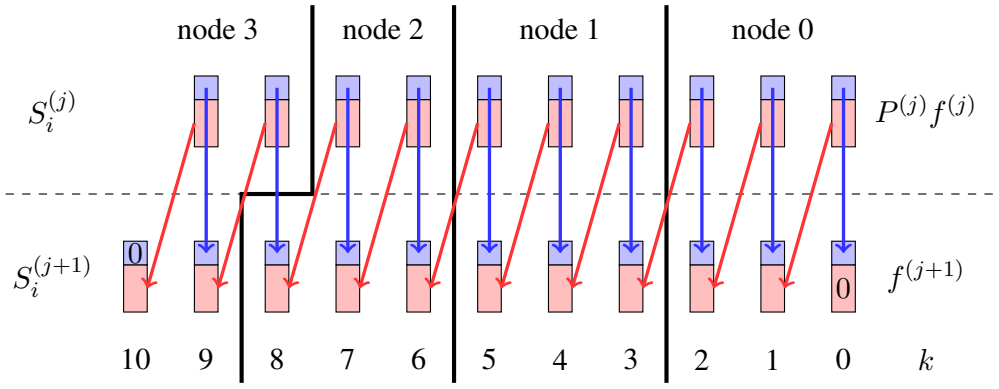


Figure 2: Example for the communication between 4 nodes. The top n rows of the coefficients are colored in blue, the bottom m rows are colored in red.

```
INPUT  GF(16) vector A
OUTPUT A * x

mask_a3     = 1000|1000|1000| ...
mask_a2a1a0 = 0111|0111|0111| ...

a3      = A AND mask_a3
tmp     = A AND mask_a2a1a0
tmp     = tmp << 1
new_a0  = a3 >> 3
tmp     = tmp XOR new_a0
add_a1  = a3 >> 2
ret     = tmp XOR add_a1

RETURN ret
```

Figure 3: Pseudocode for scalar multiplication by x.

```
INPUT  GF(16) vector A
       GF(16) element b
GLOBAL lookup table L
OUTPUT A * b

mask_low  = 00001111|00001111|00001111|...
mask_high = 11110000|11110000|11110000|...

low  = A AND mask_low
high = A AND mask_high
low  = PSHUFB(L[b], low)
high = high >> 4
high = PSHUFB(L[b], high)
high = high << 4
ret = low OR high

RETURN ret
```

Figure 4: Pseudocode for scalar multiplication using `PSHUFB`.

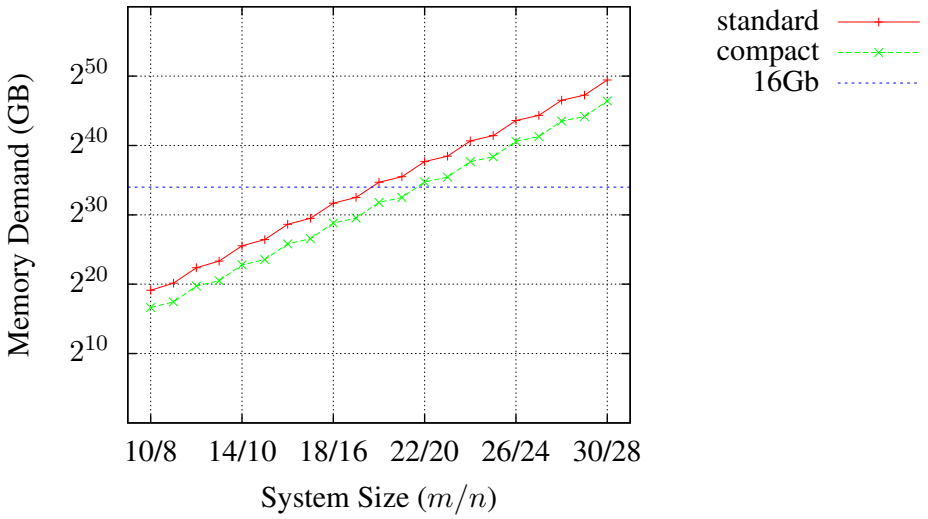


Figure 5: Memory demand of XL for several system sizes using $\mathbb{F}_{16}$ in standard and compact representation.

```
INPUT: macaulay_matrix<N, N> B;
       sparse_matrix<N, n> z;
matrix<N, n> t_new, t_old;
matrix<m, n> a[N/m + N/n + O(1)];
sparse_matrix<m, N, weight> x;

x.rand();
t_old = z;
for (unsigned i = 0; i <= N/m + N/n + O(1); i++)
{
    t_new = B * t_old;
    a[i] = x * t_new;
    swap(t_old, t_new);
}
RETURN a
```

Figure 6: Top-level iteration loop for BW1.

```
INPUT: macaulay_matrix<N, N> B;
       sparse_matrix<N, n> z;
       matrix_polynomial f;
matrix<N, n_sol> t_new, t_old;
matrix<N, n_sol> sol;

t_old = z * f[0].transpose();
for (unsigned k = 1; k <= f.deg; k++)
{
    t_new = B * t_old;
    t_new += z * f[k].transpose();
    [...] // check columns of t_new for solution
          // and cpoy found solutions to sol
    swap(t_new, t_old);
}
RETURN sol
```

Figure 7: Top-level iteration loop for BW3.

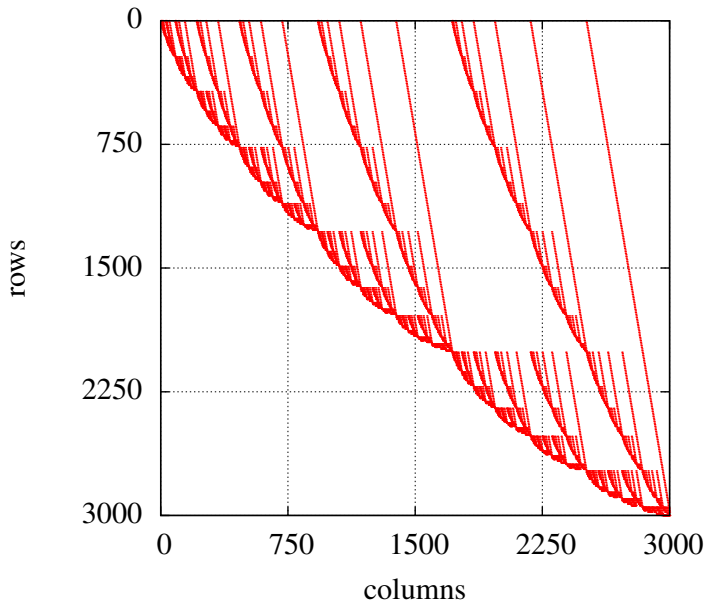


Figure 8: Plot of a Macaulay matrix over $\mathbb{F}_{16}$, 8 variables, 10 equations. Each row has 45 entries, 1947 rows have been dropped randomly to obtain a square matrix.

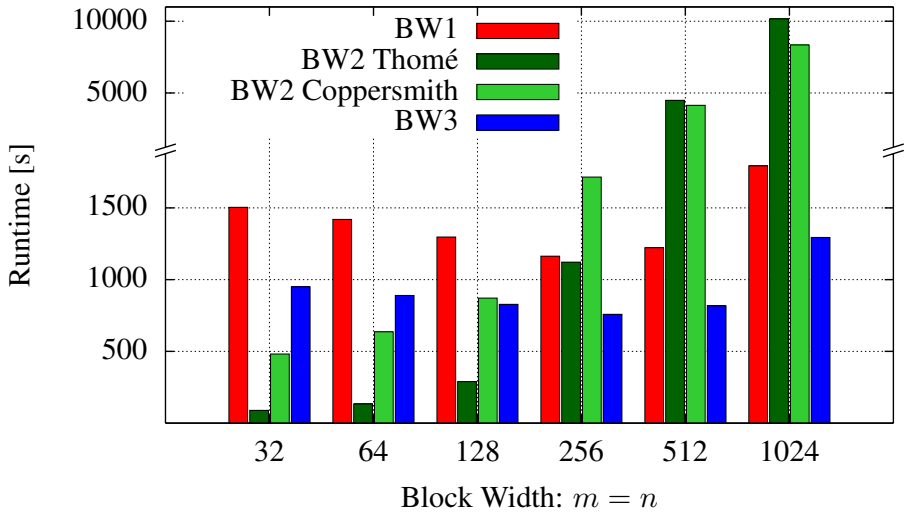


Figure 9: Runtime of XL 16-14 on one cluster node with two CPUs (8 cores in total) with different block sizes.

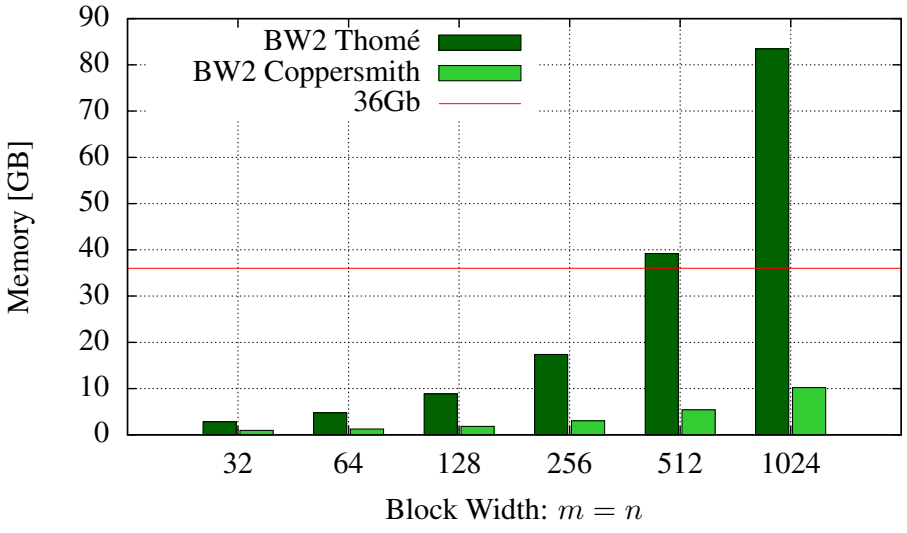


Figure 10: Memory consumption of XL 16-14 on one cluster node with 36 GB RAM.

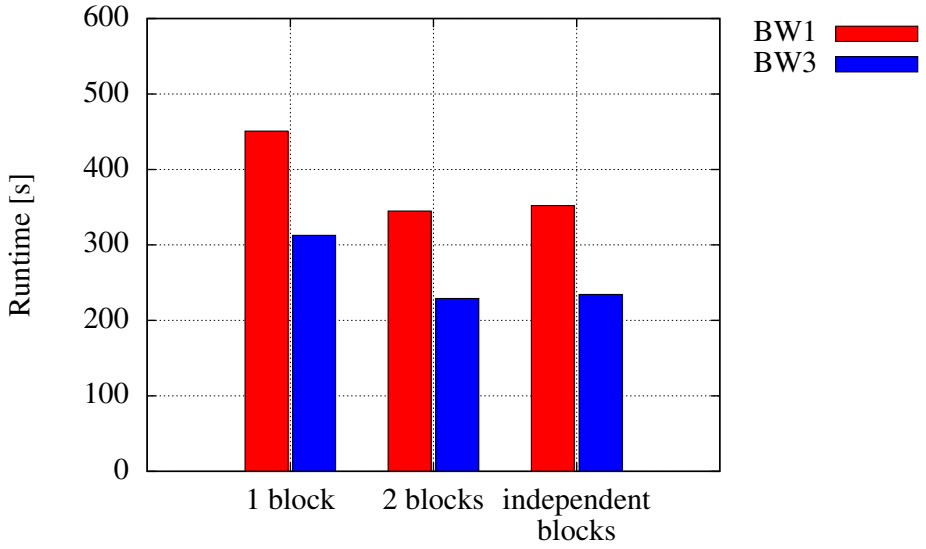


Figure 11: XL 16-14 on 4 cluster nodes using MPI with a total of 8 cores per node; one 1 block shared row-wise by all nodes, two blocks shared row-wise by all nodes, and independent blocks per node.

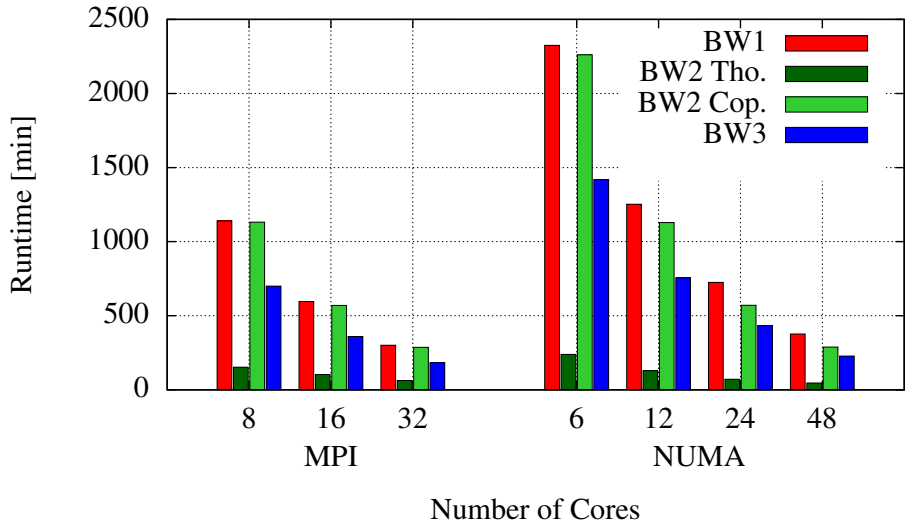


Figure 12: Runtime of XL 18-16 with Thomé's variant of BW2 on a 4 node cluster with 8 cores per node and on an 8 node NUMA systems with 6 cores per node.

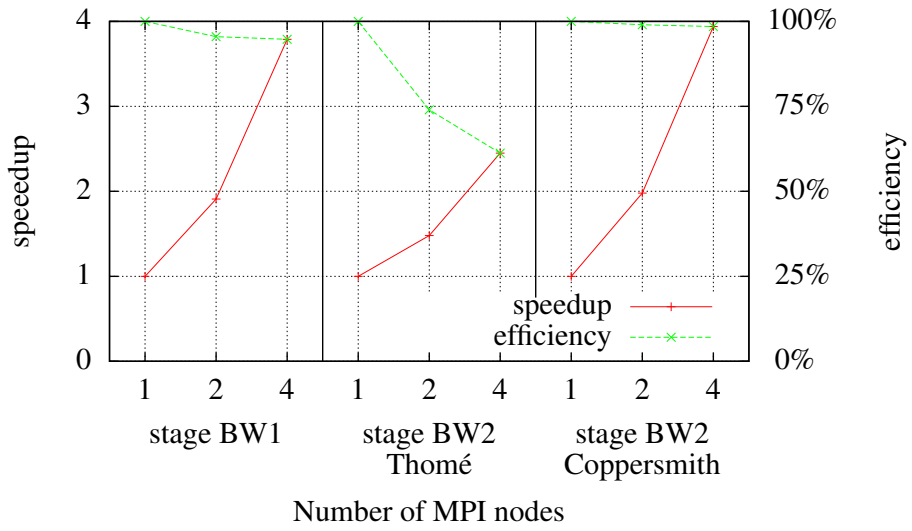


Figure 13: Speedup and efficiency of stages BW1 and BW2 on the MPI cluster.

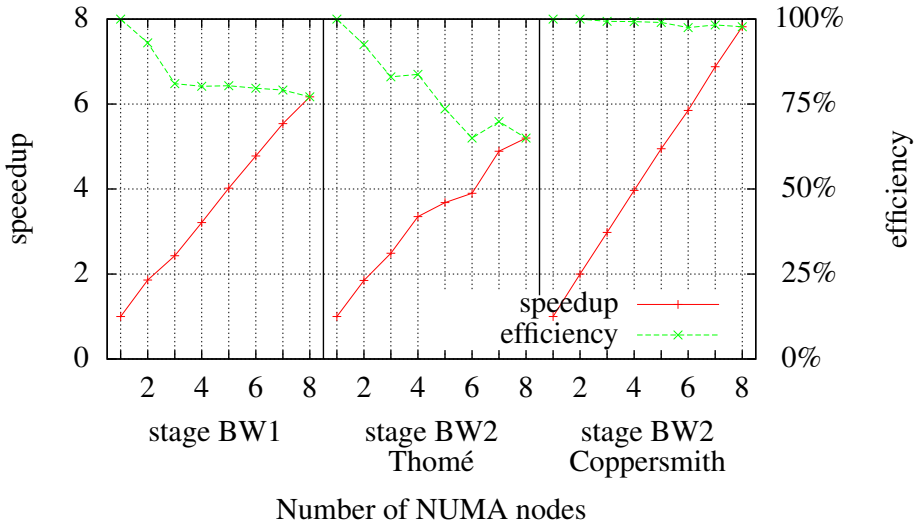


Figure 14: Speedup and efficiency of stages BW1 and BW2 on the NUMA system.

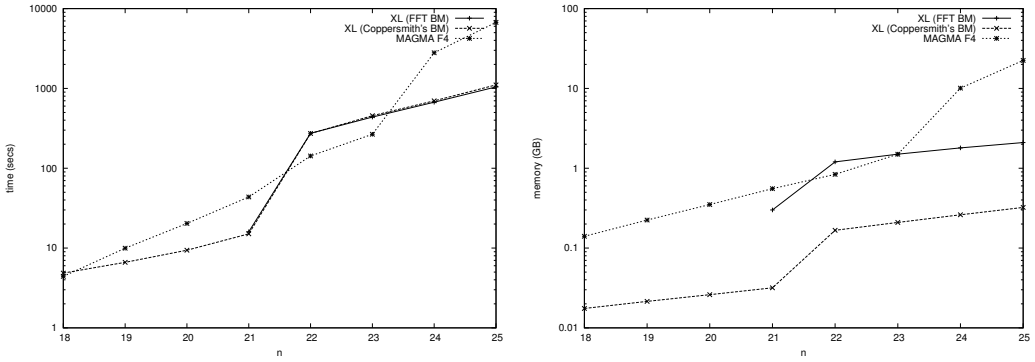


Figure 15: Time and Memory consumption of m=2n systems

iteration j	$S_3^{(j)}$	$S_2^{(j)}$	$S_1^{(j)}$	$S_0^{(j)}$	$\max(d^{(j)})$
0	$\emptyset$	$\emptyset$	{1}	{0}	1
1	$\emptyset$	{2}	{1}	{0}	2
2	{3}	{2}	{1}	{0}	3
3	{4}	{3}	{2}	{1,0}	4
4	{5}	{4}	{3,2}	{1,0}	5
5	{6}	{5,4}	{3,2}	{1,0}	6
7	{7,6}	{5,4}	{3,2}	{1,0}	7
...	...	...	...	...	...

Table 1: Example for the workload distribution over 4 nodes for Coppersmith's Berlekamp–Massey algorithm.

	NUMA	Cluster
CPU		
Name	AMD Opteron 6172	Intel Xeon E5620
Microarchitecture	Magny–Cours	Nehalem
Support for `PSHUFB`	✗	✓
Frequency	2100 MHz	2400 MHz
Memory Bandwidth per socket	2 × 25.6 GB/s	25.6 GB/s
Number of CPUs per socket	2	1
Number of cores per socket	12 (2 x 6)	4
Level 1 data cache size	12 × 64 KB	4 × 32 KB
Level 2 data cache size	12 × 512 KB	4 × 256 KB
Level 3 data cache size	2 × 6 MB	8 MB
Cache-line size	64 byte	64 byte
System Architecture		
Number of NUMA nodes	4 sockets × 2 CPUs	2 sockets × 1 CPU
Total number of cores	48	32
Number of cluster nodes	—	4
Network interconnect	—	Infiniband 40 GB/s
Memory		
Memory per CPU	32 GB	18 GB
Memory per cluster node	—	36 GB
Total memory	256 GB	144 GB

Table 2: Parameters of the computer architectures used for the experiments.

Experimentally Verifying a Complex Algebraic Attack on the Grain-128 Cipher Using Dedicated Reconfigurable Hardware

Itai Dinur[1], Tim Güneysu[2], Christof Paar[2],
Adi Shamir[1], and Ralf Zimmermann[2]

[1] Computer Science department, The Weizmann Institute, Rehovot, Israel
[2] Horst Görtz Institute for IT Security, Ruhr-University Bochum, Germany

Abstract. In this work, we describe the first single-key attack on the full version of Grain-128 that can recover arbitrary keys. Our attack is based on a new version of a cube tester, which is a factor of about 2^{38} faster than exhaustive search. To practically verify our results, we implemented the attack on the reconfigurable hardware cluster RIVYERA and tested the main components of the attack for dozens of random keys. Our experiments successfully demonstrated the correctness and expected complexity of the attack by finding a very significant bias in our new cube tester for about 7.5% of the tested keys. This is the first time that the main components of a complex analytical attack against a digital full-size cipher were successfully implemented using special-purpose hardware, truly exploiting the reconfigurable nature of an FPGA-based cryptanalytical device.

Keywords: Special-purpose hardware, Grain-128, stream cipher, cryptanalysis, cube attacks, cube testers.

1 Introduction

Special-purpose hardware, i. e., computing machines dedicated to cryptanalytical problems, have a long tradition in code-breaking, including attacks against the Enigma cipher during WWII [3]. Their use is promising if two conditions are fulfilled. First, the complexity of the cryptanalytical problem must be - as of today - in the range of approximately $2^{50} \dots 2^{64}$ operations. For problems with a lower complexity, conventional computer clusters are typically sufficient, such as the linear cryptanalysis attack against DES [4] (which required 2^{43} DES evaluations), and more than 2^{64} operations are difficult to achieve with today's technology unless extremely large budgets are available. The second condition is that the computations involved are suited for customized hardware architectures, which is often the case in symmetric cryptanalysis. Both conditions are fulfilled for the building blocks of the Grain-128 attack described in this paper.

Grain-128 [1] is a 128-bit variant of the Grain scheme which was selected by the eSTREAM project in 2008 as one of the three recommended hardware-efficient stream ciphers. The only single-key attacks published so far on this scheme which were substantially faster than exhaustive search were either on a reduced number of rounds or on a specific class of weak keys which contains about one in a thousand keys.

Contribution of this work: In this paper, we describe the first attack which can be applied to the full scheme of Grain-128 with arbitrary keys. It uses an improved cube distinguisher with new dynamic variables, which makes it possible to attack Grain-128 with no restriction on the key. Its main components were experimentally verified by running a 50-dimensional cube tester for 107 random keys and discovering a very strong bias (of 50 zeroes out of 51 bits) in about 7.5% of these keys. For these keys, we expect the running time of our new attack to be about 2^{38} times faster than exhaustive search, using 2^{63} bits of memory. Our attack is thus both faster and more general than the best previous attack on Grain-128 [2], which was a weak-key attack on one in a thousand keys which was only 2^{15} times faster than exhaustive search. However, our attack does not seem to threaten the security of the original 80-bit Grain scheme.

In order to develop and experimentally verify the main components of the attack, we had to run thousands of summations over cubes of dimension 49 and 50 for dozens of randomly chosen keys, where each summation required the evaluation of 2^{49} or 2^{50} output bits of Grain-128 (running the time-consuming initialization phase of Grain-128 for about 2^{56} different key and IV values). This process is hardware-oriented, highly

parallelizable, and well beyond the capabilities of a standard cluster of PC's. We thus decided to implement the attack on a special-purpose hardware cluster.

Even though it is widely speculated that government organizations have been using special-purpose hardware for a long time, there are only few confirmed reports about cryptanalytical machines in the open literature. In 1998, Deep Crack, an ASIC-based machine dedicated to brute-forcing DES, was introduced [5]. In 2006, COPACOBANA also allowed exhaustive key searches of DES, and in addition cryptanalysis of other ciphers [6]. However, in the latter case often only very small-scale versions of the ciphers are vulnerable. The paper at hand extends the previous work with respect to cryptanalysis with dedicated hardware in several ways. Our work is the first time that the main components of a complex analytical attack, i. e., not merely an exhaustive search, are successfully realized in a public way against a full-size cipher by using a special-purpose machine (previous attacks were either a simple exhaustive search sped up by a special-purpose hardware, or advanced attacks such as linear cryptanalysis which were realized in software on multiple workstations). Also, this is the first attack which makes use of the reconfigurable nature of the hardware. Our RIVYERA computer, consisting of 128 large FPGAs, is the most powerful cryptanalytical machine available outside government agencies (possessing more than four times as many logic resources as the COPACOBANA machine). This makes our attack an interesting case study about what type of cryptanalysis can be done with "university budgets" (as opposed to government budgets). As a final remark, it is worth noting that the same attack implemented on GPU clusters would require an extremely large number of graphic cards, which would not only require a very high budget but would consume considerably more electric energy to perform the same computations.

Outline: In Section 2, we give the necessary background regarding Grain-128, dynamic cube attacks and describe the new attack on Grain-128. Then we present our implementation of the attack in Section 3, before we present our results and conclusions in Sections 4 and 5.

2 Background

In this section we briefly discuss the background required in the remainder of this work.

2.1 Grain-128 Stream Cipher

The state of Grain-128 consists of a 128-bit LFSR and a 128-bit NFSR. The feedback functions of the LFSR and NFSR are respectively defined to be
$s_{i+128} = s_i + s_{i+7} + s_{i+38} + s_{i+70} + s_{i+81} + s_{i+96}$
$b_{i+128} = s_i + b_i + b_{i+26} + b_{i+56} + b_{i+91} + b_{i+96} + b_{i+3}b_{i+67} + b_{i+11}b_{i+13} + b_{i+17}b_{i+18} + b_{i+27}b_{i+59} + b_{i+40}b_{i+48} + b_{i+61}b_{i+65} + b_{i+68}b_{i+84}$
The output function is defined as
$z_i = \sum_{j\in\mathcal{A}} b_{i+j} + h(x) + s_{i+93}$, where $\mathcal{A} = \{2, 15, 36, 45, 64, 73, 89\}$.
$h(x) = x_0x_1 + x_2x_3 + x_4x_5 + x_6x_7 + x_0x_4x_8$
where the variables x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7 and x_8 correspond to the tap positions b_{i+12}, s_{i+8}, s_{i+13}, s_{i+20}, b_{i+95}, s_{i+42}, s_{i+60}, s_{i+79} and s_{i+95} respectively.

Grain-128 is initialized with a 128-bit key that is loaded into the NFSR, and with a 96-bit IV that is loaded into the LFSR, while the remaining 32 LFSR bits are filled with 1's. The state is then clocked through 256 initialization rounds without producing an output, feeding the output back into the input of both registers.

2.2 Dynamic Cube Attacks

Cube Testers In almost any cryptographic scheme, each output bit can be described by a multivariate master polynomial $p(x_1, .., x_n, v_1, .., v_m)$ over GF(2) of secret variables x_i (key bits), and public variables v_j (plaintext bits in block ciphers and MACs, IV bits in stream ciphers). The cryptanalyst is allowed to tweak this master polynomial by assigning chosen values to the public variables (which result in multiple derived polynomials).

To simplify our notation, we ignore in the rest of this subsection the distinction between public and private variables. Given a multivariate master polynomial with n variables $p(x_1, .., x_n)$ over GF(2) in algebraic normal form (ANF), and a term t_I containing variables from an index subset I that are multiplied together, the polynomial can be written as the sum of terms which are supersets of I and terms that miss at least one variable from I:

$$p(x_1, .., x_n) \equiv t_I \cdot p_{S(I)} + q(x_1, .., x_n)$$

$p_{S(I)}$ is called the *superpoly* of I in p. Compared to p, the algebraic degree of the superpoly is reduced by at least the number of variables in t_I, and its number of terms is smaller.

Cube testers [7] are related to high order differential attacks [8]. The basic idea behind them is that the symbolic sum over GF(2) of all the derived polynomials obtained from the master polynomial by assigning all the possible 0/1 values to the subset of variables in the term t_I is exactly $p_{S(I)}$ which is the superpoly of t_I in $p(x_1, .., x_n)$. This simplified polynomial is more likely to exhibit non-random properties than the original polynomial P.

Cube testers work by evaluating superpolys of carefully selected terms t_I which are products of public variables, and trying to distinguish them from a random function. One of the natural properties that can be tested is balance: A random function is expected to contain as many zeroes as ones in its truth table. A superpoly that has a strongly unbalanced truth table can thus be used to distinguish the cryptosystem from a random polynomial.

Dynamic Cube Attacks Dynamic Cube Attacks exploit distinguishers obtained from cube testers to recover some secret key bits. In static cube testers (and other related attacks such as the original cube attack [10], and AIDA [9]), the values of all the public variables that are not summed over are fixed to a constant (usually zero), and thus they are called static variables. However, in dynamic cube attacks the values of some of the public variables that are not part of the cube are not fixed. Instead, each one of these variables (called dynamic variables) is assigned a function that depends on some of the cube public variables and on some private variables. Each such function is carefully chosen in order to simplify the resultant superpoly and thus to amplify the expected bias (or the non-randomness in general) of the cube tester.

The basic steps of the attack are briefly summarized below (for more details refer to [2], where the notion of dynamic cube attacks was introduced).

Preprocessing Phase We first choose some polynomials that we want to set to zero at all the vertices of the cube, and show how to nullify them by setting certain dynamic variables to appropriate expressions in terms of the other public and secret variables. To minimize the number of evaluations of the cryptosystem, we choose a big cube of dimension d and a set of subcubes to sum over during the online phase. We usually choose the subcubes of the highest dimension (namely d and $d-1$), which are the most likely to give a biased sum. We then determine a set of e expressions in the private variables that need to be guessed by the attacker in order to calculate the values of the dynamic variables during the cube summations.

Online Phase The online phase of the attack has two steps that are described in the following.

Step 1: The first step consists again of two substeps:

1. For each possible vector of values for the e secret expressions, sum modulo 2 the output bits over the subcubes chosen during preprocessing with the dynamic variables set accordingly, and obtain a list of sums (one bit per subcube).
2. Given the list of sums, calculate its score by measuring the non-randomness in the subcube sums. The output of this step is a sequence of lists sorted from the lowest score to the highest (in our notation the list with the lowest score has the largest bias, and is thus the most likely to be correct in our attack).

Given that the dimension of our big cube is d, the complexity of summing over all its subcubes is bounded by $d2^d$ (using the Moebius transform [11]). Assuming that we have to guess the values of e secret expressions in order to determine the values of the dynamic variables, the complexity of this step is bounded by $d2^{d+e}$ bit operations. Assuming that we have y dynamic variables, both the data and memory complexities are bounded by 2^{d+y} (since it is sufficient to obtain an output bit for every possible vertex of the cube and for every possible value of the dynamic variables).

Step 2: Given the sorted guess score list, we determine the most likely values for the secret expressions, for a subset of the secret expressions, or for the entire key. The specific details of this step vary according to the attack.

Partial Simulation Phase The complexity of executing online step 1 of the attack for a single key is $d2^{d+e}$ bit operations and 2^{d+y} cipher executions. In the case of Grain-128, these complexities are too high and thus we have to experimentally verify our attack with a simpler procedure. Our solution is to calculate the cube summations in online step 1 only for the correct guess of the e secret expressions. We then calculate the score of the correct guess and estimate its expected position g in the sorted list of score values by assuming that incorrect guesses will make the scheme behave as a random function. Consequently, if the cube sums for the correct guess detect a property that is satisfied by a random cipher with probability p, we estimate that the location of the correct guess in the sorted list will be $g \approx \max\{p \times 2^e, 1\}$ (as justified in [2]).

2.3 Dynamic Cube Attacks on Grain-128

The parameter set we use for our attack is described — and its choice justified — in [12], and is given here again in Table 1 for the sake of completeness. However, since we focus on the implementation of the attack, we omit it from this paper. Moreover, we specify only the details of the online phase of the attack that are most relevant to our hardware implementation. For the complete details, refer to [12].

Table 1. Parameter set for the attack on the full Grain-128, given output bit 257.

Cube Indexes	{0,2,4,11,12,13,16,19,21,23,24,27,29,33,35,37,38,41,43,44,46, 47,49,52,53,54,55, 57,58,59,61,63,65,66,67,69,72,75,76,78,79,81,82,84,85,87,89,90,92,93}
Dynamic Variables	{31,3,5,6,8,9,10,15,7,25,42,83,1}
State Bits Nullified	$\{b_{159}, b_{131}, b_{133}, b_{134}, b_{136}, b_{137}, b_{138}, b_{145}, s_{135}, b_{153}, b_{170}, b_{176}, b_{203}\}$

Before executing the online phase of the attack, one should take some preparation steps (given in [12])in order to determine the list of $e = 39$ secret expressions in the key variables we have to guess during the actual attack. Online step 1 of the attack is given below:

1. Obtain the first output bit produced by Grain-128 (after the full 256 initialization steps) with the fixed secret key and all the possible values of the variables of the big cube and the dynamic variables given in Table 1 (the remaining public variables are set to zero). The dimension of the big cube is 50 and we have 13 dynamic variables and thus the total amount of data and memory required is $2^{50+13} = 2^{63}$ bits.
2. We have 2^{39} possible guesses for the secret expressions. Allocate a guess score array of 2^{39} entries (an entry per guess). For each possible value (guess) of the secret expressions:
   (a) Plug the values of these expressions into the dynamic variables (which thus become a function of the cube variables, but not the secret variables).
   (b) Our big cube in Table 1 is of dimension 50. Allocate an array of 2^{50} bit entries. For each possible assignment to the cube variables:
      i. Calculate the values of the dynamic variables and obtain the corresponding output bit of Grain-128 from the data.
      ii. Copy the value of the output bit to the array entry whose index corresponds to the assignment of the cube variables.
   (c) Given the 2^{50}-bit array, sum over all the entry values that correspond to the 51 subcubes of the big cube which are of dimension 49 and 50. When summing over 49-dimensional cubes, keep the cube variable that is not summed over to zero. This step gives a list of 51 bits (subcube sums).
   (d) Given the 51 sums, calculate the score of the guess by measuring the fraction of bits which are equal to 1. Copy the score to the appropriate entry in the guess score array and continue to the next guess (item 2). If no more guesses remain go to the next step.

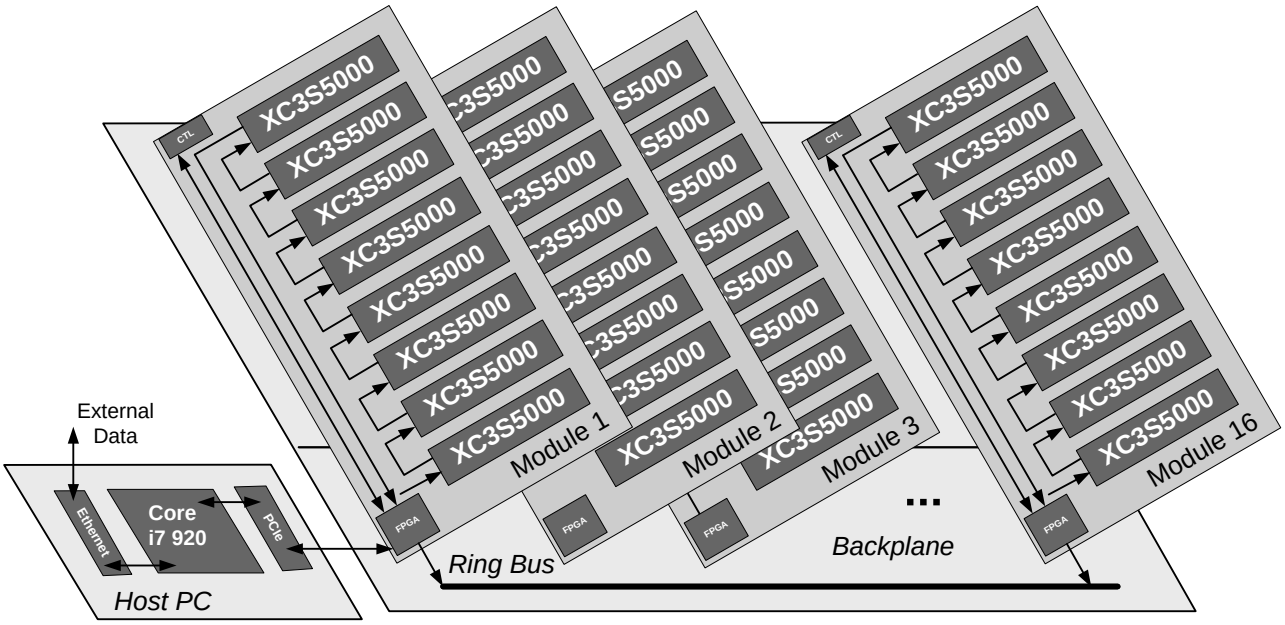


Fig. 1. Architecture of the RIVYERA cluster system

3. Sort the 2^{39} guess scores from the lowest score to the highest.

The total complexity of algorithm above is about $50 \times 2^{50+39} < 2^{95}$ bit operations (it is dominated by item 2.c, which is performed once for each of the 2^{39} possible secret expression guesses).

Given the sorted guess array which is the output of online step 1, we are now ready to perform online step 2 of the attack (which recovers the secret key without going through the difficult step of solving the large system of polynomial equations). The details of online step 2 are given in [12], which also shows that the total complexity the algorithm is equivalent to about $g \times 2^{90}$ cipher evaluations.

The attack is worse than exhaustive search if we have to try all the 2^{39} possible values of g, and thus it is crucial to provide strong experimental evidence that g is relatively small for a large fraction of keys. In order to estimate g, we executed the online part of the attack by calculating the score for the correct guess of the 39 expression values, and estimating how likely it is to get such a bias for incorrect guesses if we assume that they behave as random functions.

The simulation algorithm is a simplified version of item 2 in online step 1 (performed only for the correct key), and is described in Algorithm 2 as shown in the appendix. We performed this simulation for 107 randomly chosen keys, out of which 8 gave a very significant bias in which at least 50 of the 51 cubes sums were zero. This is expected to occur in a random function with probability $p < 2^{-45}$, and thus we estimate that for about 7.5% of the keys, $g \approx \max\{2^{-45} \times 2^{39}, 1\} = 1$ and thus the correct guess of the 39 secret expressions will be the first in the sorted score list (additional keys among those we tested had smaller biases, and thus a larger g). The complexity of online step 2 of the attack is thus expected to be about 2^{90} cipher executions, which dominates the complexity of the attack (the complexity of online step 1 is about 2^{95} bit operations, which we estimate as $2^{95-10} = 2^{85}$ cipher executions). This gives an improvement factor of 2^{38} over the 2^{128} complexity of exhaustive search for a non-negligible fraction of keys, which is significantly better than the improvement factor of 2^{15} announced in [2] for the small subset of weak keys considered in that attack. We note that for most additional keys there is a continuous tradeoff between the fraction of keys that we can attack and the complexity of the attack on these keys.

2.4 RIVYERA Special-Purpose Hardware Cluster

In this work, we employ an enhanced version of the COPACOBANA special-purpose hardware cluster that was specifically designed for the task of cryptanalysis [6]. This enhanced cluster (also known as RIVYERA [13]) is populated with 128 Spartan-3 XC3S5000 FPGAs, each tightly coupled with 32MB memory. Each Spartan-3 XC3S5000 FPGA provides a sea of logic resources consisting of 33,280 slices and 104 BRAMs enabling the implementation even of complex functions in reconfigurable hardware. Eight FPGAs are soldered on individual card modules that are plugged into a backplane which implements a global systolic ring bus for high-performance communication. The internal ring bus is further connected via PCI Express to a host PC which is also installed in the same 19" housing of the cluster. Figure 1 provides an overview of the architecture of the RIVYERA special-purpose cluster.

3 Implementation

To get a better understanding of our implementation and the design decisions, we need to examine the steps of the algebraic attack from Section 2.3. We start by formulating the steps from an implementation point-of-view and discuss the workflow in more detail, briefly discuss the different implementation options and finally describe our implementation on an FPGA cluster.

3.1 Analysis of the Algorithm

Algorithm 1 describes the attack with respect to its implementation in hardware.

Algorithm 1 Dynamic Cube Attack Simulation (Algorithm 2), Optimized for Implementation

Input: 96 bit integer *baseIV*, cube dimension d, cube $C = \{C_0, \ldots, C_d\}$ with $0 \leq C_i < 96\ \forall C_i \in C$, number of polynomials m, dynamic variable indices $D = \{D_0, \ldots, D_m\}$ with $0 \leq C_i < 96\ \forall D_i \in D$, state bit indices $S = \{S_0, \ldots, S_m\}$ with $0 \leq S_i < 96\ \forall S_i \in S$.
Output: $(d+1)$ bit cubesum s

1: IV $\leftarrow$ *baseIV*
2: $s \leftarrow 0$.

Key Selection
3: Choose random 128 bit key K.
4: Choose key-dependent polynomials $P_j(X)$ nullifying state bits S_j.

Computation
5: **for** $i \leftarrow 0$ **to** $2^d - 1$ **do**
6: **for** $j \leftarrow 0$ **to** $d - 1$ **do**
7: setBit(IV, C_j, getBit(i, j))
8: **end for**
9: **for** $j \leftarrow 0$ **to** $m - 1$ **do**
10: setBit(IV, D_j, $P_j(i)$)
11: **end for**
12: $ks \leftarrow$ first bit of Grain-128(IV, K) keystream
13: **if** $ks = 1$ **then**
14: $s \leftarrow s \oplus (1|\text{not}(i))$
15: **end if**
16: **end for**
17: **return** s.

To simplify the description, we use the function `getBit(int, pos)`, returning the bit at position `pos` of the integer `int`, and `setBit(int, pos, val)`, setting the bit at position `pos` of integer `int` to `val`.

All input arguments are included in the parameter set in Table 1. While they are fixed by the Table, we have to keep in mind that the attack is also an experimental verification of this parameter set. Thus it is important that the implementation should allow changes and pose as few restrictions as possible to these values. The algorithm will compute the cube sum of $d + 1$ bits, i. e., of size 51 bits with our parameters.

After selecting the key we wish to attack, we need to choose the polynomials, which nullify certain state bits and reset the IV to a default value. In the loop starting at line 5, we iterate over all 2^d combinations. Each time, we modify the IV by spreading the current iteration value over the cube positions (line 7) and evaluate the polynomials - boolean functions depending on these changing positions - and store the resulting bit per function at the dynamic variable positions (line 10). Now that the IV is prepared, we run a full initialization (256 rounds) of Grain-128 (line 12 and - in case the first keystream bit is not zero - we XOR the current sum with the inverse of the d bit iteration count, prefixed by a 1 (line 14).

Figure 2 describes the basic workflow of an implementation: It uses a parameter set as input, e. g., the cube dimension, the cube itself, a base IV and the number of keys to attack. It selects a random key to attack and divides the big cube into smaller worker cubes and distributes them to worker threads running in parallel. Please note that for simplicity the figure shows only one worker. If 2^w workers are used in parallel, the iterations per worker are reduced from 2^d to 2^{d-w}.

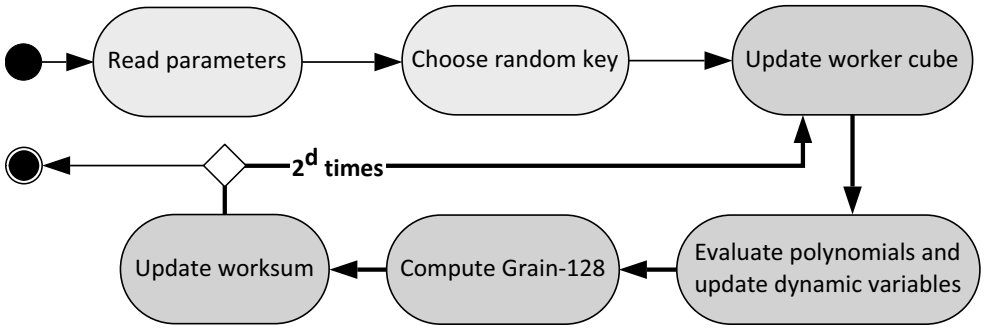


Fig. 2. Cube Attack — Program flow for cube dimension d.

The darker nodes and the bold path show the steps of each independent thread: As each worker iterates over a distinct subset of the cube, it evaluates polynomials on the worker cube (dynamic variables) and updates the IV input to Grain-128. Using this generated IV and the random key, it computes the output of Grain-128 after the initialization phase. With this output, the thread updates an intermediate value — the worker sum — and starts the next iteration. In the end, the software combines all worker sums, evaluates the result and may chose a new random key to start again.

We can see that the algorithm is split into three parts: First, we manipulate the worker cube positions and derive an IV from it. Then, we compute the output of the Grain-128 stream cipher using the given key, data and derived IV. Before we start the next iteration, the worksum is updated.

Grain-128: The second part is straight-forward and seems to be the main computational task. It concerns only the Grain implementation: With a cube of dimension d, the attack on one key computes the first output bit of Grain-128 2^d times. As we already need 2^{50} iterations with the current parameter set, it is necessary to implement Grain-128 as efficiently as possible in order to speed up the attack.

Taking a closer look at the design of the stream cipher (see Section 2.1), it yields much potential for an efficient hardware implementation: It consists mainly of two shift registers and some logic cells. While using bit-slicing techniques potentially decreases the overhead of CPUs when computing expensive bit-manipulations, Aumasson *et al.*already proposed a fast and small FPGA implementation as a good choice when implementing cube testers on Grain-128 in [14].

IV Generation: To create an independent worker, it also needs to include the IV generation. This process takes the default IV and modifies $d + m$ bits, which is easily done in software by storing the generated IV as an array and accessing the positions directly. Changing the parameters to compute larger cube dimensions d, to allow more than m polynomials poses no problem either.

Considering a possible hardware implementation, this increases the complexity a lot. In contrast to the software design, we cannot create a generic layout, which reads the parameter set: We need multiplexers for all IV bits to allow dynamic choices and even more multiplexers to allow all possible combinations of boolean functions to support all possible polynomials.

As this problem seems very easy in software and difficult in hardware, a software approach seems more reasonable. Nevertheless, the communication overhead to supply many workers with new IVs every few clock cycles explodes. To estimate the effort of building an independent worker in hardware, we need to know how many dynamic inputs we have to consider in the IV generation process, as these modifications are very inefficient in hardware: In order to compute the cipher, we need a key and an IV. The value of the key varies, as it is chosen at random.

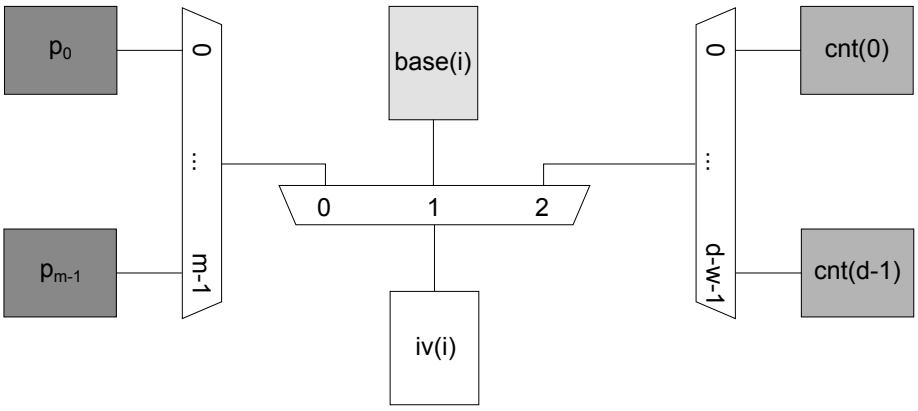


Fig. 3. Necessary Multiplexers for each IV bit (without Optimizations) of a worker with worker cube size $d - w$ and m different polynomials. This is an $(m + d - w + 1)$-to-1 bit multiplexer, i. e., with the current parameter set a $(64 - w)$-to-1 bit multiplexer.

The IV is a 96 bit value, where each bit utilizes one of three functions as Figure 3 shows: it is either a value given by the base IV (light grey) provided by the parameter set, part of the current counter spread across the worker cube (grey) or a dynamic variable (dark grey). As the function of each bit differs not only per parameter set, but also when assigning partial cubes to different workers, this input also varies and we need to create an $(m + d - w + 1)$-to-1 bit multiplexer for each bit, resulting in $96 \times (64 - w)$-to-1 bit multiplexers for our current parameter set.

The first two functionalities are both restricted and can be realized by simple multiplexers in hardware. The dynamic variable on the other hand stores the result of a polynomial. As we have no set of pre-defined polynomials and they are derived at runtime, every possible combination of boolean functions over the worker cubes must be realized. Even with tight restrictions, i. e., a maximum number of terms per monomial and monomials per polynomial, it is impossible to provide such a reconfigurable structure in hardware. As a consequence, a fully dynamic approach leads to extremely large multiplexers and thus to very high area consumption on the FPGA, which is prohibitively slow. Therefore, we need to choose a different strategy to implement this part in hardware.

Worksum update: In order to finish one iteration, the worksum is modified. To simplify the synchronization between the different threads, each worker updates a local intermediate value. In order to generate $d + 1$ bit intermediate values from the $d - w$ bit sums, we precede the number not by a constant 1 but also with

the w bit number of the worker thread. Please note that the actual implementation, we do not use $d+1$ bit XOR operations: If the number of XORs is even, we need to prefix the constant, otherwise, we need to prefix zeroes. Thus, a simple 1 bit value is sufficient to choose between these two values. When all workers are finished, the real result needs to be computed by a XOR operation over all results.

Overall, the complexity of the algebraic attack is too high for a single machine and a cluster of some kind is necessary. As the most cost-intensive operation concerns the 2^d computations of the 256 step Grain initialization, the use of a PC cluster is only marginally feasible, as CPUs are ill-suited for performing computations relying heavily on bit-permutations.

We thus decided to implement and experimentally verify the complex attack on dedicated reconfigurable hardware using the RIVYERA special-purpose hardware cluster, as described in Section 2.4. For the following design decisions, we remark that RIVYERA provides 128 powerful Spartan-3 FPGAs, which are tightly connected to an integrated server system powered by an Intel Core i7 920 with 8 logical CPU cores. This allows us to utilize dedicated hardware and use a multi-core architecture for the software part.

3.2 FPGA Design

In this section, we give an overview of the hardware implementation. As the total number of iterations for one attack (for the correct guess of the 39 secret expression values) is 2^d, the number of workers for an optimal setup has to be a power of two to decrease control logic and communication overhead.

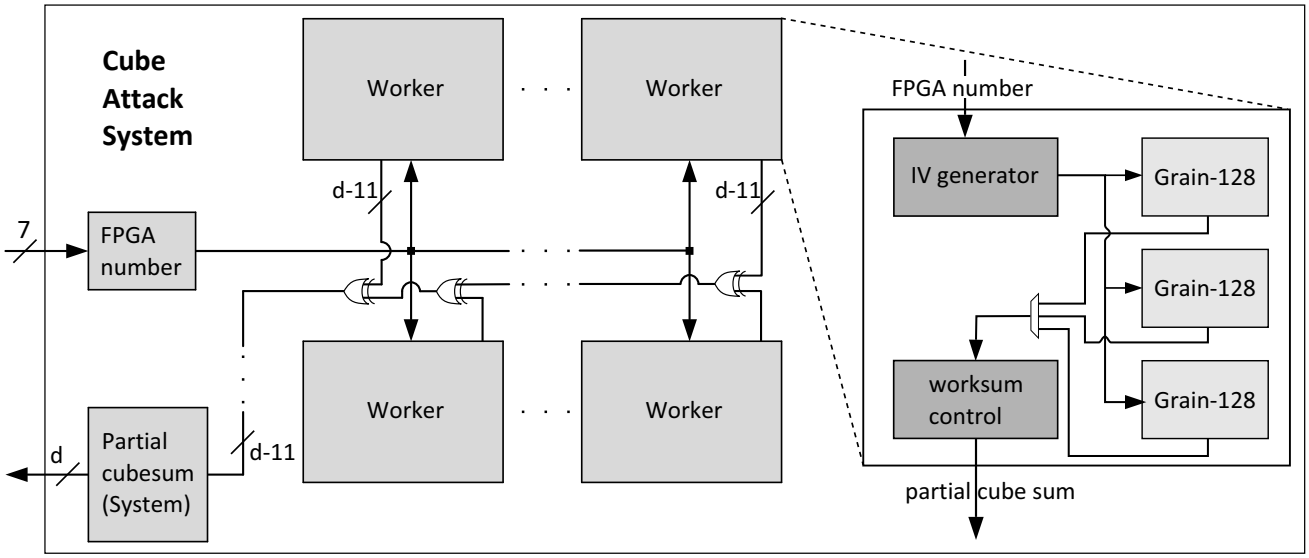


Fig. 4. FPGA Implementation of the online phase for cube dimension d.

Figure 4 shows the top level overview. Each of the 128 Spartan-3 5000 FPGAs features 2^4 independant workers and each of these workers consists of its own IV generator to control multiple Grain-128 instances.

The IV generator needs three clock cycles per IV and we need a corresponding number of Grain instances to process the output directly. As it is possible to run more than one initialization step per clock cycle in parallel, we had to find the most suitable time/area trade-off for the cipher implementation.

Table 2 shows the synthesis results of our Grain implementation. In comparison, Aumasson *et al.* also used 2^5 parallel steps, which is the maximum number of supported parallel steps without additional overhead, on the large Virtex-5 LX330 FPGA used in [14]. By using three Grain instances, we do not lose clock cycles where IV generation or cipher computation idle and - analyzing the critical path of the full design - the Grain module is not the limiting factor.

Table 2. Synthesis results of Grain-128 implementation on the Spartan-3 5000 FPGA with different numbers of parallel steps per clock cycle.

Parallel Steps	2^0	2^1	2^2	2^3	2^4	2^5
Clock Cycles (Init)	256	128	64	32	16	8
Max. Frequency (MHz)	227.169	226.706	236.016	234.357	178.444	159.210
Max. Frequency (MHz)	227	226	236	234	178	159
FPGA Resources (Slices)	165	170	197	239	311	418
Flip Flops	278	285	310	339	393	371
4 input LUTs	288	297	345	420	583	804

The results of the cipher instances are gathered in the worksum control, which updates the partial cubesum per worker, which is the output of all worker instances. The FPGA computes a partial cubesum of all workers on the FPGA and returns it upon request.

As mentioned before, it is not possible to create an unrestricted IV generation. To circumvent this problem, we locally fix certain values per key. This enables us to reduce the complexity of the system, as dynamic inputs are changed to constants. The drawback is that we need to generate a bitstream, which is dependent on the parameter set and - more important - on the key we wish to attack.

By looking at the discussion on the dynamic input to the IV generation, we can see that by fixing the parameter set, we already gain an advantage on the iteration over the cube itself: By sorting these positions and a smart distribution among the FPGAs, we reduce the complexity of the first part of the IV generation. By setting the base IV constant, we can optimize the design automatically and with the constant key, we remove the need to transfer any data to the FPGAs after programming them.

Nevertheless, the most important unknown are the polynomials. While we do have some restrictions from the way these polynomials (consisting of `and` and `xor` operations) are generated, we cannot forecast the impact of them: Remember that we use 13 different boolean functions in this parameter set. Each of these can have up to 50 monomials, where every monomial can - in theory - use all d positions of the cube. Luckily, on average, most polynomials depend on less than 5 variables.

3.3 Software Design

Now that we described the FPGA design and the need of key-dependent configurations, we will go into detail on the software side of the attack. In order to successfully implement and run the attack on the RIVYERA cluster and benefit from its massive computing power, we propose the following implementation. Figure 5 shows the design of the modified attack.

The software design is split into two parts: We use all but one core of the i7 CPU to generate attack specific bitstreams, i. e., configuration files for the FPGAs, in parallel to prepare the computation on the FPGA cluster. Each of these generated designs configures the RIVYERA for a complete attack on one random key. As soon as one bitstream was generated and waits in the queue, the remaining core programs all 128 FPGAs with it, starts the attack, waits for the computation to finish and fetches the results. With the partial cubesums per FPGA, the software computes the final result and evaluates the attack on the chosen key to measure the effectiveness of the attack.

In contrast to the first approach, which uses the generic structure realizable in software and needed a lot of communication, we generate custom VHDL code containing constant settings and fixed boolean functions of the polynomials derived from the parameter set and the provided key. Building specific configuration files for each attack setup allows us to implement as many fully functional, independent, parallel workers as possible without the area consumption of complex control structures. In addition, this allows us to strip down the communication interface and data paths to a minimum: only a single 7 bit parameter, distributing the workspace between all 128 FPGAs, is necessary at runtime to start the computation and receive a d bit return value. This efficiently circumvents all of the problems and overhead of a generic hardware design at the cost of rerunning the FPGA design flow for each parameter/key pair.

Please note that in this approach the host software modifies a basic design by hard-coding conditions and adjusting internal bus and memory sizes for each attack. We optimized this basic layout as much as

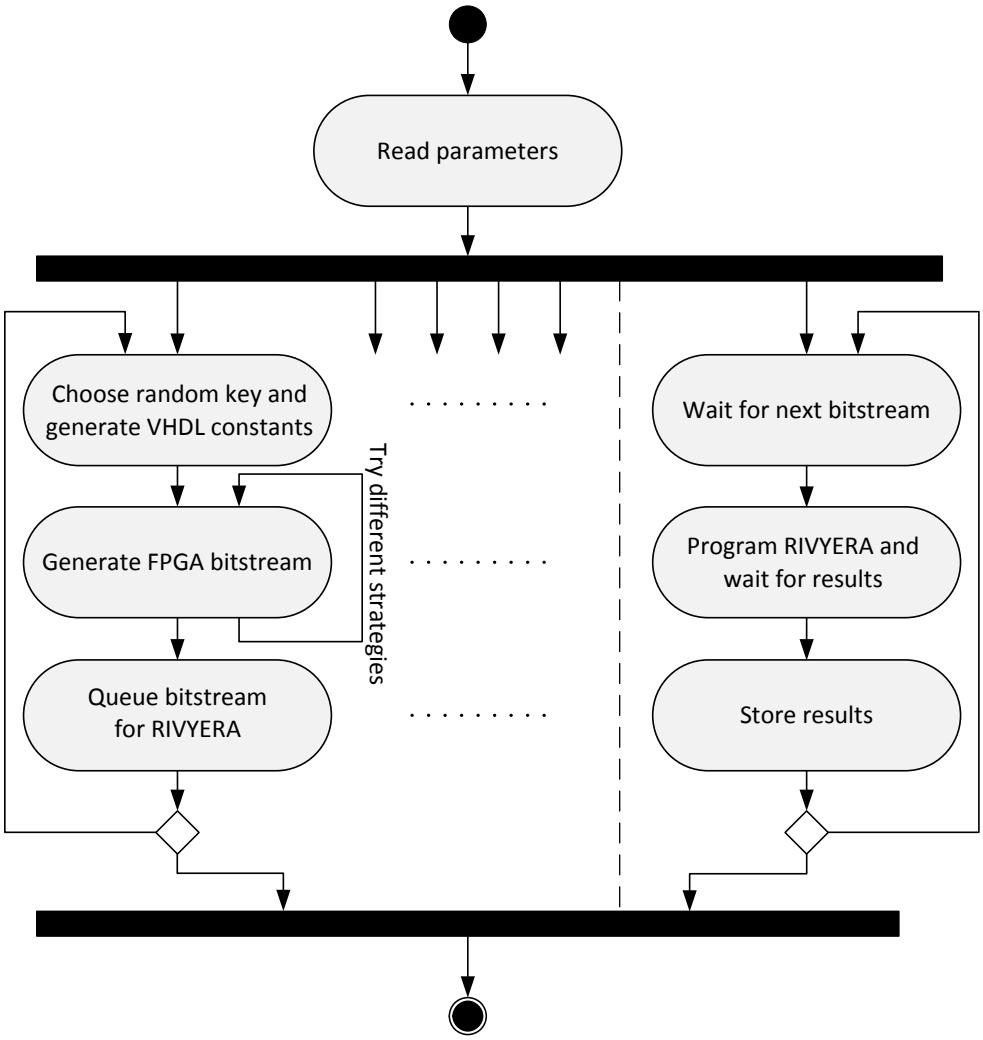


Fig. 5. Cube Attack Implementation on Special-Purpose Hardware

possible for average sized boolean functions, but the different choices of the polynomial functions lead to different combinatorial logic paths and routing decisions, which is bound to change the critical path in the hardware design. As the clock frequency is directly linked to the critical path, we implemented different design strategies as well as multiple fall-back options to modify the clock frequency constraints in order to prevent parameter/key pairs from resulting in an invalid hardware configurations. As a result, you can see a fallback path in Figure 2, which tries different design strategies automatically if the generated reports indicate failures during the process or timing violations after the analysis phase.

4 Results

In this section, we present the results of our implementation. The hardware results are based on Xilinx ISE Foundation 13 for synthesis and place and route. We compiled the software part using GCC 4.1.2 and the OpenMP library for multi-threading and ran the tests on the i7 CPU integrated in the RIVYERA cluster.

The hardware design was used to test different parameter sets and chose the most promising parameters. The resulting attack system for the online phase — consisting of the software and the RIVYERA cluster — uses 16 workers per FPGA and 128 FPGAs on the cluster in parallel. This means that the number of Grain computations per worker is reduced to 2^{d-11}, i. e., 2^{39} with the current cube dimension. The design ensures

that each key can be attacked at the highest possible clock frequency, while it tries to keep the building time per configuration moderate.

Table 3. Strategy Overview for the automated build process. The strategies are sorted from top to bottom. In the worst case, all 16 combinations may be executed.

Global Settings	Worker Clock (MHz)			
2.4× Input Clk	120			
2.2× Input Clk	110			
2.0× Input Clk	100			
RIVYERA Input Clk	50			
Map Settings	Placer Effort	Placer Extra Effort	Register Duplication	Cover Mode
Speed	Normal	None	On	Speed
Area	High	Normal	Off	Area
Place and Route Settings	Overall Effort	Placer Effort	Router Effort	Extra Effort
Fast Build	High	Normal	Normal	None
High Effort	High	High	High	Normal

Table 3 explains the different strategies in more detail: Each line represents one choice of settings, while the three blocks represent the impact on the subsequent build process. The design is synthesized with one of the four clock frequency settings. When the build process reaches the mapping stage, it tries first the speed optimized settings and runs the fast place and route. In case this fails, it tries the high effort place and route setting. If this fails, it tries the Area settings for the mapping and may fall back to a lower clock frequency setting, repeating the complete build process again.

As the user clock frequency of the RIVYERA architecture is 50 MHz, the Xilinx Tools will easily analyze the paths for a scaling factor 1.0 and 2.0. As the success rate when routing the design with 2.5 times the input clock frequency (125 MHz) was too low, we removed this setting due to the high impact on the building time.

Table 4. Results of the generation process for cubes of dimension 46, 47 and 50. The duration is the time required for the RIVYERA cluster to complete the online phase. The Percentage row gives the percentage of configurations built with the given clock frequency out of the total number of configurations built with cubes of the same dimension.

Cube Dimension d	46			47	50	
Clock Frequency (MHz)	100	110	120	120	110	120
Configurations Built	1	7	8	6	60	93
Percentage	6.25	43.75	50	100	39.2	60.8
Online Phase Duration	17.2 min	15.6 min	14.3 min	28.6 min	4h 10 min	3h 49 min

Table 4 reflects the results of the generation process and the distribution of the configurations with respect to the different clock frequencies. It shows that the impact of the unknown parameters is not predictable and that fallback strategies are necessary. Please note that the new attack tries to generate configurations for multiple keys in parallel. This process — if several strategies are tried — may require more than 6 hours before the first configuration becomes available. Smaller cube dimensions, i. e., all cube dimensions lower than 48, result in very fast attacks and should be neglected, as the building time will exceed the duration of the attack in hardware. Further note that the duration of the attack increases exponentially in d, e. g., assuming 100 MHz as achievable for larger cube dimensions, $d = 53$ needs 1.5 days and $d = 54$ needs 3 days.

5 Conclusions

Cube attacks and testers are notoriously difficult to analyze mathematically. To test the attack experimentally, to find and validate suitable parameters and to verify its complexity, we were restricted by the limitations of

CPU clusters, making further evaluations difficult. Due to its high complexity and hardware-oriented nature, the attack was developed and verified using the RIVYERA hardware cluster.

We presented an architecture making use of both the integrated i7 CPU and the 128 FPGAs of the RIVYERA cluster and heavily relying on the reconfigurability of the cluster system. This way, we were able to successfully implement the first attack on Grain-128, which is considerably faster than exhaustive search and, unlike previous attacks, makes no assumptions on the secret key.

While we were unable to conduct the full attack in this work, we can estimate its results by running a partial version. Our experimental results showed that for about 7.5% of the keys we could achieve a signficant improvement by a factor of 2^{38} over exhaustive search.

References

1. M. Hell, T. Johansson, A. Maximov, and W. Meier, "A Stream Cipher Proposal: Grain-128," in *Information Theory, 2006 IEEE International Symposium on*, july 2006, pp. 1614 –1618.
2. I. Dinur and A. Shamir, "Breaking Grain-128 with Dynamic Cube Attacks," in *FSE*, ser. Lecture Notes in Computer Science, A. Joux, Ed., vol. 6733. Springer, 2011, pp. 167–187.
3. S. Budiansky, *Battle of wits: the complete story of codebreaking in World War II*. Free Press, 2000.
4. M. Matsui, "The First Experimental Cryptanalysis of the Data Encryption Standard," in *CRYPTO*, ser. Lecture Notes in Computer Science, Y. Desmedt, Ed., vol. 839. Springer, 1994, pp. 1–11.
5. E. F. Foundation, *Cracking DES: Secrets of Encryption Research, Wiretap Politics and Chip Design*, M. Loukides and J. Gilmore, Eds. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1998.
6. T. Güneysu, T. Kasper, M. Novotny, and C. Paar, "Cryptanalysis with COPACOBANA," *IEEE TRANSACTIONS ON COMPUTERS*, vol. 57, no. 11, pp. 1498–1513, 2008.
7. J.-P. Aumasson, I. Dinur, W. Meier, and A. Shamir, "Cube Testers and Key Recovery Attacks on Reduced-Round MD6 and Trivium," in *FSE*, ser. Lecture Notes in Computer Science, O. Dunkelman, Ed., vol. 5665. Springer, 2009, pp. 1–22.
8. X. Lai, "Higher Order Derivatives and Differential Cryptanalysis," *Symposium on Communication, Coding and Cryptography*, pp. 227–233, 1994, Higher Order Derivatives and Differential Cryptanalysis. In *"Symposium on Communication, Coding and Cryptography", in honor of James L. Massey on the occasion of his 60th birthday*, pages1994.
9. M. Vielhaber, "Breaking ONE.FIVIUM by AIDA an Algebraic IV Differential Attack," *IACR Cryptology ePrint Archive*, vol. 2007, p. 413, 2007.
10. I. Dinur and A. Shamir, "Cube Attacks on Tweakable Black Box Polynomials," in *EUROCRYPT*, ser. Lecture Notes in Computer Science, A. Joux, Ed., vol. 5479. Springer, 2009, pp. 278–299.
11. A. Joux, *Algorithmic Cryptanalysis*. Chapman & Hall/CRC, 2009.
12. I. Dinur, T. Güneysu, C. Paar, A. Shamir, and R. Zimmermann, "An Experimentally Verified Attack on Full Grain-128 Using Dedicated Reconfigurable Hardware," in *ASIACRYPT*, ser. Lecture Notes in Computer Science, D. H. Lee and X. Wang, Eds., vol. 7073. Springer, 2011, pp. 327–343.
13. T. Güneysu, G. Pfeiffer, C. Paar, and M. Schimmler, "Three Years of Evolution: Cryptanalysis with COPACOBANA Special-Purpose Hardware for Attacking Cryptographic Systems," *Workshop on Special-purpose Hardware for Attacking Cryptographic Systems – SHARCS*, 2009, "uneysu, Gerd Pfeiffer, Christof Paar, and Manfred Schimmler. Three Years of Evolution: Cryptanalysis with COPACOBANA. In *Workshop on Special-purpose Hardware for Attacking Cryptographic Systems – SHARCS 2009*, September2009.
14. J.-P. Aumasson, I. Dinur, L. Henzen, W. Meier, and A. Shamir, "Efficient FPGA Implementations of High-Dimensional Cube Testers on the Stream Cipher Grain-128," *Workshop on Special-purpose Hardware for Attacking Cryptographic Systems – SHARCS*, September 2009.

A Appendix: Simulation Algorithm

Algorithm 2 The Dynamic Cube Attack Simulation

Input: 128-bit key K.
Input: Expressions $e_1, ..., e_{13}$ and the corresponding indexes of the dynamic variable $i_1, ..., i_{13}$.
Input: Big cube $C = (c_1, ..., c_{50})$ containing the indexes of the 50 cube variables.
Output: The score of K.

$S \leftarrow (0, ..., 0)$ ▷ the 51 cube boolean sums, where $S[51]$ is the sum of the big cube
$IV \leftarrow (0, ..., 0)$ ▷ as the initial 96-bit IV
for $j \leftarrow 1$ **to** 13 **do**
    $e_j \leftarrow eval(e_j, K)$ ▷ Plug the value of the secret key into the expression
end for
for all cube indexes CV **from** 0 to 2^{50} **do**
    for $j \leftarrow 1$ **to** 50 **do**
        $IV[c_j] \leftarrow CV[j]$ ▷ Update IV with the value of the cube variable
    end for
    for $j \leftarrow 1$ **to** 13 **do**
        $IV[i_j] \leftarrow eval(e_j, IV)$ ▷ Update IV with the evaluation of the dynamic variable
    end for
    $b \leftarrow$ Grain-128(IV, K) ▷ Calculate the first output bit of Grain-128
    for $j \leftarrow 1$ **to** 50 **do**
        if $CV[j] = 0$ **then**
            $S[j] \leftarrow S[j] + b \pmod 2$ ▷ Update cube sum
        end if
    end for
    $S[51] \leftarrow S[51] + b \pmod 2$
end for
$HW \leftarrow 0$
for $j \leftarrow 1$ **to** 51 **do**
    if $S[j] = 0$ **then**
        $HW \leftarrow HW + 1$.
    end if
end for
return $HW/51$

Usable assembly language for GPUs: a success story

Daniel J. Bernstein[*], Hsieh-Chung Chen[†], Chen-Mou Cheng[‡], Tanja Lange[§], Ruben Niederhagen[¶§], Peter Schwabe[‖¶], Bo-Yin Yang[¶]

[*] Department of Computer Science
University of Illinois at Chicago
851 S. Morgan Street, Chicago, IL 60607-7053, USA
Email: djb@cr.yp.to

[†] Harvard University
Email: kc@crypto.tw

[‡] Department of Electrical Engineering
National Taiwan University
1, Section 4, Roosevelt Road, Taipei 10617, Taiwan
Email: doug@crypto.tw

[§] Department of Mathematics and Computer Science
Eindhoven University of Technology
Den Dolech 2, 5600 MB Eindhoven, Netherlands
Email: tanja@hyperelliptic.org

[¶] Institute of Information Science
Academia Sinica
No 128, Academia Road, Section 2, Nankang, Taipei 11529, Taiwan
Email: by@crypto.tw, ruben@polycephaly.org

[‖] Research Center for Information Technology Innovation
Academia Sinica
No 128, Academia Road, Section 2, Nankang, Taipei 11529, Taiwan
Email: peter@cryptojedi.org

***Abstract*—The NVIDIA compilers `nvcc` and `ptxas` leave the programmer with only very limited control over register allocation, register spills, instruction selection, and instruction scheduling. In theory a programmer can gain control by writing an entire kernel in van der Laan's `cudasm` assembly language, but this requires tedious, error-prone tracking of register assignments.**

This paper introduces a higher-level assembly language, `qhasm-cudasm`, that allows much faster programming while providing the same amount of control over the GPU. This language has been used successfully to build a 90000-machine-instruction kernel for a computation described in detail in the paper, the largest public cryptanalytic project in history. The best GTX 295 speed that has been obtained for this computation with `nvcc` and `ptxas` is 25 million iterations per second; the best GTX 295 speed that has been obtained with `qhasm-cudasm` is 63 million iterations per second.

I. Introduction

Decades of advances in the design of optimizing compilers have *reduced*, but have not *eliminated*, the need for some performance-critical functions to be written in assembly language. For high-performance implementations in C it is common practice to write the critical parts in assembly.

This paper introduces a new assembly language `qhasm-cudasm` for programming Graphics Processing Units (GPUs), specifically NVIDIA's Tesla-architecture GPUs. There is no analogue to assembly in CUDA, NVIDIA's standard GPU programming toolkit. NVIDIA's "PTX" is labelled as an assembly language, but it is in fact compiled and hides too many machine details to be usable for writing cutting-edge code. We started from van der Laan's `cudasm`, a true assembly language for Tesla GPUs; we completely redesigned the syntax for readability and incorporated a powerful register allocator from `qhasm` [3], ultimately obtaining a *usable* assembly language for GPUs.

We have used `qhasm-cudasm` successfully to produce highly optimized code for a major cryptanalytic computation, the "ECC2K-130" computation, an order of magnitude larger than the RSA-768 factorization announced in [11]. This paper describes this implementation in detail as a case study for `qhasm-cudasm`. Finishing the computation in two years with the fastest CPU software developed would require 1595 quad-

core (3.2 GHz AMD Phenom II X4 955) PCs, but would require just 534 GTX 295 graphics cards with our software, or just 229 cost-optimized PCs each having a quad-core CPU and two GTX 295 graphics cards.

Our software was optimized for and tested on a GTX 295 graphics card containing two 1.242GHz G200b GPUs. We later tried the same software on the T10 GPUs (in Tesla S1070-500 units) in TeraGrid's Lincoln cluster, similar GPUs in the NCF/SARA cluster, and the FX 5800 GPUs in TeraGrid's Longhorn cluster; documentation indicates that these GPUs have essentially the same microarchitecture as the GTX 295, except for small differences in clock speed, and as expected we saw very similar cycle counts. Of course, assembly languages need to be redone for each new architecture, but massive speedups are ample justification for this effort.

A. Cores and multiprocessors

Each of the targeted GPUs contains 30 "multiprocessors"; each multiprocessor contains 8 "streaming processors" and a few auxiliary units; each "streaming processor" performs, typically, one 32-bit operation per cycle. **Warning:** NVIDIA documentation often refers to each "streaming processor" as a "core." We find that terminology misleading: GPU multiprocessors are much more closely analogous to CPU cores, and GPU streaming processors are much more closely analogous to CPU ALUs. **In this paper we refer to multiprocessors as cores and streaming processors as ALUs.**

This paper describes a computation running on one core of a GPU. Cycle counts in the paper are cycle counts for a single GTX 295 core. Of course, to make full use of the GPU we actually run 30 independent computations on the 30 cores of the GPU, i.e., 60 independent computations on the GTX 295.

B. Threads

There is an additional level of parallelism within each GPU core: the core runs many threads of computation simultaneously. The number of threads is chosen by the programmer, but is limited by various resources shared among threads.

There are many cycles of latency between successive operations in a thread, and standard advice is to run at least 192 threads per core to hide this latency. For several reasons we chose to run only 128 threads per core in the particular computation discussed in this paper. The penalty for using only 128 threads turns out to be quite small: experiments show that 128 threads performing arithmetic operations in registers can keep all of the ALUs busy in a GPU core, and that occasional loads and stores from "shared memory" can be handled by the "special function units" in the GPU core without slowing down the ALUs. One important reason to limit the number of threads, although not the only reason, is that all threads on a core have to share a total of 16384 registers; 128 threads are each given 128 registers (minus a few special registers), and we make good use of those registers. A smaller number of threads would linearly reduce the ALU utilization and would not provide more registers per thread: the hardware does not allow one thread to address more than 128 registers.

C. Bitslicing

The particular computation targeted in this paper consists of a series of iterations summarized in Section II. Each iteration is a straight-line sequence of approximately 70000 bit operations. The bit operations are two-input ANDs (the output is 1 if both inputs are 1, otherwise 0) and two-input XORs (the output is 1 if the inputs are different, otherwise 0).

We carry out 128 independent iterations in parallel on a single GPU core, with no communication between the iterations. We apply each bit operation as a 128-way SIMD operation. GPU instructions, like CPU instructions, are word-oriented, so we adopt a standard strategy called "bitslicing": the 32 consecutive bits in a 32-bit register, or in a 32-bit memory location, are from 32 separate iterations. A 128-way XOR is then built straightforwardly from four 32-bit XORs, and a 128-way AND is built straightforwardly from four 32-bit ANDs.

We emphasize that these are 128 *bit* operations. For comparison, a single 32-bit instruction followed by 128 threads in parallel is carrying out 4096 bit operations. In other words, 128-way parallelism is only 1/32nd of the parallelism required to keep 128 threads busy. We would like to carry out many more independent iterations in parallel, but we are limited by the amount of shared memory available on a GPU core, especially for multiplication; see Section IX.

II. Target: the ECC2K-130 challenge

Many cryptographic implementations base their security on the hardness of the Discrete Logarithm Problem (DLP) on elliptic curves. Cryptanalysis, the science of breaking cryptography, plays an important role in determining the hardness of the DLP in relation to the different ways of choosing these curves—critical information for users who want to choose curves that balance efficiency and security.

In general, the hardness of the DLP grows with the size of the curve and there are several types of curves that are particularly hard to break. To give good estimates for the cost of breaking the DLP, challenge problems are posed at various sizes and for the different types of curves. From the time necessary to break these challenges one can extrapolate the costs for breaking larger systems and estimate how much computational effort is necessary to break a real-world system; this then translates into how much money is necessary to buy the hardware necessary to break actual cryptographic implementations. Of course the cost estimates depend on the type of hardware—customized ASICs are likely to have a better price-performance ratio than off-the-shelf PCs but also require more specialized knowledge and come at a high initial cost for designing and fabricating the chip. Challenges put no restrictions on the hardware used but so far all breaks of challenges in public-key cryptography have used standard CPUs with ample RAM per thread, such as the Intel Core 2 or the PlayStation 3.

The computation described in this paper is part of a big cryptanalytic project, namely the breaking of the Certicom challenge ECC2K-130 [7]. The ECC2K-130 challenge was

posed in 1997 and remains unbroken today. Our estimates are that about 2^{77} bit operations are necessary to break this challenge, which just barely brings this computation into reach for academic teams. Full details about the challenge and the mathematical background are described in [1]. That paper also gives an overview of how the core part of the ECC2K-130 computation can be implemented on various platforms, ranging from standard CPUs to FPGAs and ASICs. The paper contains a section on GPUs but achieves worse performance on an entire GTX 295 than on a PlayStation 3, even though the GTX 295 is theoretically capable of performing almost eight times as many bit operations per second (10× as many cores, 2× as many bit operations per cycle per core, 0.388× the clock speed). More details on the fast implementation that we use here to illustrate the features of `qhasm-cudasm` are given in [4].

The computation is embarrassingly parallel and does not require much communication or storage. The challenge for the implementor is to perform as many meaningful bit operations per cycle as possible. Unfortunately, as described in the next section, the available compiler tools were not able to schedule the instructions and memory accesses in a way suitable to keep the arithmetic units of the GPU busy.

Our ECC2K-130 computation works with two representations of the finite field $\mathbf{F}_{2^{131}}$: a polynomial-basis representation introduced in [5], and a standard normal-basis representation. Multiplications naturally begin with polynomial-basis inputs but can efficiently produce outputs in either polynomial basis or normal basis. Squarings are most efficient in normal-basis representation.

The computation consists of a large number of iterations, with an essentially unlimited number of iterations to perform in parallel. We repeat here the description of the iteration function from [5, Section 5]. The input to an iteration is a pair (x,y) of elements of $\mathbf{F}_{2^{131}}$ satisfying certain conditions, notably that x has even Hamming weight w in normal basis. The output is another pair (x',y') defined by

$$x' = \lambda^2 + \lambda + x + x^{2^j},$$
$$y' = \lambda(x + x') + x' + y,$$

where $j = 3 + ((w/2) \bmod 8)$ and $\lambda = (y + y^{2^j})/(x + x^{2^j})$.

Each iteration involves three multiplications to compute the reciprocal of $x + x^{2^j}$, two further multiplications, four conversions from normal basis to polynomial basis, and many squarings and additions. See [5] for further details and analysis of the number of bit operations per iteration. Major subroutines in our software include `ppn`/`ppp` for a multiplication producing normal-basis/polynomial-basis output respectively, `multprep` for a basis conversion, `hamming` for a Hamming-weight computation, `add` for an addition, and `sq` for a squaring. These operations are described in later sections when we explain details of their implementation in the new framework of our usable assembly language.

III. Trouble with the CUDA toolchain

The hardware architecture of GPUs is expected to change even faster than the architecture of CPUs, not only in respect to the number of cores and sizes of caches but also in respect to the instruction set and register file organization. However, NVIDIA has committed to keep CUDA programs forward-compatible, ensuring that code written today runs on new video cards of tomorrow. NVIDIA covers a wide range of GPUs over several generations by using layers of abstraction.

We briefly recap the normal compilation process supported by NVIDIA. The programmer writes software in a C-like language, either CUDA or OpenCL. The NVIDIA compiler `nvcc` compiles this software into instructions for a pseudo-machine called PTX ("parallel thread execution"). The second-stage NVIDIA compiler `ptxas` (normally invoked implicitly by `nvcc`) reads PTX code and produces actual GPU machine language in an undocumented format called `.cubin`. The CUDA driver loads the `.cubin` file onto a GPU and runs it. The `.cubin` format was text before CUDA 3.0, but was then replaced by a binary format requiring less parsing.

Writing software in CUDA or OpenCL, or in the intermediate language PTX, allows the software to be easily adapted to new hardware generations: in principle one should be able to simply recompile the software for a new GPU. However, this convenience does not seem to be compatible with our goal of high performance. Most of the GPU clusters that we have access to have the same Tesla architecture, increasing the value of optimizations targeting that architecture.

Since the PTX instruction definition is not very far from the actual instruction set of the Tesla-architecture GPUs, one could reasonably hope that using register variables in PTX followed by assembling to binary by `ptxas` from the NVIDIA toolchain would give the necessary control over the resources of a GPU. Unfortunately, we encountered major drawbacks of `ptxas`, including bad run-time performance, excessive compilation times, and sometimes complete failures to produce binary code.

A critical problem: Apparently the register allocator of `ptxas` was not designed for large kernels; the intended target applications are graphics shaders and small computing kernels. When the kernel is large, the allocation "leaks" (allocates too many) registers even when we explicitly generate code to be runnable within a smaller number of registers. Execution is often drastically slowed down when the compiler spills values to "local memory"; spilling values to "shared memory" is often even worse, preventing us from launching as many threads as we would like.

It is possible to code around the deficiencies of `ptxas`. However, the cost is significant in run time *and* programmer time. In our initial implementation work, we were forced after several months and much pulling of hair to employ a simple "schoolbook" method of implementing our critical multiplication subroutine, instead of any of the more advanced methods such as those mentioned in [1]. The end result was an overall performance hit of around 50%.

In many other cases the only workaround is to split the computation into several kernels which are executed one after another. This introduces additional overhead for kernel invocation. An ideal program would need to have as large kernels as possible to be handled by ptxas to avoid as much invocation overhead as possible. This would require much effort and experimentation on ideal kernel sizes.

We take a different approach, replacing ptxas by a tool that is able to handle large kernels without penalty. The next section introduces our new qhasm-cudasm tool. Beyond the basic benefit of supporting large kernels, qhasm-cudasm introduces a new input language that gives the programmer much more control over the hardware than PTX does, while at the same time achieving higher readability.

IV. cudasm, qhasm, AND qhasm-cudasm

This section describes the existing cudasm and qhasm assembly-language tools, and our new qhasm-cudasm assembly language.

A. cudasm

In 2007 van der Laan reverse-engineered the machine language of the NVIDIA GPUs. He released a decuda disassembler, translating each machine instruction into a readable format (somewhat similar to NVIDIA's documented PTX format, although the machine language turned out to be more complicated than PTX). Shortly afterwards he released a cudasm assembler. See [14].

Our experience is that decuda is by far the easiest way to figure out what ptxas is doing wrong. Anecdotal evidence suggests that decuda became moderately popular among serious programmers for exactly this reason. However, cudasm attracted far less attention. One paper [8] reported a successful application of decuda and cudasm to manually rewrite a small section of ptxas output, but said that this was "tedious" and hampered by cudasm bugs: "we must extract minimum region of binary code needed to be modified and keep remaining binary code unchanged ... implementation of cudasm is not entirely complete, it is not a good idea to write whole assembly manually and rely on cudasm."

We fixed various bugs in cudasm: for example, we found that memory offsets were sometimes silently ignored. Our fixed version of cudasm is capable of generating a fully functional 90000-GPU-instruction kernel for our software, and in fact is exactly what we use to generate all of the kernels that we are now running. However, *writing* these kernels in the cudasm input language would have been an extremely time-consuming job; we actually wrote our software in a new language, as discussed below.

B. qhasm

Years ago, one of the authors of this paper (Bernstein) tried very hard to convince gcc to emit his desired sequence of floating-point instructions for a performance-critical cryptographic application on an x86 CPU. Unfortunately, the x86 architecture had (at the time) only 8 floating-point registers; gcc expected to keep one of those registers in reserve for stack management; and trying to fit this particular application into 7 registers, rather than 8, seemed to irreparably compromise performance. He resorted to writing the same function in x86 assembly language; this required manually maintaining a chart of the floating-point values in each register, and redoing the chart several times to accommodate changes in the code.

The same author subsequently developed a higher-level assembly language, qhasm, to give him the same control as a traditional x86 assembly language with far less programming time. Many cryptographic speed records were set by software written in qhasm; see, e.g., [2], [6], and [9].

There are several differences between qhasm and a traditional assembly language. The most visible difference is the choice of syntax: qhasm uses readable C-like syntax such as `d = c + 3`, while a traditional assembler might use `lea 3(%ecx),%edx`. The most important difference, however, is that qhasm includes a very fast, very smart register allocator, mapping an unlimited stream of programmer-selected names to the small number of registers provided by a CPU. The programmer still controls the selection of instructions, still controls the order of instructions, and still controls *which* values remain in registers, but qhasm handles the tedious task of assigning registers. The programmer *can* easily specify a register assignment but almost never has to.

C. qhasm-cudasm

To bring the same level of assembly-language usability to GPUs we have built a new qhasm-cudasm language, reusing the qhasm register allocator to generate code that can be fed as input to our fixed version of cudasm. We wrote our new ECC2K-130 software in qhasm-cudasm; see subsequent sections of this paper for details.

The core of qhasm-cudasm is a GPU machine-description file, currently 2818 lines; many of those lines are automatically generated from a shorter script. What follows is a typical line from that file, split into five lines here for readability:

```
r = y + t:
>r=low32:
<y=low32:
<t=low32:
asm/add.b32 >r, <y, <t:
```

This line says that if r and y and t have been declared to be low32 registers then the qhasm-cudasm instruction `r = y + t` reads y, reads t, writes r (eliminating any previous value), and corresponds to the cudasm instruction `add.b32 r, y, t`. Another line (with dots here for brevity) specifies the set of 64 low32 registers:

```
:name:low32:$r0:$r1:...:$r63:
```

The reader might be wondering how each thread can have access to nearly 128 registers (as mentioned earlier) if there are only 64 registers listed here. The answer is that there are 60 extra high32 registers:

```
:name:high32:$r64:$r65:...:$r123:
```

These are distinguished in `qhasm-cudasm` because they are distinguished in the GPU machine language: typical instructions can use `high32` in the first operand, or in the second operand, but not in the third operand.

We developed the machine-description file at the same time as writing code for the ECC2K-130 computation. We often changed syntax to improve readability or to avoid common error patterns. We added new instructions whenever we needed them:

```
r = s[p+m] if e signed<:
<e=cond:
inplace>r=high32:
<p=offset:
<r=high32:
#m:
asm/@<e.lt mov.b32 <r, s[<p+#m]:
```

This example is a predicated 32-bit load from shared memory into a `high32` register. The output value `r` can depend on the input value `r`, and the user can rely on it being assigned to the same register; this is what `inplace>r=high32` accomplishes. `#m` indicates an immediate constant. The register changes to the contents of the load if the `signed<` bit is set in the `e` predicate register.

We added an extra layer of preprocessing in front of `qhasm-cudasm`, using Ward's `m5` macro preprocessor [15]. An `m5` script is, except for some syntactic sugar, an `awk` program that prints another program; each of our `m5` scripts is a program that prints a `qhasm-cudasm` program.

We are using `qhasm-cudasm` for new applications, and are continuing to add instructions to the machine-description file. We are also building scripts to automate larger portions of the generation of the machine-description file.

D. Engineering a new implementation

For a typical function in the ECC2K-130 computation, such as `add` (described in the next section), we wrote a series of three implementations. The first implementation consisted of

- a simple C++ implementation `add.cpp` of a `paralleladd` function operating on data in CPU memory; and
- a test driver `addtest.cpp` calling `paralleladd`.

The second implementation consisted of

- CUDA code for `sharedadd` in `add.cu`, operating on data in shared memory;
- CUDA code for `kerneladd` in `kadd.cu`, operating on data in global memory by copying from global memory to shared memory and calling `sharedadd`;
- CUDA code for `paralleladd`, also in `kadd.cu`, operating on data in CPU memory by copying from the CPU to the GPU and then calling `kerneladd`; and
- the same test driver `addtest.cpp`.

The third implementation consisted of

- `qhasm-cudasm` code for `sharedadd` in `add.mq`;
- `qhasm-cudasm` code for `kerneladd` in `kadd.mq`, inlining `sharedadd` by including `add.mq`;
- the same CUDA code for `paralleladd` in `kadd.cu`, calling `kerneladd`; and
- the same test driver `addtest.cpp`.

Testing each version with the same test driver allowed typical bugs to be caught quickly.

It is common practice in GPU programming to implement functions twice, once in C and once in CUDA, with the same test driver; it is common practice in assembly-language programming to implement functions twice, once in C and once in assembly, with the same test driver. Our split between the first and second implementations followed the first practice, and our split between the second and third implementations imitated the second practice.

While working on the `qhasm-cudasm` versions we wrote a "big-kernel" implementation of the ECC2K-130 main loop. We wrote this main loop as a series of operations such as

```
xshift = r^2
xshift += r
d = global_Nd[j]
xshift += d
multprep r
```

in an ad-hoc language (not to be confused with `qhasm-cudasm`). We wrote a translator that converted this ad-hoc language to CUDA, replacing (e.g.) `xshift += r` with an appropriate call to the `sharedadd` function defined in `add.cu`, and replacing `d = global_Nd[j]` with an appropriate copy from global memory to shared memory. This CUDA implementation was extremely slow, and took an extremely long time for NVIDIA's tools to compile, but allowed the main loop to be tested independently of `qhasm-cudasm`.

We then wrote a translator that converted the same main loop to `qhasm-cudasm` code, automatically inlining individual `qhasm-cudasm` functions such as the `sharedadd` function defined in `add.mq`. Some data-flow misdeclarations had slipped past the individual tests and forced too many registers to be allocated in the main loop; in retrospect this could have been caught earlier by an extension to `qhasm-cudasm`, but in any case it was easy to diagnose and fix. This implementation took much less time to compile than the CUDA version and was much faster.

Afterwards we focused on optimizing various functions, producing the details described in subsequent sections. The third implementation described above was still in place, so any changes in `add.mq` were automatically tested by `addtest.cpp`. We also set up further scaffolding to measure time spent in various parts of the software, guiding our subsequent optimizations.

V. SCHEDULING INSTRUCTIONS: `add` AND `cadd`

The simplest arithmetic operation in the ECC2K-130 computation is an addition in a field of size 2^{131}: in other words, a XOR of two 131-bit input vectors, producing a 131-bit output vector. In our software this operation is called `add`. A "conditional" variant of the same operation, `cadd`, has an

extra input bit that masks the second input vector: whenever a mask bit is 0, `cadd` simply copies the corresponding input bit from the first input, ignoring the second input.

As discussed earlier, we actually perform 128 independent computations in parallel. The `add` function actually takes two 131×128-bit input matrices (each stored in bitsliced row-major form as 131 consecutive 128-bit vectors), and produces an output matrix of the same size. The `cadd` function actually takes an extra 128-bit vector that masks the second input matrix.

This section describes our implementations of `add` and `cadd`. These functions are only small parts of the ECC2K-130 computation, but they are a useful starting point to illustrate the capabilities of `qhasm-cudasm`.

A. Predicate registers

Our 128 threads handle a vector of $131 \times 128 = 524 \times 32$ bits as 128×32 bits, then 128×32 bits, then 128×32 bits, then 128×32 bits, and finally 12×32 bits.

For the last 12×32 output bits, only 12 of the 128 threads are active. Each thread compares its "thread ID" (between 0 and 127) to the constant 12, and temporarily deactivates itself unless the thread ID is smaller than 12.

The GPU hardware has four special "predicate registers" in each thread (in addition to other registers) to store the results of such comparisons. Our ECC2K-130 software sets up one of these registers as follows:

```
cond tid12
tid12 tid - $(const(12))
```

A typical GPU instruction allows one input from "constant memory"; `qhasm-cudasm` converts `$(const(12))` into a constant-memory location, and arranges for that location to contain the integer 12. The line `cond tid12` declares `tid12` as a predicate register; `tid12 tid - $(const(12))` compares `tid` to 12 and puts the result of the comparison into `tid12`.

To save time inside functions such as `add` we perform this particular comparison once in the caller, rather than performing it again each time the function is called. This means that one of the four predicate registers is reserved long-term for `tid12`, but `tid12` is used frequently enough to justify this.

B. Shared-memory offsets

Each GPU core has 16384 bytes of "shared memory." A typical GPU instruction allows one input from shared memory. Reading shared memory is *often* as fast as reading a register, although it can trigger some additional bottlenecks. Shared memory has two advantages over registers: first, it allows threads to quickly communicate with each other; second, shared-memory indices can be variables, while register numbers are always constant.

The GPU hardware has four special "offset registers" in each thread to store indices into shared memory. In our ECC2K-130 software the caller sets up an offset register before calling `add`, `cadd`, etc.:

```
offset tid4off
tid4off = tid << 2
```

Here `offset tid4off` declares `tid4off` to be one of the offset registers, and `tid4off = tid << 2` performs a shift by 2, i.e., a multiplication by 4. Note that a shift of a general-purpose register can put its output into an offset register.

C. How the `add` function works

Our `m5` function `add(to,from)` prints code that reads two inputs from shared memory, XORs the inputs, and writes the output on top of the first input. The input addresses are `to` and `from`; the output address is `to`. This code is not a machine-level function requiring `call` and `ret` instructions; it is inlined into the code generated by the caller, just like a macro expansion.

The `add` function begins by loading the first input:

```
syncthreads
new @x4
@x0 = s[tid4off + $to]
@x1 = s[tid4off + $(to + 512)]
@x2 = s[tid4off + $(to + 1024)]
@x3 = s[tid4off + $(to + 1536)]
@x4 = s[tid4off + $(to + 2048)] if tid12 signed<
```

Here `$` starts a compile-time computation; if `to` happens to be 4000, for example, then the fourth line above is converted into `@x1 = s[tid4off + 4512]`.

Recall that the `tid12` predicate register compared the thread ID to 12. The predicate `if tid12 signed<` skips the instruction on the same line unless the thread ID is smaller than 12. Note that each predicate register actually has four different comparison bits, allowing a variety of different predicates: `signed<`, `signed<=`, etc.

Two aspects of the above code help the `qhasm` register allocator manage data flow. First, `@` creates (at the `m5` level) a register name specific to this function call, avoiding accidental data flow between function calls. Second, `new @x4` informs the register allocator that there is no data flow from any previous use of the `@x4` register. This is not necessary for `@x0`, `@x1`, `@x2`, `@x3`: a register written by a *non-predicated* assignment has value independent of its previous value.

The `add` function continues by loading the second input:

```
new @y4
@y0 = s[tid4off + $from]
@y1 = s[tid4off + $(from + 512)]
@y2 = s[tid4off + $(from + 1024)]
@y3 = s[tid4off + $(from + 1536)]
@y4 = s[tid4off + $(from + 2048)] if tid12 signed<
```

The function then computes and stores the results:

```
@x0 ^= @y0
@x1 ^= @y1
@x2 ^= @y2
@x3 ^= @y3
@x4 ^= @y4
```

```
s[tid4off + $to] = @x0
s[tid4off + $(to + 512)] = @x1
s[tid4off + $(to + 1024)] = @x2
s[tid4off + $(to + 1536)] = @x3
s[tid4off + $(to + 2048)] = @x4 if tid12 signed<
```

The `cadd` function is similar but includes two extra instructions to load the mask and five extra AND instructions.

One can sometimes merge a load instruction and an XOR instruction into a single load-and-XOR instruction. However, the load-and-XOR instruction requires the immediate value used in the address computation to be in the range from -128 to 127, while the immediate values in the code shown above are almost never in this range. We subsequently experimented with rearranging the responsibilities of threads, assigning the first thread to the first 5×32 bits, the second thread to the next 5×32 bits, etc., so that each load address would be just 4 bytes after the previous rather than 512 bytes after the previous. This saved five instructions but cost four instructions to initialize two new offset registers; competition for offset registers meant that we could not keep these two offset registers longer than the function call. We plan to experiment further with eliminating this cost by more comprehensively rearranging our data structures, interleaving several 131×128-bit matrices with each other.

D. Performance

One can crudely model a GPU core as following 8 instructions per cycle. This model suggests that 128 threads would follow the 21 instructions shown above (10 loads, 5 stores, 5 XORs, 1 synchronization) in 336 cycles, and would follow the 28 `cadd` instructions in 448 cycles.

The GPU has a cycle counter. This counter is labelled as `halfclock` in `qhasm-cudasm`, because it actually counts once every two cycles. The cycle counter shows that the `cadd` instructions *plus* the cycle-counting time actually take 644 cycles, while our CUDA implementation of the same function takes 1106 cycles. An inspection of the machine-language code produced by `nvcc` and `ptxas` shows several sources of inefficiency, such as excessive use of offset registers. We could try to tweak our CUDA code, hoping for better output from `nvcc` and `ptxas`, but writing the code in `qhasm-cudasm` is less effort.

VI. Handling memory conflicts: `sq` and `csq`

The ECC2K-130 computation involves 8 different `sq` operations. Each of these operations takes as input one 131×128 bit matrix and applies a fixed permutation to the rows of the input matrix to produce again a 131×128 bit matrix as output. What is different in these 8 `sq` operations is the permutation applied to the rows. Each matrix row is stored in 4 successive 32-bit integers, so that one `sq` operation requires $4 \cdot 131 = 524$ load instructions and another 524 store instructions.

This `sq` operation takes 5 load instructions and 5 store instructions issued to 128 threads; here two instructions (one load and one store) are conditional depending on the thread ID. However, the performance of 128 concurrent load or store operations performed by 128 threads is highly dependent on the addresses of the data loaded or stored.

Shared memory is organized into 16 "banks" of memory: 16 consecutive 32-bit words are spread across the banks, one per bank. The 128 threads are organized into 4 "warps"; each warp is divided into 2 "half-warps". The 16 threads of one half-warp can execute a load operation of the `sq` operation in one cycle only if they all load from different memory banks. If two or more threads within the same half-warp load from or store to different addresses on the same memory bank in the same instruction, these requests are serialized.

We first group every four adjacent threads and let them operate on the four 32-bit words of one matrix row. Four such groups form a half-warp. We try to assign the 131 rows to such groups in a way that avoids all memory-bank conflicts.

We use a lookup table in constant memory to assign the 131 rows to 32 thread groups. Each entry of this lookup table requires only one byte; we pack 4 lookup-table entries into one 32-bit value.

The memory-bank restrictions also hold for storing the data, so avoiding memory-conflicts only for loading may yield bank conflicts when storing the data at the locations given by the fixed row permutation. We implemented a tool that uses a greedy approach to compute two lookup tables, one for loading and one for storing, that avoid almost all bank conflicts. These tables are linked through the permutation given by the `sq` operation.

Four of the `sq` operations are used very frequently, so we load the packed 20 load and 20 store positions from constant memory once at the beginning of the computation and then keep them in long-term registers:

```
sqseq_tmp = tid uint32>> 2
sqseq_pos = sqseq_tmp << 2

low32 sqseq_in0
low32 sqseq_in1
low32 sqseq_in2
low32 sqseq_in3
low32 sqseq_in4
sqseq_in0 = c0[sqseq_pos + 0]
sqseq_in1 = c0[sqseq_pos + 128]
sqseq_in2 = c0[sqseq_pos + 256]
sqseq_in3 = c0[sqseq_pos + 384]
sqseq_in4 = c0[sqseq_pos + 512]

low32 sqseq_out0
low32 sqseq_out1
low32 sqseq_out2
low32 sqseq_out3
low32 sqseq_out4
sqseq_out0 = c0[sqseq_pos + $(0+1048)]
sqseq_out1 = c0[sqseq_pos + $(128+1048)]
sqseq_out2 = c0[sqseq_pos + $(256+1048)]
sqseq_out3 = c0[sqseq_pos + $(384+1048)]
sqseq_out4 = c0[sqseq_pos + $(512+1048)]
```

Extracting one of the 4 packed positions from a 32-bit integer requires (at most) one shift and one mask instruction. For example, the following code extracts one position from `sqseq_out0`:

```
@p0 = sqseq_out0 uint32>> 4
@p0 &= $(const(4080))
```

Similar comments apply to `csq`, a conditional version of `sq`.

VII. Reducing serialization: `hamming`

The `hamming` operation takes as input a 131×128 bit matrix, computes the Hamming weight (sum of bits) for each column of the matrix, and stores the binary representation of the Hamming weight in the first bits of each column.

Adding up all bits *in order* is a completely serial process, but we obtain some parallelization by changing the order of additions. The basic operation takes three input vectors `a`, `b`, and `c` and computes two output vectors `bot` and `top` so that for each position i it holds that $\mathtt{a}[\mathtt{i}]+\mathtt{b}[\mathtt{i}]+\mathtt{c}[\mathtt{i}] = \mathtt{top}[\mathtt{i}] \cdot 2 + \mathtt{bot}[\mathtt{i}]$. It is possible to schedule 32 such operations in parallel under some conditions on the addresses modulo 16. This means that in the first step a total of $3 \times 32 = 96$ vectors can be handled, leading to 32 resulting vectors on level 1 (`top`) and $32 + (131 - 96) = 67$ results on level 0 (`bot`). In the next step results on level 2 can be computed by using vectors on level 1 as inputs; we carefully arrange vector positions so that in the same instruction results on level 1 and 0 are computed.

The computation is composed of two routines, FULLADDER(`x,in0,in1,in2,out0,out1,threads`) and HALFADDER(`x,in0,in1,out0,out1,threads`). The core of the FULLADDER routine consists of the following code:

```
syncthreads
@todo tid - $(const(4 * threads))
new @a
new @b
new @c
@a=s[tid4off + $(x + 16*in0)] if @todo signed<
@b=s[tid4off + $(x + 16*in1)] if @todo signed<
@c=s[tid4off + $(x + 16*in2)] if @todo signed<
@sum=@a ^ @b
@aandb=@a & @b
@candsum=@c & @sum
@bot=@sum ^ @c
@top=@aandb ^ @candsum
s[tid4off + $(x + 16*out0)]=@bot if @todo signed<
s[tid4off + $(x + 16*out1)]=@top if @todo signed<
```

The core of the HALFADDER routine consists of the following code:

```
syncthreads
@todo tid - $(const(4 * threads))
new @a
new @b
@a=s[tid4off + $(x + 16*in0)] if @todo signed<
@b=s[tid4off + $(x + 16*in1)] if @todo signed<
@bot=@a ^ @b
@top=@a & @b
s[tid4off + $(x + 16*out0)]=@bot if @todo signed<
s[tid4off + $(x + 16*out1)]=@top if @todo signed<
```

Overall we use 19 FULLADDER stages and 2 HALFADDER stages.

VIII. Batching operations: `multprep` etc.

The `multprep` operation takes as input a 131×128 bit matrix and transforms this matrix in place using a particular pattern of XORs. We first explain the transformation as a series of operations on 128-bit vector variables `c0`, ..., `c130` holding the matrix rows, and then discuss parallelization of the transformation.

The transformation starts with the short initial computation

```
c126^=c128
c125^=c129
c124^=c130
```

and continues with 6 levels of conversion. Level 1 consists of one computation involving 126 rows:

```
c62^=c64
c61^=c65
c60^=c66
...
c0^=c126
```

Level 2 consists of 2 computations, each involving 62 rows:

```
c30^=c32
c29^=c33
...
c0^=c62
```

and

```
c94^=c96
c93^=c97
...
c64^=c126
```

Level 3 consists of 4 computations, each involving 30 rows; level 4 of 8 computations, each involving 14 rows; level 5 of 16 computations, each involving 6 rows; and level 6 of 32 computations, each involving 2 rows. The basic structure of each of these computations on each level is XORing the upper rows into the lower rows as shown for levels 1 and 2.

We merge levels 1 and 2 of conversion, and then assign the resulting 125 computations to 128 threads as follows. As in previous sections, each group of four adjacent threads operates on the four 32-bit integers of one matrix row. We assign the first group (threads 0, 1, 2, 3) to the operations in levels 1 and 2 on `c0`, `c62`, `c64`, and `c126`:

```
c62^=c64
c0^=c126
c64^=c126
c0^=c62
```

The actual qhasm-cudasm code consists of the following 11 instructions, where x is the starting position of the input matrix in shared memory:

```
@a=s[tid4off + $(x)]
@f=s[@zoff + $(x+4*4*(62+64))]
@b=s[tid4off + $(x+4*4*64)]
@c=s[@zoff + $(x+4*4*62)]
@a^=@f
@c^=@b
@a^=@c
@b^=@f
s[tid4off + $(x)]=@a
s[tid4off + $(x+4*4*64)]=@b if @check signed<
s[@zoff + $(x+4*4*62)]=@c
```

Thread 1 performs the same operations on c1, c61, c65 and c125, and so on; thread 30 performs the operations on c30, c32, c94 and c96. In a similar way we merge levels 3 and 4 and levels 5 and 6.

Merging two consecutive levels keeps most threads busy during the whole computation, and results in only a small number of memory-bank conflicts for 128 threads working on the whole matrix.

IX. Minimizing code size: ppn and ppp

The most time-consuming operation in our software is field multiplication. This consists of a $131 \times 131 \to 261$-bit polynomial multiplication, followed by a $261 \to 131$-bit reduction. As mentioned earlier, field multiplication has two flavors, ppn and ppp; these have the same polynomial multiplication but differ in the details of the reduction.

Our polynomial-multiplication code consists of 725 instructions and uses almost all of the 16KB shared memory. See [4, Section 5] for details of our multiplication strategy. Reduction is similar to the multprep operation described in Section VIII but is about twice as long.

Our main loop originally consisted of straight-line code for a batch of B iterations, using $5B+5$ multiplications. We wanted B to be reasonably large, at least 32, so as not to notice the $+5$ overhead here, but we ran into a performance problem: 165 straight-line multiplications do not fit into the second-level GPU instruction cache. The cores cannot load instructions from global memory quickly enough to keep the ALUs busy.

To avoid this problem we reduced code size by the following compression strategy: identify a large, contiguous, frequently used part of the code, and convert it into a machine-level function. Our code uses Tesla instructions call.label and return to enter and leave the function; these instructions manage return addresses in hardware.

We are departing here from the standard practice on GPUs, namely to inline all code. There is no common calling convention; saving registers to a stack in global memory would be highly inefficient. We avoid register spills, and circumvent the introduction of a calling convention, by instructing the register allocator to find a fixed register assignment suitable for all calls to a particular function.

The largest contiguous section of code in our kernel, and also the largest consumer of time, is the 131-bit polynomial multiplication. We put just one copy of the code into the kernel, labeled as mult_131x131. All m5-level calls to the multiplication — which would normally cause inlining in the assembly code — are replaced by machine-level calls to mult_131x131.

This mechanism for function calls goes beyond any previous use of qhasm. It increases pressure on the register allocator and requires us to place careful hints to the register allocator about expired register variables. But the code works, and most of its run time is spent on instructions that fit into the instruction cache.

We also arranged large parts of the computation into size-B loops, but the most natural way to do this still contained 16 multiplications. Function calls are more flexible than loops.

We further reduced code size by using *half instructions*. Most instructions are encoded in 64-bit words, but some simple instructions can be encoded in 32-bit words:

```
shortinsn @p0 += @slicex
shortinsn @p1 += @slicex
shortinsn @p2 += @slicex
shortinsn @p3 += @slicex
```

A pair of these half instructions fills up one slot of a "regular" full instruction.

X. Instruction scheduling

NVIDIA's documentation does not suggest any real importance to the order of instructions. There are warnings regarding large latencies for global-memory access, but there is no documentation of the latencies of any other instructions, or of any other interactions between nearby instructions.

We nevertheless found two ways to save time by reordering instructions. First, given dependent instructions such as

```
@c ^= @b
@a ^= @c
```

we moved another instruction in between. If the first instruction is issued to 128 threads then it will occupy the 8 ALUs in the core for only 16 cycles, but apparently it has a latency of more than 16 cycles, stalling the next instruction if there is a dependency.

Second, we tried to avoid adjacent shared-memory accesses. There seems to be a latency of more than 16 cycles from one shared-memory instruction to the next shared-memory instruction, even if the instructions are not dependent.

After this round of optimization we found that our qhasm-cudasm code was running at 63 million iterations per second on a GTX 295, more than 4 trillion useful bit operations per second. This is only a quarter of the maximum theoretical capacity of a GTX 295 but it is more than twice as fast as our best code compiled by nvcc and ptxas.

XI. Conclusion

In this paper, we have described a high-level assembly-language toolchain, called qhasm-cudasm, that simultaneously

allows efficient programming and complete control over raw GPU hardware. `qhasm-cudasm` programmers have full control over instruction selection and scheduling, as well as register allocation and spills. Furthermore, `qhasm-cudasm` has automated the most tedious task of register assignment while giving the programmer the maximum degree of freedom in software architectural exploration, allowing creative programming to reach new heights of performance on the GPU. This allowed us to build a 90000-machine-instruction kernel for the largest cryptanalysis project in history, and a streamlined kernel outperforming the best `nvcc` implementation by 148%. We expect to be able to use the `qhasm-cudasm` toolchain to set speed records for many other applications, and in fact have already used it to almost double the speed of a computation in quantum chemistry.

There are already several projects that aim to provide analogous features to `cudasm` for the latest GPUs from NVIDIA and AMD: asfermi [10] targets NVIDIA's Fermi architecture; amdasm [13] targets AMD GPUs; calasm [12], from one of the authors of this paper (Niederhagen), also targets AMD GPUs. Future versions of our software will build upon these projects, bringing the usability of `qhasm-cudasm` to all of these GPUs.

REFERENCES

[1] Daniel V. Bailey, Lejla Batina, Daniel J. Bernstein, Peter Birkner, Joppe W. Bos, Hsieh-Chung Chen, Chen-Mou Cheng, Gauthier van Damme, Giacomo de Meulenaer, Luis Julian Dominguez Perez, Junfeng Fan, Tim Güneysu, Frank Gurkaynak, Thorsten Kleinjung, Tanja Lange, Nele Mentens, Ruben Niederhagen, Christof Paar, Francesco Regazzoni, Peter Schwabe, Leif Uhsadel, Anthony Van Herrewege, and Bo-Yin Yang. Breaking ECC2K-130. Cryptology ePrint Archive, Report 2009/541, 2009. http://eprint.iacr.org/2009/541.

[2] Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In *Public Key Cryptography—PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006. http://cr.yp.to/papers.html#curve25519.

[3] Daniel J. Bernstein. `qhasm` software package, 2007. http://cr.yp.to/qhasm.html.

[4] Daniel J. Bernstein, Hsieh-Chung Chen, Chen-Mou Cheng, Tanja Lange, Ruben Niederhagen, Peter Schwabe, and Bo-Yin Yang. ECC2K-130 on NVIDIA GPUs. In *Progress in Cryptology—INDOCRYPT 2010*, volume 6498 of *LNCS*, pages 328–346. Springer, 2010. http://eprint.iacr.org/2012/002/.

[5] Daniel J. Bernstein and Tanja Lange. Type-II optimal polynomial bases. In *Arithmetic of Finite Fields—WAIFI 2010*, volume 6087 of *Lecture Notes in Computer Science*, pages 41–61. Springer, 2010. http://eprint.iacr.org/2010/069.

[6] Daniel J. Bernstein and Peter Schwabe. New AES software speed records. In *Progress in Cryptology—INDOCRYPT 2008*, volume 5365 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2008. http://eprint.iacr.org/2008/381.

[7] Certicom. Certicom ECC challenge. http://www.certicom.com/images/pdfs/cert_ecc_challenge.pdf, 1997.

[8] Lung-Sheng Chien. Hand-tuned SGEMM on GT200 GPU. http://oz.nthu.edu.tw/~d947207/NVIDIA/SGEMM/HandTunedSgemm_2010_v1.1.pdf, 2010.

[9] Neil Costigan and Peter Schwabe. Fast elliptic-curve cryptography on the Cell Broadband Engine. In *Progress in Cryptology—AFRICACRYPT 2009*, volume 5580 of *Lecture Notes in Computer Science*, pages 368–385. Springer, 2009. http://eprint.iacr.org/2009/016.

[10] Yunqing Hou. asfermi: An assembler for the NVIDIA Fermi instruction set, 2011. http://code.google.com/p/asfermi/, accessed Nov. 1, 2011.

[11] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman J. J. te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 333–350. Springer, 2010.

[12] Ruben Niederhagen. Calasm, 2011. http://www.polycephaly.org/projects/calasm/.

[13] Ádám Rák. AMD-GPU-Asm-Disasm, 2011. https://github.com/rakadam/AMD-GPU-Asm-Disasm/, accessed Nov. 1, 2011.

[14] Wladimir J. van der Laan. Cubin utilities. http://wiki.github.com/laanwj/decuda/, 2007.

[15] William A. Ward, Jr. Algorithm 803: a simpler macro processor. *ACM Transactions on Mathematical Software*, 26:310–319, 2000.

Solving Circuit Optimisation Problems in Cryptography and Cryptanalysis*

Nicolas T. Courtois[1,2], Daniel Hulme[1,2], and Theodosis Mourouzis[1]

[1] University College London, UK,
[2] NP-Complete Ltd, London, UK

Abstract. In this paper we look at the problem of compact representation and optimization of some small circuits such as S-boxes in cryptographic algorithms. Our ambition is to show that it is a lot more than an essential software task in industrial hardware implementations of standard cryptographic algorithms [20, 25, 13, 8]. We show that that it can be done in a new way and should lead to many exciting new applications in particular in cryptanalysis.

The starting point is the notion of Multiplicative Complexity recently used to find very good optimizations of the AES S-box [5, 8, 7]. We have developed a method and software to optimize the multiplicative complexity and also the linear components using SAT solvers. We produce a compact implementation of two block ciphers PRESENT and GOST known for their exceptionally low hardware cost. We cover all the numerous variants of GOST and have released an open source bitslice implementation of PRESENT which is now the best publicly available [1]. We explain why our methodology is suitable and should be used in implementations aiming at preventing side channel attacks on cryptographic chips such as DPA. We show that our method can give better results that with standard and gate-level logical optimization tools such as the famous Berkeley Logic Friday software. Moreover, most of our results are **optimal** and cannot be improved which is a rare achievement in complexity.

This kind of optimizations can be seen as an essential ingredient and key step underlying a number of recent attacks on symmetric ciphers such as in [17, 16], and any further optimization effort should result in either improved attack speeds or a proof of optimality. For example we have been able to obtain a single-key attack on the full 32-round GOST block cipher which is faster than brute force.

Key Words: Block ciphers, PRESENT, GOST, non-linearity, algebraic attacks, circuit complexity, logic-level optimization, multiplicative complexity, algebraic cryptanalysis, block cipher implementation, bitslice implementation, side-channel attacks, DPA.

* This research was supported by the European Commission under the FP7 project number 242497 "Resilient Infrastructure and Building Security (RIBS)" and by the UK Technology Strategy Board under project 9626-58525.

Note: A short and early version of this paper was included in the electronic proceedings of 2nd IMA Conference Mathematics in Defence 2011, 20 October 2011, Defence Academy of the United Kingdom, Swindon, UK.

1 Introduction

The problems of circuit complexity is one of the hardest and yet very important problems in computer science and complexity theory. For the great majority of concrete circuits, it is not known what will be the lowest bound on their complexity, neither how to compute very good circuits efficiently. Not everybody in the industry cares about improving their gate count by a small factor, but such optimizations are particularly important in hardware implementation of standard cryptographic algorithms [20, 25, 13, 8], which in many security chips such as smart cards and RFID, will be one of the most costly components. Here even a small gain can produce measurable savings.

Many heuristic algorithms for this problem have been invented, and with a lot of computing power one can find very decent optimizations [20], but these optimizations are frequently subject to further substantial improvement. In this paper we particularly focus on optimizing the S-boxes for industrial block ciphers.

Much less known and very surprising is that this is also an important topic in cryptanalysis. As shown in [13, 16, 14] such optimizations are also very important in order to speed up so called algebraic attacks on symmetric ciphers, and in the space of attacks which require very small quantities of data, these methods lead to currently best known attacks on a few ciphers (with more data, typically faster attacks will exist). In this paper we focus mostly on 4x4 S-boxes in ciphers such as PRESENT and GOST. These ciphers are known for their exceptionally low hardware implementation cost [25]. But this is also what makes them vulnerable to algebraic cryptanalysis.

Sometimes the very existence of an attack on the cipher which would be faster than brute force will depend on a concrete circuit optimization problem, precisely because the time complexity must be fast enough to beat the brute force attacks. Our work on cryptanalysis makes extensive use of SAT solver software, both at the optimization stage, when a "compact algebraic description" of a cipher is produced, and a later solving stage, where the equations are solved to in order to compute the secret key.

1.1 Specific Topics of Interest

In 2008 Boyar and Peralta introduced a new heuristic methodology to minimize the complexity of digital circuits [5, 8, 7]. It is based on the notion of Multiplicative Complexity (MC).

Multiplicative Complexity (MC) is a well-known and very deep notion of arithmetic complexity invariant w.r.t. affine transformations, which minimizes the number of non-linear elementary transformations, see [30, 4, 5]. Their main heuristics is that a two step-process based on MC appears to be able to produce very good gate efficient implementation of several famous circuits such as the AES S-box, and some other circuits related to finite fields and algebra, Several such results can be found in [8, 7]. In this paper we apply this methodology to some cryptographically significant functions $GF(2)^4 \rightarrow GF(2)^4$ (i.e. 4x4 S-boxes). We developed software which allows us to compute **optimal** representations of these S-boxes w.r.t to this methodology.

1.2 Motivations For Achieving Low-MC and Low Gate Count

More generally, we are interested in all sorts of minimal decompositions of cryptographic circuits into very basic components. We can see at least six distinct reasons and motivations, which will leading to closely related (but in fact distinct) requirements for such optimizations. For example we may want to:

1. Lower the hardware implementation cost of a cipher in silicon.
2. Develop certain software implementations such as in [1].
3. Prevent Side Channel Attacks (SCA) on smart cards such as Differential Power Analysis (DPA) [26]. Here Multiplicative Complexity (MC) seems perfect: XORs are believed easier to protect against side-channel attacks, see [26] and minimizing the number of AND gates is likely to minimize the overall cost of such protections. More generally we expect that MC and similar optimizations are very helpful especially in general-purpose protection methods against side channel attacks, which aim at implement securely completely arbitrary digital circuits see [26, 23]. For particular ciphers, there may be better (dedicated) solutions.
4. We may try to discover hidden vulnerabilities inside ciphers.
5. It is known that certain algebraic software attacks on symmetric ciphers such as in [13, 14] benefit from very compact representations of S-boxes.
6. In symbolic computing and numerical algebra, this kind of optimizations can be applied recursively to produce asymptotically fast algorithms to solve very famous and important practical problems such as Gaussian reduction and matrix multiplication, see [10].

2 Bitslice Gate Complexity and Multiplicative Complexity

In this section we define two particular models for gate complexity of digital circuits.

Definition 2.0.1 (Bitslice Gate Complexity (BGC)).
Given a function $GF(2)^n \to GF(2)^m$ we define its Bitslice Gate Complexity (BGC) as the **minimum** number of 2-input gates of types XOR,OR,AND,OR needed.

Note: we do NOT allow gates of type NOR and NAND. This is a very simple model, in which the cost of all these gates is considered to be the same, and which is relevant for example in so called Bit-slice implementations of block ciphers, such as for example in [1]. However it is not an optimal model for silicon implementations, where certain gates are more costly to implement, while NOR and NAND gates are actually less costly.

Now we recall the definition of MC [30, 4, 5, 8]:

Definition 2.0.2 (Multiplicative Complexity (MC)).
Given a function $GF(2)^n \to GF(2)^m$ we define its Multiplicative Complexity (MC) as the minimum number of AND gates which need to be used to implement this function, with an unlimited number of XOR and NOT gates.

This model considers that linear operations come "for free" and ask to minimize just the number of AND gates. The problem with Bitslice Gate Complexity (BGC) is that we are not in general able to determine its value, algorithms which find such optimizations are typically random stochastic explorations of large trees of solutions [20] and we are not sure if the optimizations are final or if they can still be improved. However, as we will see in this paper, at least for small circuits, the Multiplicative Complexity (MC) can be computed **exactly** by our methods which use SAT solver software.

We have also the following fact which results directly from the definition:

Fact 1. The Multiplicative Complexity is invariant w.r.t. to multivariate affine transformations at the input and at the output. As a consequence, for a 4x4 bit S-box, its Multiplicative Complexity is the same for the whole affine equivalence class, which classes are studied in [22, 28].

2.1 Multiplicative Complexity As A Tool For Gate Complexity

Boyar and Peralta have a developed a heuristic methodology, where they optimise for of Multiplicative Complexity (MC) in order to produce also gate-efficient implementations:

1. **(Step 1)** First compute the multiplicative complexity.
2. **(Step 2)** Then optimise the number of XORs separately, see [6, 19].
3. Optional Step 3: At the end do additional optimizations to decrease the circuit depth, and possibly additional software optimizations, see [5, 8],

This methodology was then used to produce new worldwide records in gate efficient implementation of several famous circuits such as the AES S-box, and many other circuits related to finite fields and algebra, [8, 7, 7].

2.2 Our Method to Compute the Multiplicative Complexity

In this paper we focus on optimisation of functions $GF(2)^4 \rightarrow GF(2)^4$ which are immensely popular in cryptography [28]. We have implemented fully and with our own optimisation methods, both Steps 1. and 2. above.

The crucial feature of our implementation is that BOTH our Steps 1. and 2. are OPTIMAL, i.e. they produce the best possible optimizations which can be obtained by following these two steps. Optimality was achieved due to SAT solver software, we convert our problem to SAT and it either outputs SAT, and a solution, which we convert to a concrete circuit optimization, or it outputs UNSAT, and we are certain that there is no solution. There is third possibility, that the SAT solver software runs for a very long time and we do not have enough computing power to decide whether the result is SAT or UNSAT, but this have never happened for 4x4 S-boxes. Accordingly, we were able to produce optimal optimizations or this type for every 4x4 S-box we have ever tried. This is very rare in complexity: to be able to completely determine the best possible result.

We must say that these methods are at prototyping stage and they are so far slower than other known methods [20]. Likewise, we do not claim that we can

optimise the linear parts as quickly as by recent methods described in [5, 6], but only that we can optimize to the strictest minimum possible, which probably can also be achieved in [19] by similar methods and SAT solvers. This is for linear circuits. However it seems that we are the first to apply SAT solvers also to optimize non-linear circuits.

We have also obtained some very good results on bi-linear circuits, see [10].

2.3 Provable Aspects of Our Method

Our solutions are optimal and thus proven to be impossible to improve (automated software proof with UNSAT). This is they would be provably optimal, if we had a proof of correctness of the SAT solver software.

It will also be correct, but not proven correct, if there is no bug in the SAT solver software. Such a bug, where a problem which is SAT is claimed to be UNSAT by another solver, will quickly and easily be found, because we have a portfolio of many different SAT solver software, and regularly check these results by at least a few SAT solvers. Even if we assume the presence of bugs in this software, one can consider that our proofs are "probabilistic proofs", but still the probability of error can be easily made as small as desired.

Thus we achieve a proof of impossibility when our program outputs UNSAT for smaller sizes. We also claim that what we do could be extended to produce fully verifiable mathematical proofs written in a formal language, which prove these optimality results. Some SAT solvers already have the ability to output such proofs. However what is missing is also a proof that all our conversions are correct and preserve correctness. This can be done in future research. Such proofs would not be published in scientific papers, but rather as lengthy computer files, which should come together with a formal system able to efficiently check the correctness of such proofs. This is a major topic for further research which would require one to develop a whole new formal language and software to manipulate it.

2.4 An Alternative Method to Compute the Multiplicative Complexity

We can note that for Step 1, and only for 4x4 S-boxes, there is a simple and alternative method to compute the of Multiplicative Complexity (MC), in step 1, following the work on classification and equivalence of 4x4 S-boxes [22, 28]. It is as follows:

1. Determine another S-box for which our S-box is an affine equivalent of another S-box, for which the MC was already computed.
2. The affine equivalence can be determined by methods of [2] which are actually essentially the same methods which have been proposed at the same conference 10 years earlier [9] in a slightly different context.

3 Optimizing the PRESENT S-box

The PRESENT S-box is defined as $\{12, 5, 6, 11, 9, 0, 10, 13, 3, 14, 15, 8, 4, 7, 1, 2\}$. We will number the least significant bits starting from 1.

Theorem 3.0.1. The Multiplicative Complexity of the PRESENT S-box is exactly 4.

Proof: For 3 AND gates our thoroughly designed and tested system outputs UNSAT. This could be converted to a formal proof that the Multiplicative Complexity is at least 3. We have obtained an automated proof of this fact which takes a few seconds on a PC and can reproduced and checked. For 4 AND gates, our system outputs SAT and a solution. Further optimisation of the linear part, which is also optimal as we also obtained UNSAT for lower numbers, allowed us to minimize the number of XORs to the strict minimum possible (prove by additional UNSAT results). As a result, for example we have obtained an implementation of the PRESENT S-box with 25 gates, 4 AND, 20 XOR, 1 NOT which is optimal w.r.t our Boyar-Peralta 2-step methodology but not optimal in overall gate complexity. 25 gates are still not very satisfactory.

A better result in terms of gate complexity can be achieved by the following method: we observe that AND gates and OR gates are affine equivalents, and it is likely that **if** we implement certain AND gates with OR gates, we might be able to further reduce the overall complexity of the linear parts. We may try all possible 2^4 cases where some AND gates are implemented with OR gates. Even better results can be obtained if we consider also NOR and NAND gates. By this method, starting with the right optimization with MC=4, as several such optimizations may exist, we can obtain the following new implementation of the PRESENT S-box which requires only 14 gates total (!):

```
T1=X2^X1; T2=X1&T1; T3=X0^T2; Y3=X3^T3; T2=T1&T3; T1^=Y3; T2^=X1;
T4=X3|T2; Y2=T1^T4; T2^=~X3; Y0=Y2^T2; T2|=T1; Y1=T3^T2;
```

Fig. 1. Our implementation of the PRESENT S-box with only 14 gates

Applications. This implementation is used in our recent bit-slice implementation of PRESENT, see [1]. In addition we postulate that this implementation of the PRESENT S-box is in certain sense optimal for DPA-protected hardware implementations with linear masking, as it minimizes the number of non-linear gates (there are only 4 such gates).

Discussion. Our best optimisation of the PRESENT S-box does **not** contradict the Boyar-Peralta heuristic to the effect that some of the best possible gate-efficient implementations are very closely related to the notion of multiplicative complexity. However the most recent implementations of the AES S-box, in the second paper by Boyar and Peralta, show that further improvements, and also circuit depth improvements, can be achieved also by relaxing the number of ANDs used as in the latest optimization of the 4-bit inverse in $GF(2^4)$ for AES given on Fig 1. in [8].

4 The GOST S-boxes

We consider the main standard and most widely known version of the GOST block cipher, known as "GostR3411_94_TestParamSet" in [21]. and also known as the one used by the Central Bank of the Russian Federation [25]. By running the same method and programs we obtained the following result:

Theorem 4.0.2. The Multiplicative Complexity of the eight GOST S-boxes S1,S2,S3,S4,S5,S6,S7,S8 is exactly equal to respectively 4,5,5,5,5,5,4,5.

Related Work: We can compare this to the results in Table on page 226 of [25] where we see that these 8 S-boxes are also on average more expensive than the PRESENT S-box in the sense of Gate Equivalent (GE) cost, (the GOST S-boxes cost 23.5 GE on average per S-box while the PRESENT S-box appears to require about 27 GE). In Table 3 in [25] we see that PRESENT S-box is better against linear and differential cryptanalysis. However in our Multiplicative Complexity (MC) metric, in our Bitslice Gate Multiplicative Complexity (BGC) metric, and also in the strict GE cost metric in [25], it is clear that the complexity of the PRESENT S-box is always lower and therefore we conjecture that PRESENT S-box will be weaker than the GOST S-boxes, against many types of algebraic cryptanalysis such as attacks described in [16, 17]. Thus it is probably a bad idea to use the GOST cipher with the PRESENT S-box, as proposed in [25].

4.1 Additional Standard GOST S-boxes

Remark: In the future works we will publish much more results for all the 64 known GOST S-boxes and their inverses, and also other optimizations of these S-boxes, and also the exact application of these results in cryptanalysis. The table below contains some preliminary results.

Table 1. Multiplicative Complexity for all known GOST S-Boxes

S-box Set Name	$S1$	$S2$	$S3$	$S4$	$S5$	$S6$	$S7$	$S8$
GostR3411_94_TestParamSet	4	5	5	5	5	5	4	5
GostR3411_94_CryptoProParamSet	4	5	5	4	5	5	4	5
Gost28147_TestParamSet	4	4	4	4	4	5	5	5
Gost28147_CryptoProParamSetA	5	4	5	4	4	4	5	5
Gost28147_CryptoProParamSetB	5	5	5	5	5	5	5	5
Gost28147_CryptoProParamSetC	5	5	5	5	5	5	5	5
Gost28147_CryptoProParamSetD	5	5	5	5	5	5	5	5
GostR3411_94_SberbankHashParamset	4	4	4	5	5	4	4	4

We believe that this table gives some first and early indications which versions of GOST will be more secure against algebraic cryptanalysis, this however requires much more extra work.

5 Multiplicative Complexity of Whole Ciphers

It appears that, from here we are able to **provably minimize** the number of non-linear gates in a whole given cipher, to a proven lower bound.

In order to do this we need to look at any other existing non-linear components of the cipher, and also compute their Multiplicative Complexity (MC).

Then we also need to prove that the Multiplicative Complexity is not reduced by the combination.

Such a reduction is not always very likely, but if it occurs, it could be considered as a potential structural flaw in the cipher. It could be seen as a sign that somewhat the designers have maybe "wasted" the computational resources in hardware, for a given security level. Alternatively, it could also be a source of potential shortcuts to implement the cipher more efficiently.

6 Multiplicative Complexity of Whole GOST Cipher

We would like to minimize the number of non-linear gates in the whole given cipher. We sketch how this can be done for the GOST block cipher. We basically need to compute the Multiplicative Complexity (MC) for each component and add them.

6.1 Modular Addition

In addition to S-boxes, the GOST cipher uses addition modulo 2^{32}. The interesting question is what is the multiplicative complexity of this operation.

In order to optimize this addition modulo 2^{32} we follow the first method described in [15]. Let us consider three n-bit words $(x_{n-1}, \ldots, x_0)$, $(y_{n-1}, \ldots, y_0)$ and $(z_{n-1}, \ldots, z_0)$ with z_0 being the low-order bit. The modular addition

$$(x, y) \mapsto z = x \boxplus y \mod 2^n$$

can be described the following way by $(*)$ and $(*')$, using new variables that are carry bits, represented by the $(n-1)$-bit word $c = (c_{n-1}, \ldots, c_1)$:

$$(*) \begin{cases} z_0 = x_0 + y_0 \\ z_1 = x_1 + y_1 + c_1 \\ z_2 = x_2 + y_2 + c_2 \\ \vdots \\ z_i = x_i + y_i + c_i \\ \vdots \\ z_{n-1} = x_{n-1} + y_{n-1} + c_{n-1}, \end{cases} \qquad (*') \begin{cases} c_1 = x_0 y_0 \\ c_2 = x_1 y_1 + (x_1 + y_1) c_1 \\ \vdots \\ c_i = x_{i-1} y_{i-1} + (x_{i-1} + y_{i-1}) c_{i-1} \\ \vdots \\ c_{n-1} = x_{n-2} y_{n-2} + (x_{n-2} + y_{n-2}) c_{n-2} \end{cases}$$

We claim that:

Theorem 6.1.1. The Multiplicative Complexity (MC) of the addition modulo 2^n is exactly $n - 1$.

Proof: This is is not obvious at the first sight, it may seem that it is $2(n-1)$. However in characteristic 2 we have:

$$xy + (x+y)c = (x+c)(y+c) + c$$

which allows to reduce the number of multiplications to 1 in each line: we obtain $(x_{i-1} + c_{i-1})(y_{i-1} + c_{i-1}) + c_{i-1}$. Thus we have established it is at most $n-1$.

To prove it is at least $n-1$ we observe that the algebraic degree of the ANF of the last output bit z_{n-1} as a function of the x_i and the y_i is always $n-1$. This is easy to see from the formulas because each new carry c_i contains a multiplication of the previous carry c_{i-1} by new independent variables. Therefore the ANF degree of z_{n-1} is $n-1$ and at least $n-1$ multiplications are needed to compute it, and therefore at least $n-1$ multiplications are needed overall.

7 Application to Cryptanalysis of GOST

It appears that we are able to **provably minimize** the number of non-linear gates in a whole given cipher such as GOST, to a proven lower bound.

Now we can encode the whole GOST cipher as follows, which is based on the concept of the multiplicative complexity, but some details can only be figured out experimentally (we also need to keep the equations sparse and not too introduce too many additional variables):

1. For $\boxplus$, surprisingly, we use our encoding with $2(n-1)$ multiplications, not the optimal one from Theorem 6.1.1 (which gives comparable but slower results).
2. For the multiplicative complexity we use the optimal implementations we have obtained with our SAT solver software. Surprisingly, the attack is slightly faster if we do NOT try to further to reduce the number of XORs to the smallest possible value (which is an option in our software).

7.1 How To Solve It?

Now we will convert the system of equations obtained to SAT and solve it. The best results are obtained as follows:

1. We use the Courtois-Bard-Jefferson converter. A Java source code and a working Windows distribution of this program can be found at [12].
2. We use the well-known open-source SAT solver MiniSat 2.06. [24] we have confirmed by a substantial amount of simulations and testing that with default options, MiniSat 2.06. is up to several times faster than later versions of MiniSat and also much faster than CryptoMiniSat, which is another very well-known SAT solver based on MiniSat.

From here we obtained the following result:

Fact 2 (Key Recovery for 4 Rounds and 2 KP). Given 2 P/C pairs for 4 rounds of GOST the 128-bit key can be recovered in time equivalent to 2^{24} GOST encryptions on the same software platform (it takes a few seconds). The memory requirements are very small.

Justification: A ready windows executable program and all the necessary files (one for each S-box), to do just that, together with the solver software, can be obtained from the authors. The command line options used are:

```
axel.exe 1711 4 /ins2 /sat /fix0 /mcom /bard 0 0 1-100
```

This will run 100 randomized instances of the problem and display various statistics including the median running time for 100 runs which is less than 10 seconds on a modern CPU.

7.2 How To Break the Full 32-round GOST

Several different black-box reduction methods to transform Fact 2 which only break 4 rounds of GOST to **a single-key attack on a full 32-round GOST faster than brute force** can be found in [17, 16]. The fastest of these attacks has a time complexity of about 2^{216} GOST computations and requires 2^{64} KP.

What's Next? This paper can be seen as developing a sort of "basic technology" which underpins a large number of software cryptographic attacks such as Fact 2, which in turn are an essential and indispensable ingredient and building block in an even larger number of cryptanalytic attacks (cf. [14, 17, 16]). Any improvement in this basic technology is therefore likely to improve many different cryptographic attacks.

Currently no theory is able to give recommendations about how to produce the fastest algebraic attack on a given cipher, and there are many competing techniques for solving the NP-hard problems involved in these attacks, see for example [13, 18, 27]. The crucial question is the choice of representation which we can consider to be a form of "Algebraization" of something such as a block cipher which is precisely designed not to be easy to break by solving a system of algebraic equations. We view it as a major practical problem in cryptography and it should be seen also as a problem of optimisation. We point out that it is a very closely related problem to the problem of efficient implementation of cryptographic S-boxes and we use very similar techniques for both problems.

This is an area where there is a need to built sophisticated optimization software. We should however note that these optimizations need to be run only once and their running time is therefore irrelevant for the efficiency of the actual key recovery attack which is an independent question.

We conjecture that the possibility to reduce the Multiplicative Complexity (MC) of the whole cipher to the lowest possible number, and also other metrics of circuit complexity, should play an important role in finding the best possible attacks in algebraic cryptanalysis.

8 Conclusion

In this paper we study the notion of Multiplicative Complexity (MC) which minimizes the number of elementary non-linear operations (AND gates) at the cost of linear operations, which can also minimized separately, as a second step. We have implemented both these steps in an innovative way, where each problem is converted to a satisfiability problem and solved by SAT solver software.

This type of methodology was previously applied to optimize linear circuits [19] and bi-linear circuits [10] and even (with a lot more work) to derive the best AES S-box yet found [5, 8] but it appears it is for the first time it is used to optimise arbitrary non-linear S-boxes.

The key interesting point is that many SAT solvers will be able to detect when the problem is not solvable, leading to results which are proven to be optimal, a rare thing in complexity. Thus we are able to compute Multiplicative Complexity (MC) **exactly**, for all sufficiently small circuits, and also to optimize the linear parts exactly. Our method is practical though rather slow, so far we have been able to optimize every 4x4 S-box we tried, but not many larger S-boxes. Yet it is a unique and very powerful method, because all the results are optimal and one could produce and publish a formal mathematical proof (automatically found by the software) that they cannot be improved.

We have applied this notion to derive very efficient implementations of the S-boxes in two ciphers, PRESENT and GOST. Our optimization is the key ingredient in a new open-source bitslice implementation of PRESENT which we have released [1].

Interestingly, from here we are able to **provably minimize** the number of non-linear gates in a whole given cipher such as PRESENT or GOST, to a rather unexpectedly low number such as 4 or 5 per S-box.

This has two sorts of applications in cryptography. First, such optimizations are important in synthesis of implementations of circuits secure against side-channel attacks, which is an important and hot research topic, see for example [26, 23].

Moreover, we show how to use these optimizations to break the full-round block cipher GOST and some variants [21, 25]. It is extremely rare to see a real-life block cipher which can be broken faster than brute force.

References

1. Martin Albrecht, Nicolas T. Courtois, Daniel Hulme, Guangyan Song: *Bit-Slice Implementation of PRESENT in pure standard C,* v1.5, 26/08/2011, open-source code available at `https://bitbucket.org/malb/algebraic_attacks/src/tip/present_bitslice.c`
2. A. Biryukov, C. De Cannière, A. Braeken, and B. Preneel: *A toolbox for cryptanalysis: Linear and affine equivalence algorithms.* In Eurocrypt 2003, LNCS 2656, pp. 3350, Springer, 2003.
3. A. Bogdanov, L.R. Knudsen, G. Leander, C. Paar, A. Poschmann, M.J.B. Robshaw, Y. Seurin, and C. Vikkelsoe: *PRESENT: An Ultra-Lightweight Block Cipher,* In CHES 2007, LNCS 4727, pp. 450466, Springer, 2007.
4. Joan Boyar, René Peralta, Denis Pochuev: *On the multiplicative complexity of Boolean functions over the basis (AND, XOR, 1),* In Theor. Comput. Sci. 235(1): 43-57 (2000).
5. Joan Boyar, René Peralta: *A New Combinational Logic Minimization Technique with Applications to Cryptology.* In SEA 2010: 178-189.
   An early version was published in 2009 at `http://eprint.iacr.org/2009/191`. It was revised 13 Mar 2010.
6. Joan Boyar, Philip Matthews, René Peralta: *On the Shortest Linear Straight-Line Program for Computing Linear Forms,* In MFCS 2008: 168-179.
7. Web page with all circuit minimialisation results obtained at Yale University, `http://cs-www.cs.yale.edu/homes/peralta/CircuitStuff/CMT.html`.
8. Joan Boyar and Rene Peralta; *A depth-16 circuit for the AES S-box,* `http://eprint.iacr.org/2011/332`
9. Jacques Patarin, Nicolas Courtois, Louis Goubin: *Improved Algorithms for Isomorphism of Polynomials*; Eurocrypt'98, LNCS 1403, Springer, pp.184-200,
10. Nicolas T. Courtois, Gregory V. Bard and Daniel Hulme: A New General-Purpose Method to Multiply 3x3 Matrices Using Only 23 Multiplications, At `http://arxiv.org/abs/1108.2830`.
11. Nicolas Courtois: *General Principles of Algebraic Attacks and New Design Criteria for Components of Symmetric Ciphers,* in AES 4, LNCS 3373, pp. 67-83, Springer, 2005.
12. Gregory V. Bard, Nicolas T. Courtois and Chris Jefferson: *Efficient Methods for Conversion and Solution of Sparse Systems of Low-Degree Multivariate Polynomials over GF(2) via SAT-Solvers,* `http://eprint.iacr.org/2007/024/`.
13. Nicolas Courtois, Gregory V. Bard: *Algebraic Cryptanalysis of the Data Encryption Standard,* In Cryptography and Coding, 11-th IMA Conference, pp. 152-169, LNCS 4887, Springer, 2007. Preprint available at `eprint.iacr.org/2006/402/`.
14. Nicolas Courtois, Gregory V. Bard, David Wagner: *Algebraic and Slide Attacks on KeeLoq,* In FSE 2008, pp. 97-115, LNCS 5086, Springer, 2008.
15. Nicolas Courtois and Blandine Debraize: *Algebraic Description and Simultaneous Linear Approximations of Addition in Snow 2.0.,* In ICICS 2008, 10th International Conference on Information and Communications Security, 20 - 22 October, 2008, Birmingham, UK. In LNCS 5308, pp. 328-344, Springer, 2008.
16. Nicolas Courtois: *Algebraic Complexity Reduction and Cryptanalysis of GOST,* Preprint available at `http://www.nicolascourtois.com/papers/gostac11.pdf`.
17. Nicolas Courtois: *Security Evaluation of GOST 28147-89 In View Of International Standardisation,* in Cryptologia, Volume 36, Issue 1, pp. 2-13, 2012. An earlier version which was officially submitted to ISO in May 2011 can be found at `http://eprint.iacr.org/2011/211/`.

18. Jean-Charles Faugère: *A new efficient algorithm for computing Gröbner bases without reduction to zero (F5),* Workshop on Applications of Commutative Algebra, Catania, Italy, 3-6 April 2002, ACM Press.
19. Carsten Fuhs and Peter Schneider-Kamp: *Synthesizing Shortest Linear Straight-Line Programs over GF(2) Using SAT,* In SAT 2010, Theory and Applications of Satisfiability Testing, Springer LNCS 6175, pp. 71-84, 2010.
20. B. R. Gladman, software for efficient boolean function decompositions for the eight Serpent S boxes and their inverses, available at `http://gladman.plushost.co.uk/oldsite/cryptography_technology/serpent/index.php`.
21. A Russian reference implementation of GOST implementing Russian algorithms as an extension of TLS v1.0. is available as a part of OpenSSL library. The file gost89.c contains eight different sets of S-boxes and is found in OpenSSL 0.9.8 and later: `http://www.openssl.org/source/`
22. Gregor Leander, Axel Poschmann: *On the Classification of 4 Bit S-Boxes, In Proceedings of WAIFI'07, 1st international workshop on Arithmetic of Finite Fields.*
23. *Svetla Nikova, Vincent Rijmen, Martin Schläffer: Secure Hardware Implementation of Nonlinear Functions in the Presence of Glitches,* Special Issue on Hardware and Security of Journal of Cryptology, 27 pages, 2011. `http://homes.esat.kuleuven.be/~snikova/JOC_2011.pdf`
24. MiniSat 2.0. An open-source SAT solver package, by Niklas Eén, Niklas Sörensson, cf. `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/`
25. Axel Poschmann, San Ling, and Huaxiong Wang: *256 Bit Standardized Crypto for 650 GE GOST Revisited,* In CHES 2010, LNCS 6225, pp. 219-233, 2010.
26. Emmanuel Prouff, Christophe Giraud, Sébastien Aumônier: *Provably Secure S-Box Implementation Based on Fourier Transform,* In CHES 2006, Springer LNCS 4249, pp: 216-230, 2006. Slides can be found at `http://www.iacr.org/workshops/ches/ches2006/presentations/EmmanuelProuff.pdf`
27. Haavard Raddum and Igor Semaev: New Technique for Solving Sparse Equation Systems, ECRYPT STVL website, January 16th 2006, available also at `eprint.iacr.org/2006/475/`
28. Markku-Juhani O. Saarinen: *Cryptographic Analysis of All 4 x 4 - Bit S-Boxes,* In SAC 2011, August 2011 Toronto, Canada, Springer LNCS. A version is available at `eprint.iacr.org/2011/218/`.
29. I. Schaumuller-Bichl: *Cryptanalysis of the Data Encryption Standard by the Method of Formal Coding,* In Cryptography, Proc. Burg Feuerstein 1982, LNCS 149, pp. 235-255, Springer-Verlag, 1983.
30. Claus-Peter Schnorr: *The Multiplicative Complexity of Boolean Functions,* In Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, AAECC-6, LNCS 357, pp. 45-58, 1988.

Platinum Sponsor:

Sponsors: