Parallel Cryptanalysis

Ruben Niederhagen

Parallel Cryptanalysis

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op maandag 23 april 2012 om 16.00 uur

door

Ruben Falko Niederhagen

geboren te Aken, Duitsland

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. T. Lange
en
prof.dr. D.J. Bernstein

Copromotor:
dr. C.-M. Cheng

A catalogue record is available from the Eindhoven University of Technology Library.
ISBN: 978-90-386-3128-8

Printed by Printservice Technische Universiteit Eindhoven
Cover design by Verspaget & Bruinink, Nuenen

Public domain

Commissie:

prof.dr. D.J. Bernstein, promotor (University of Illinois at Chicago)
dr. C.-M. Cheng, copromotor (National Taiwan University)
prof.dr. A.M. Cohen, chairman
prof.dr. R. Bisseling (Utrecht University)
prof.dr. T. Lange, promotor
prof.dr. W.H.A. Schilders
prof.dr. G. Woeginger
prof.dr. B.-Y. Yang (Academia Sinica)

Acknowledgements

I would like to thank my supervisors Daniel J. Bernstein and Tanja Lange as well as my supervisors in Taiwan, Chen-Mou Cheng and Bo-Yin Yang, for the opportunity to enjoy my PhD studies commuting between the Netherlands and Taiwan. Furthermore, I would like to thank them for their support throughout my studies and the writing of my thesis.

I also would like to thank Rob Bisseling, Arjeh Cohen, Wil Schilders, and Gerhard Woeginger for joining my doctorate committee and for their valuable feedback on the draft of my dissertation.

Special thanks go to Tung Chou, Andreas Gierlich, and Peter Schwabe for proofreading of the first versions of my thesis and for suggesting corrections and improvements.

Finally, I want to thank Stefan Lankes from the Lehrstuhl für Betriebssysteme (research group for operating systems) at RWTH Aachen University for his support in the first years of my PhD studies.

Contents

1 Introduction 1

2 Overview of parallel computing 3
2.1 Parallel architectures 5
2.1.1 Microarchitecture 5
2.1.2 Instruction set 7
2.1.3 System architecture 8
2.2 Parallel programming 12
2.2.1 Shared memory 12
2.2.2 Message passing 14
2.2.3 Summary 14
2.3 General-purpose GPU programming 15
2.3.1 Programming NVIDIA GPUs 17
2.3.2 Assembly for NVIDIA GPUs 18

3 Parallel implementation of Pollard's rho method 23
3.1 The ECDLP and the parallel version of Pollard's rho method . . . 24
3.2 ECC2K-130 and the iteration function 25
3.3 Implementing ECC2K-130 on the Cell processor 26
3.3.1 A Brief Description of the Cell processor 27
3.3.2 Approaches for implementing the iteration function 29
3.3.3 ECC2K-130 iterations on the Cell processor 31
3.3.4 Using DMA transfers to increase the batch size 33
3.3.5 Overall results on the Cell processor 34
3.4 Implementing ECC2K-130 on a GPU 35
3.4.1 The GTX 295 graphics card 36
3.4.2 Approaches for implementing the iteration function 38
3.4.3 Polynomial multiplication on the GPU 39
3.4.4 ECC2K-130 iterations on the GPU 42
3.4.5 Overall results on the GPU 44
3.5 Performance comparison 45

4 Parallel implementation of the XL algorithm **47**
4.1 The XL algorithm . . . 48
4.2 The block Wiedemann algorithm . . . 49
4.3 The block Berlekamp–Massey algorithm . . . 53
  4.3.1 Reducing the cost of the Berlekamp–Massey algorithm . . . 54
  4.3.2 Parallelization of the Berlekamp–Massey algorithm . . . 55
4.4 Thomé's version of the block Berlekamp–Massey algorithm . . . 58
  4.4.1 Matrix polynomial multiplications . . . 58
  4.4.2 Parallelization of Thomé's Berlekamp–Massey algorithm . . . 58
4.5 Implementation of XL . . . 59
  4.5.1 Reducing the computational cost of BW1 and BW3 . . . 60
  4.5.2 SIMD vector operations in $\mathbb{F}_{16}$ . . . 60
  4.5.3 Exploiting the structure of the Macaulay matrix . . . 62
  4.5.4 Macaulay matrix multiplication in XL . . . 64
  4.5.5 Parallel Macaulay matrix multiplication in XL . . . 66
4.6 Experimental results . . . 73
  4.6.1 Impact of the block size . . . 74
  4.6.2 Performance of the Macaulay matrix multiplication . . . 76
  4.6.3 Scalability experiments . . . 78

5 Parallel implementation of Wagner's birthday attack **81**
5.1 Wagner's generalized birthday attack . . . 82
  5.1.1 Wagner's tree algorithm . . . 82
  5.1.2 Wagner in storage-restricted environments . . . 83
5.2 The FSB hash function . . . 85
  5.2.1 Details of the FSB hash function . . . 85
  5.2.2 Attacking the compression function of FSB_{48} . . . 87
5.3 Attack strategy . . . 88
  5.3.1 How large is a list entry? . . . 88
  5.3.2 What list size can be handled? . . . 89
  5.3.3 The strategy . . . 89
5.4 Implementing the attack . . . 92
  5.4.1 Parallelization . . . 92
  5.4.2 Efficient implementation . . . 94
5.5 Results . . . 96
  5.5.1 Cost estimates . . . 96
  5.5.2 Cost measurements . . . 97
  5.5.3 Time-storage tradeoffs . . . 97
5.6 Scalability analysis . . . 98

Bibliography **101**

1

Introduction

Most of today's cryptographic primitives are based on computations that are hard to perform for a potential attacker but easy to perform for somebody who is in possession of some secret information, the key, that opens a back door in these hard computations and allows them to be solved in a small amount of time. Each cryptographic primitive should be designed such that the cost of an attack grows exponentially with the problem size, while the computations using the secret key only grow polynomially. To estimate the strength of a cryptographic primitive it is important to know how hard it is to perform the computation without knowledge of the secret back door and to get an understanding of how much money or time the attacker has to spend. Usually a cryptographic primitive allows the cryptographer to choose parameters that make an attack harder at the cost of making the computations using the secret key harder as well. Therefore designing a cryptographic primitive imposes the dilemma of choosing the parameters strong enough to resist an attack up to a certain cost while choosing them small enough to allow usage of the primitive in the real world, e.g. on small computing devices like smart phones.

Typically a cryptographic attack requires a tremendous amount of computation—otherwise the cryptographic primitive under attack can be considered broken. Given this tremendous amount of computation, it is likely that there are computations that can be performed in parallel. Therefore, parallel computing systems are a powerful tool for the attacker of a cryptographic system. In contrast to a legitimate user who typically exploits only a small or moderate amount of parallelism, an attacker is often able to launch an attack on a massively parallel system. In practice the amount of parallel computation power available to an attacker is limited only by the amount of money he is able to spend; however the amount of parallelism he can exploit depends on the particular attack which he is going to mount against a particular cryptographic primitive. The main challenge of implementing a cryptographic attack for a parallel computing system is to explore which parts of the computation can be computed in parallel and how this parallelism can be exploited most efficiently.

This thesis investigates three different attacks on particular cryptographic systems: Wagner's generalized birthday attack is applied to the compression function of the hash function FSB. Pollard's rho algorithm is used for attacking Certicom's ECC Challenge ECC2K-130. The implementation of the XL algorithm has not been specialized for an attack on a specific cryptographic primitive but can be used for attacking certain cryptographic primitives by solving multivariate quadratic systems. All three attacks are general attacks, i.e. they apply to various cryptographic systems; therefore the implementations of Wagner's generalized birthday attack and Pollard's rho algorithm can be adapted for attacking other primitives than those given in this thesis. The three attacks have been implemented on parallel architectures, each attack on a different architecture. The results give an estimate of the scalability and cost of these attacks on parallel systems.

Overview

Chapter 2 gives an introduction to parallel computing. The work on Pollard's rho algorithm described in Chapter 3 is part of a large research collaboration with several research groups; the computations are embarrassingly parallel and are executed in a distributed fashion in several facilities on a variety of parallel system architectures with almost negligible communication cost. This dissertation presents implementations of the iteration function of Pollard's rho algorithm on Graphics Processing Units and on the Cell Broadband Engine. Chapter 4 describes how XL has been parallelized using the block Wiedemann algorithm on a NUMA system using OpenMP and on an InfiniBand cluster using MPI. Wagner's generalized birthday attack is described in Chapter 5; the attack has been performed on a distributed system of 8 multi-core nodes connected by an Ethernet network.

2

Overview of parallel computing

Parallelism is an essential feature of the universe itself; as soon as there exists more than one entity and there is interaction between these entities, some kind of parallel process is taking place. This is the case for quantum mechanics as well as astrophysics, in chemistry and biology, in botanics and zoology, and human interaction and society. Therefore, exploiting parallelism also in information technology is natural and has started already at the beginning of modern computing.

The main condition for parallel computing is that there are parts of the computation which can be executed in parallel. This is the case if there are at least two parts of the computation which have no data dependencies, i.e., the input data of each part is not depending on the result of the computation of the other part. In [LS08], Lin and Snyder classify parallelism into two types as follows:

- In case the same sequence of instructions (e.g., the same mathematical function) is applied to a set of independent data items, the computation on these data items can be performed in any order and therefore, in particular, in parallel. This kind of parallelism is called *data parallelism.* The amount of parallelism scales with the number of independent data items.
- Some computations can be split in several tasks, i.e., independent instruction sequences which compute on independent data items. Since the tasks are independent from each other, they can be computed in parallel. This is called *task parallelism.* The number of independent tasks for a given computation is fixed and does not scale for a larger input.

Given that a program exhibits some parallelism, this parallelism can be exploited on a parallel computer architecture. In [Fly66] Flynn classifies computer architectures with respect to their ability to exploit parallelism by looking at the *instruction stream*, the sequence of instructions that are executed by a computer, and the *data stream*, the sequence of data that is processed by the instruction stream. Using these two terms, Flynn defines the following categories of parallel computer architectures:

- *Single Instruction Stream – Single Data Stream (SISD):* The computer follows only one instruction stream that operates on data from one single data stream.
- *Single Instruction Stream – Multiple Data Streams (SIMD):* Each instruction of the instruction stream is executed on several data streams.
- *Multiple Instruction Streams – Single Data Stream (MISD):* Several instructions from different instruction streams are executed on the same single data stream.
- *Multiple Instruction Streams – Multiple Data Streams (MIMD):* Instructions from different instruction streams are applied to independent data streams.

Data parallelism can naturally be exploited on SIMD architectures. By duplicating the instruction stream it also fits to MIMD architectures. Task parallel applications are usually suitable for MIMD architectures.

MISD architectures are merely stated for theoretical completeness. Nevertheless, one may argue that MISD architectures are used, e.g., for fault detection (although in this case the same instruction stream would be executed multiple times on the same data stream) or in pipelining systems (although one may argue that the data changes after each instruction).

Many applications exhibit data parallelism as well as task parallelism. Therefore, many computer systems incorporate SIMD and MIMD techniques and are able to perform mixed data and task parallel workloads efficiently.

The reason for spending effort in exploiting parallelism is to gain some *speedup* in the execution. For a given task, the speedup S is defined as the ratio between the time t_s of execution for the sequential, non-parallelized implementation and the time t_p of execution in parallel:

$$S = \frac{t_s}{t_p}.$$

The optimal speedup on n execution units therefore is

$$S_{\text{optimal}} = \frac{t_s}{\frac{t_s}{n}} = n.$$

However, when running a program on a computer, not all instructions may be able to be executed in parallel due to dependencies. The improvement one can gain from parallelism in terms of speedup is restricted according to *Amdahl's law* (as proposed by Amdahl in [Amd67]): let p be the part of the program that can be executed in parallel and s the part that must be executed sequentially, i.e., $p + s = 1$; the best speedup possible when using n execution units is

$$S = \frac{1}{s + \frac{p}{n}}.$$

Consequently, a high gain in speedup can only be reached if the sequential part of a program is sufficiently small. In case s is equal or extremely close to zero the program is called *embarrassingly parallel.* If s is too large, adding more and more execution units does not give noticeable benefit.

Amdahl's law seems to suggest that it is not worth to split workload over hundreds or even thousands of execution units if the sequential part is not negligible. However, that is not necessarily the case. Amdahl assumes the problem size to be fixed. In [Gus88] Gustafson points out that for many scientific applications a higher number of execution units gives the opportunity to increase the problem size and thus to get more accurate results. When scaling the problem size, for many applications the sequential part remains almost the same while the parallel part increases. In case the workload scales linearly with the number of execution units, an estimate of the speedup can be obtained from Gustafson's law [Gus88] as

$$S = \frac{s + np}{s + p} = s + np = n + s(1 - n).$$

In cryptanalytic applications the problem size is usually fixed, for example an attack on an encryption using a certain encryption standard with a certain parameter set. Nevertheless, the problem size and therefore the workload naturally is very large. Thus, cryptanalytic applications are good candidates for parallelization—given that they exhibit a sufficient amount of parallelism.

2.1 Parallel architectures

There are many types of parallel architectures, which mainly differ in the physical distance between the execution units. The distance gives a lower bound on the time that is required to deliver information from one unit to another. Therefore it has a big impact on how data is exchanged between the execution units and how communication is accomplished. This section introduces parallel architectures on the level of the microarchitecture, the instruction set, and the system architecture.

2.1.1 Microarchitecture

The microarchitecture of a processor is the lowest level on which computations can be executed in parallel. The microarchitecture focuses on the computation of a single instruction stream. If a sequential instruction stream contains one or more instructions that operate on different data items, i.e., if each input does not depend on the output of another one, these instructions can be executed in any order and therefore in parallel. This kind of parallelism is called *instruction level parallelism.*

Instruction pipelining

The execution of any instruction can be split into several stages [HP07, Appendix A]: for example it is split into instruction fetch, instruction decode, execution, memory access, and write back of results to the output register. For different instructions the execution stage might take a different amount of time depending on the complexity of the instruction: for example an `XOR`-instruction typically takes less time than a double-precision floating-point multiplication.

If all these stages were executed in one single clock cycle of a fixed length, much time would be wasted on the faster instructions since the processor would need to wait as long as the slowest instruction needs to finish. Furthermore, while an instruction is processed, e.g., by the execution unit all other stages would be idle.

Instead, the processor frequency and instruction throughput can be increased by pipelining the different stages of the instruction execution and by splitting the execution stage in several clock cycles depending on instruction complexity. After the first instruction has been loaded, it is forwarded to the decoder which can decode the instruction in the next cycle. At the same time, the second instruction is loaded. In the third cycle the first instruction starts to be executed while the second one is decoded and a third one is loaded in parallel.

This requires all instructions which are in the pipeline at the same time to be independent from each other; in case the instructions lack instruction level parallelism, parts of the pipeline have to stall until all dependencies have been resolved.

There are two ways to achieve a high throughput of instructions: Either the instructions are scheduled by a compiler or by the programmer in a way that minimizes instruction dependencies on consecutive instructions in a radius depending on the pipeline length. Or the fetch unit looks ahead in the instruction stream and chooses an instruction for execution which is independent from the instructions that are currently in the pipeline; this is called *out-of-order execution*. Details on out-of-order execution can be found in [HP07, Section 2.4].

The physical distance between the pipeline stages is very small and the communication time between the stages can almost be neglected. From the outside a pipelined processor appears to be a SISD architecture; internally the pipeline stages actually resemble a MIMD architecture.

Superscalar processors

Another way to exploit instruction level parallelism is to use several *arithmetic logic units* (ALUs) to compute an instruction stream. An instruction scheduler forwards several independent instructions to several ALUs in the same clock cycle. The ALUs may share one common instruction set or they may have different instruction sets, e.g., a processor may have two integer units and one floating point unit. As in the case of pipelining, the instruction scheduler of a superscalar processor may examine several upcoming instructions out of order to find independent instructions.

The ALUs communicate directly through registers without communication costs; after the output of one instruction has been computed it is usually available for all ALUs in the following cycle. Similar to pipelining architectures, superscalar processors appear to be SISD architectures but internally they operate in an MIMD fashion.

Instruction pipelining as well as superscalar execution is very common and used in most of today's microarchitectures like the x86 and AMD64 processors from Intel and AMD, IBM's PowerPC processors and so on. Both approaches are often combined: several stages of the instruction pipeline can be duplicated to allow parallel execution of independent instructions.

The drawback of these architectures is that they lose performance if there is not enough instruction level parallelism available or found during compilation or execution. In particular programs that have many conditions or branches suffer from these techniques. Furthermore, the logic for instruction scheduling and dependency detection consumes resources which are not available for execution units any more.

Another type of parallelism that can be exploited by a microarchitecture is *bit level parallelism*: Processors handle data in units of a certain number of bits, the *word size*. By increasing the word size a processor is able to compute on more data in parallel: a processor of a word size of 64 bits can compute an addition of two 64-bit numbers using one instruction, a 32-bit processor would need at least two instructions plus possibly more instructions to handle overflows. For example the x86 processor architecture started with a word size of 16 bits in 1978, was extended to 32 bits in the mid eighties, and eventually supported a word size of 64 bits with the introduction of the Pentium 4F processor in 2004.

2.1.2 Instruction set

Microarchitectures mainly aim at the exploitation of instruction level parallelism. However, the instruction set of a single processor may also offer parallelism which can be exploited either by the programmer or by a compiler.

Vector processors

Vector processors implement the SIMD architecture: the same instruction is executed on independent data elements (vector elements) from several data streams in parallel. The elements are loaded into vector registers which consist of several slots, one for each data element. Vector instructions, often called SIMD instructions, operate on the vector registers.

The number of execution units does not necessarily need to be equal to the number of vector elements. The processor may operate only on parts of the vector registers in each clock cycle. Usually vector processors only have 4 (e.g., NEC SX/8) to 8 (e.g., Cray SV1) ALUs (see [HP07, Appendix F]). Nevertheless, operating on a large vector in several steps on groups of the vector register slots allows the processor to hide instruction and memory latencies: since it takes several cycles before the execution on the last group of register slots has started, the operation on the first group is likely to be finished so that all dependencies that might exist for the next instruction are resolved.

Initially, a very high number of vector elements was envisioned, for example as much as 256 elements for the early prototype architecture Illiac IV (which eventually was built with only 64 vector elements) [BDM$^+$72]. The Cray-1 had 8 vector registers of 64 vector elements each as well. Today, many specialized scientific computers also offer wide vector registers of 64 to 256 vector elements [HP07, Figure F.2].

Apart from these cases also short-vector architectures are very common. Most of today's x86-processors have 128-bit vector registers, which can be used for 8-, 16-, 32-, or 64-bit vector operations on sixteen, eight, four, or two vector elements. The Synergistic Processing Units of the Cell processor have a similar design and the PowerPC processors also provide 128-bit vector registers. Note that 128-bit vector registers can also be used for 1-bit operations, i.e. logical operations, on 128 vector elements. This is exploited by a technique called bitslicing; see Chapter 3 for an application of this technique.

Very Long Instruction Word processors

Superscalar processors exploit instruction level parallelism by choosing several data-independent instructions from the instruction stream for execution. The logic for this run-time instruction scheduling is relatively complex and consumes extra resources. But the scheduling of instructions can also be done off-line: the compiler or the programmer schedules independent instructions into instruction groups that can be forwarded by the fetch unit at once to several ALUs. Therefore, the ALUs can be kept busy without the need for dedicated scheduling hardware. This architecture is called *Very Long Instruction Word (VLIW).*

VLIW processors are not easily classified by Flynn's taxonomy. On a large scale, a single instruction stream operates on a single data stream. However, due to the fact that VLIW processors contain several ALUs they may also be categorized as MIMD or even MISD architectures. VLIW processors were first mentioned by Fisher in 1983 [Fis83] but have not been widely deployed before the IA-64 architecture by HP and Intel and its implementation as Intel Itanium processor starting in 2001 (see [HP07, Appendix G]). The most recent, widespread VLIW processors are the graphics processing units of AMD.

2.1.3 System architecture

On a large scale, a parallel architecture can be designed by connecting several execution units by a variety of network interconnects and/or system buses. The following paragraphs give a short introduction to several parallel system architectures.

Symmetric Multiprocessing

Several execution units can be connected by a system bus. If all execution units have the same architecture, this system architecture is called *Symmetric Multiprocessing* (SMP). All execution units of an SMP architecture share a joint memory space.

Multi-core processors: A *multi-core processor* consists of several execution units called *cores* which compute in an MIMD fashion. Several resources might be shared, such as (parts of) the cache hierarchy as well as IO and memory ports. The operating system has access to all cores and schedules processes and threads onto the cores. Communication between the cores can be accomplished via shared cache or via main memory.

Almost all of today's high-end CPUs are multi-core CPUs. Multi-core processors are used in a large range of devices: servers, desktop PCs, even embedded systems and mobile phones are powered by multi-core processors. Common configurations range from dual-core CPUs to ten- and even twelve-core CPUs (in May 2011). Core count of mainstream processors is envisioned to raise even higher, even though up to now only a small fraction of desktop applications benefit from multi-core architectures.

Non-Uniform Memory Access: If several conventional processors are placed on one mainboard and are connected by a system bus, each processor has a memory controller of its own even though all processors have full access to all memory. Since latency as well as throughput may vary depending on which physical memory is accessed, this architecture is called *Non-Uniform Memory Access* (NUMA) architecture as opposed to *Uniform Memory Access* (UMA) architectures (for example multi-core processors) where accessing any physical memory address has the same performance. Several multi-core processors may be used to set up a NUMA system.

Since the mainboards for NUMA systems are more expensive than off-the-shelf boards, NUMA systems are usually used for commercial or scientific workloads. The communication distance between the execution units is higher than in the case of multi-core processors since caches are not shared and all communication must be accomplished via main memory.

Simultaneous multithreading: In general a lack of instruction level parallelism on pipelined or superscalar processors as well as high-latency instructions (like memory access or IO) may lead to stalls of one or several ALUs. This can be compensated for by a technique called *simultaneous multithreading* (SMT): A processor with an SMT architecture concurrently computes two or more instruction streams, in this case called processes or threads. Therefore the instruction scheduler has more options to choose an instruction from one of the independent instruction streams. For concurrent execution, each instruction stream requires his own instance of resources like the register file. In contrast to multi-core processors, simultaneous multithreading is not really a parallel architecture even though it appears to be a MIMD architecture from the outside. Actually, it just allows a higher exploitation of the computation resources of a single processor.

Since each thread has its own register file, communication between the instruction streams is accomplished via memory access. In the best case, shared data can be found in on-chip caches but communication might need to be channeled via main memory.

SMT is state of the art in many of today's processors. Usually, two concurrent threads are offered. Intel calls its SMT implementation *Hyper-Threading Technology* (HTT) and uses it in most of its products. AMD, IBM and others use SMT as well.

Accelerators

Using *accelerators* in contrast to SMP architectures leads to a heterogeneous system design. Accelerators can be any kind of computing device in addition to a classical processor. Either the accelerators are connected to the processor via the system bus as for example in the case of Field-Programmable Gate Array (FPGA) cards and graphics cards or they can be integrated onto the die of the processor like the Synergistic Processing Elements of the Cell processor, Intel's E600C Atom chips with integrated FPGAs, or the Fusion processors of AMD, which include Graphics Processing Units (GPUs); there is a strong trend for moving accelerators from the system bus into the processor chip.

Parallel computing on FPGAs is not part of this thesis and therefore will not be further examined; GPUs will be described in more detail in Sections 2.3 and 3.4.1, the Cell processor in Section 3.3.1.

Supercomputers

The term *supercomputer* traditionally describes a parallel architecture of tightly coupled processors where the term *tight* depends on the current state of the art. Many features of past supercomputers appear in present desktop computers, for example multi-core and vector processors. Today's state of the art supercomputers contain a large number of proprietary technologies that have been developed exclusively for this application. Even though supercomputers are seen as a single system, they fill several cabinets or even floors due to their tremendous number of execution units.

Since supercomputers are highly specialized and optimized, they have a higher performance compared to mainstream systems. On the TOP500 list of the most powerful computer systems from November 2011 only 17.8% of the machines are supercomputers but they contribute 32.16% of the computing power. The machines of rank one and two are supercomputers. Even though the number of supercomputers in the TOP500 list has been declining of the past decade, they still are the most powerful architectures for suitable applications. The TOP500 list is published at [MDS$^+$11].

Supercomputers are MIMD architectures even though they commonly are treated as *Single Program Multiple Data (SPMD)* machines (imitating Flynn's taxonomy): the same program is executed on all processors, each processor handles a different fraction of the data, and communication is accomplished via a high-speed interconnect. In general, supercomputers provide a large number of execution units and fast communication networks, which make them suitable for big problem instances of scientific computations.

Cluster computing

Off-the-shelf desktop computers or high-end workstations can be connected by a high-speed interconnect to form a *computer cluster*. Classical high-speed interconnects are InfiniBand and Myrinet; Gigabit-Ethernet is becoming more and more widespread as well. Several clusters can be coupled by a wide area network to form a *meta-cluster*. The connected computers of a cluster are called *nodes*.

Clusters differ from supercomputers by providing lower communication bandwidth and higher latencies. Similar to supercomputers they provide a homogeneous computing environment in terms of processor architecture, memory amount per node, and operating system.

On the TOP500 list from November 2011 mentioned above, 82% of the machines are cluster systems. They contribute 67.76% of the total computing power on the list. Ten years earlier, these numbers were both below 10%. This shows that cluster systems are more and more widespread. Reasons for their popularity are that cluster systems are relatively cheap and easy to maintain compared to supercomputers while they are nonetheless suitable for scientific computing demands.

Clusters are MIMD architectures. Nevertheless, they are usually programmed in a SPMD fashion in the same way as supercomputers. Clusters are used for data-parallel applications that have a suitable demand for performance in computation and communication.

Distributed computing

If the communication distance is very high, usually the term *distributed computing* is used. This term applies if no guarantee for communication latency or throughput is given. This architecture consists of a number of nodes which are connected by a network. The nodes can be desktop computers or conventional server machines. The Internet or classical Ethernet networks may be used as interconnecting network.

In general, the physical location of the nodes is entirely arbitrary; the nodes may be distributed all over the world. SETI@home [KWA$^+$01] was one of the first scientific projects for distributed computing. Its grid computing middleware "BOINC" [And04] is used for many different scientific workloads today.

If the location of the nodes is more restricted, e.g., if the desktop computers of a company are used for parallel computing in the background of daily workload or exclusively at night, the term *office grid* is used. Some companies like Amazon offer distributed computing as *cloud services*.

Distributed computing is a MIMD architecture. It is suitable if the demand for communication is small or if high network latency and low throughput can be compensated or tolerated. This is the case for many data parallel applications. Usually computing resources are used which otherwise would remain unexploited; in case of cloud computing resources can be acquired on demand.

2.2 Parallel programming

The previous section introduced parallel architectures on three different levels of abstraction: the microarchitecture, the instruction set, and the system architecture.

On the level of the microarchitecture, the only way to influence how instructions are executed in parallel is the order in which the instructions are placed in the machine code of the program. For programming languages like C, C++, or Fortran, the order is defined by the compiler; nowadays all optimizing compilers have some heuristics that try to produce a reasonable scheduling of instructions for the requested microarchitecture. If more control over instruction scheduling is needed, e.g. in case the compiler fails to produce efficient machine code, the program (or parts of the program) can be written in assembly language.

Programming in assembly language also gives full control over program execution on the level of the instruction set. In general compilers do not give direct access to the instruction set; however, some compilers allow the programmer to embed assembly code into the program code for this purpose. Compilers for VLIW architectures handle data flow and instruction scheduling internally without direct control of the programmer. Vectorizing compilers can parallelize sequential source code for vector processors in certain cases in an automated fashion. Some programming languages offer SIMD vector operations as part of the language.

On the level of the system architecture, the program is executed by several computing units. The workload of the computation is distributed over these units. Cooperation between the computing units is achieved by communicating intermediate results. The following paragraphs discuss two concepts for communication between the computing units of a parallel system: Either the data is exchanged between computing units implicitly via *shared memory* or explicitly via *message passing*.

2.2.1 Shared memory

Shared memory works like a bulletin board in the lobby of a student dorm: The students place messages on the board. Messages can be read, modified or removed by all students. The location of the board is well known to all students. Students can access the board at any time in any order. Communication over a bulletin board is most efficient if the students have some agreement about what information is placed on what position on the board and who is allowed to remove or modify the information.

From a programming perspective, communication is accomplished by making a certain memory region available to all instances of a program which are executed in parallel. All instances of the program write data to or read data from this memory region. Shared memory programming is available on most parallel system architectures listed in the previous section. However, it fits most naturally to SMP architectures with a joint memory space.

A popular standard for shared-memory programming on SMP architectures is OpenMP (Open Multi-Processing, see [OMP]). OpenMP defines a set of compiler directives for C, C++, and Fortran compilers. These directives instruct the compiler to distribute the workload of the specified code regions over several threads. Since all threads are executed in the same address space, they have full access to all data of the parent process. The compiler automatically inserts code for starting, managing, and stopping of threads. Using OpenMP, sequential code can be parallelized relatively easily by inserting compiler directives into sequential source code.

If OpenMP is not available or does not fit the programming model, the programmer can control the parallel execution on an SMP architecture directly by using, e.g., POSIX (Portable Operating System Interface) threads instead of OpenMP. POSIX threads are in particular useful for implementing task-parallel applications. Shared-memory communication between separate processes is carried out by mapping a joint shared memory segment into the address space of each process. An emerging programming standard for general-purpose computing on GPUs is the C-language derivative OpenCL (Open Computing Language) which also can be used for programming of hybrid SMP and GPU computing systems. See Section 2.3 for details on general-purpose GPU programming and OpenCL.

Shared-memory programming brings the risk of race conditions: Consider a segment of program code where a value in shared memory is incremented by two threads A and B. Each thread first reads the value from the memory location to a register, increments the value, and writes it back to the old memory location. Everything is fine if A and B pass this entire code section one at a time. If these three instructions are processed by both threads at the same time, the resulting value in the shared memory location is undefined: Assume thread A is the first one to read the value, but thread B reads the same value before it is modified and stored by A. After both threads have passed the code segment, the value in shared memory has been incremented by only one instead of two.

To make the code segment work, one must ensure that it is processed *atomically*, i.e. the code segment must only be entered by a thread while no other thread is currently executing it. The most efficient way to implement this particular example is to use processor instructions that perform the incrementation as an atomic operation—if such an instruction is provided by the instruction set and is accessible through the programming language. Many instruction sets offer several flavors of atomic operations. Access to large critical code sections can be controlled with semaphores, locks, mutexes, or any other control mechanism that is offered by the programming environment or the operating system. For shared-memory programming, the programmer is obliged to provide access control to critical code sections and to guarantee deterministic program behaviour.

2.2.2 Message passing

Message passing is similar to sending a letter by mail: When the sender finished writing the letter, it is picked up by a postman and delivered to the receiver. After the letter has been handed over to the postman, the sender has no access to the letter anymore. The receiver checks his mailbox for incoming mail. At least the address of the receiver needs to be known for successful delivery of the letter.

The message-passing programming paradigm is usually implemented as a library that offers an interface of function calls which can be invoked by the communicating processes. Typically pairs of functions are offered: one for sending, one for receiving a message. Sending and receiving can either be blocking (the sender waits until the postman picks up the letter, the receiver waits until the postman delivers the letter) or non-blocking (the postman picks up and drops off the letter at the mail boxes of sender and receiver while those two are free to continue with their work).

Message passing is available for tightly coupled SMP architectures and for more loosely coupled architectures like clusters; message passing fits most naturally to the latter one. Message passing can compensate for the disadvantages of remote memory access on NUMA architectures for some applications: One process is executed on each NUMA node and message passing is used for the communication between the NUMA nodes.

The most common message-passing interface for high-performance computing is the MPI (Message Passing Interface) standard. This standard defines syntax and semantics of a message-passing interface for the programming languages C and Fortran. Popular implementations of the MPI standard are Open MPI available at [OMPI] and MPICH2 available at [ANL]. Most MPI implementations provide command-line tools and service programs that facilitate starting and controlling MPI processes on large clusters. The performance of communication-intensive applications depends heavily on the capability of the MPI implementation to offer efficient communication on the targeted parallel system.

Communication channels such as pipes or sockets provided by the operating system are more general approaches than MPI. They are commonly used for communication on distributed systems but less commonly on more tightly coupled systems such as clusters or SMP architectures. The operation mode of sockets is particularly useful for applications with a varying number of computing nodes. MPI requires the number of nodes to be fixed over the whole execution time of the application; failing nodes cause the whole computation to be aborted.

2.2.3 Summary

Neither of the two approaches is more expressive than the other: An implementation of the MPI standard might use shared-memory programming for message delivery on an SMP architecture; a distributed shared-memory programming language like Unified Parallel C (UPC, see [LBL]) might use message-passing primitives to keep shared-memory segments coherent on cluster architectures.

The choice between these paradigms depends on the application and its targeted system architecture. The paradigms are not mutually exclusive; hybrid solutions are possible and common on, for example, clusters of SMP architectures. In this case one process (or several processes on NUMA architectures) with several threads is executed on each cluster node. Local communication between the threads is accomplished via shared memory, remote communication between the cluster nodes (and NUMA nodes) via message passing.

2.3 General-purpose GPU programming

Graphics Processing Units (GPUs) are computing devices which are specialized and highly optimized for rendering 3D graphic scenes at high frame rates. Since the 1990s graphics cards have become a mass-market gaming device. Market competition led to a rapid development of computational power alongside a tremendous drop of prices. State-of-the-art GPUs have a single-precision floating-point peak performance in the range of teraflops at relatively small power consumption and moderate prices. Therefore, graphics cards have become more and more attractive for High Performance Computing.

Already in the early '90s the computational power of GPUs has been used for other applications than computer graphics (e.g. [LRD$^+$90]). In 2003, Mark Harris established the acronym GPGPU for the general-purpose use of graphics cards [Har03]. In the beginning, it was very tedious to program GPUs. The algorithm had to be expressed with the shader instructions of the rendering pipeline. General-purpose programmability was greatly increased when NVIDIA introduced a unified shader hardware architecture with the GeForce 8 Series graphics cards in November 2006. NVIDIA made an essential change in the hardware architecture: Instead of implementing several different types of graphic shaders in hardware, they developed a massively parallel multi-core architecture and implemented all shaders in software. NVIDIA's architecture is called Compute Unified Device Architecture (CUDA). In 2007, AMD also released a GPU with a unified shader architecture, the Radeon R600 series.

AMD and NVIDIA both offer software development kits (SDKs) that give access to the GPUs from a host program running on the CPU. The host program controls the execution of a shader program, the so called kernel. NVIDIA's SDK is called "CUDA SDK"; the SDK of AMD was first named "ATI Stream SDK". Nowadays it is promoted as "AMD Accelerated Parallel Processing (APP) SDK". Initially, both vendors used different C-like programming languages, NVIDIA offered a language called CUDA-C, AMD used Brook+, an enhanced version of BrookGPU (see [BFH$^+$04]). Today both vendors support the execution of programs written in OpenCL. OpenCL is a specification of a programming framework consisting of a programming language and an API. It facilitates writing and executing parallel programs in heterogeneous computing environments consisting of e.g. multi-core CPUs and GPUs. The OpenCL specification can be found at [Mun11].

Even though GPUs have been used successfully to accelerate scientific algorithms and workloads (e.g. [BGB10; RRB$^+$08]), their performance can only be fully exploited if the program is carefully optimized for the target hardware. There are also reports of cases where GPUs do not deliver the expected performance (e.g. [BBR10]). Apart from the fact that not all applications are suitable for an efficient implementation on GPUs, there may be several reasons for an underutilization of these computing devices, e.g. poor exploitation of data locality, bad register allocation, or insufficient hit rates in the instruction cache. In particular the latter two cases cannot easily, maybe not at all, be prevented when using a high-level programming language: If the compiler does not deliver a suitable solution there is no chance to improve the performance without circumventing the compiler.

To solve this issue, Bernstein, Schwabe, and I constructed a toolbox for implementing GPU programs in a directly translated assembly language (see Section 2.3.2). The current solution works only with NVIDIA's first generation CUDA hardware. Due to NVIDIA's and AMD's lack of public documentation the support for state-of-the-art GPUs still is work in progress.

Since 2006 NVIDIA released several versions of CUDA. Furthermore, the processor architecture has been modified several times. The architectures are classified by their respective *compute capability*. The first generation of CUDA GPUs has the compute capabilities 1.0 to 1.3 and was distributed until 2010. In 2010, NVIDIA introduced the second generation of CUDA called Fermi. Up to now Fermi cards have the compute capabilities 2.0 and 2.1. Release steps within each generation introduce minor additions to the instruction set and minor extensions to the hardware architecture. The step from 1.3 to 2.0 introduced a completely redesigned hardware architecture and instruction set.

Because CUDA allows major changes of the hardware implementation, CUDA software is usually not distributed in binary form. Instead, the device driver receives the CUDA-C or OpenCL source code or an instruction-set-independent intermediate assembly code called Parallel Thread Execution (PTX). In either case the source code is compiled for the actual hardware and then transferred to the GPU for execution.

NVIDIA does not offer an assembler for CUDA; PTX-code is compiled like a high level language and therefore does not give direct access to instruction scheduling, register allocation, register spills, or even the instruction set. Nevertheless, Wladimir J. van der Laan reverse-engineered the byte code of the first generation CUDA 1.x instruction set and implemented an assembler and disassembler in Python. These tools are available online at [Laa07] as part of the *Cubin Utilities*, also known as *decuda*.

In April 2011, Yun-Qing Hou started a similar project called *asfermi* in order to provide an assembler and disassembler for the instruction set of CUDA 2.x Fermi. Hou's code is available at [Hou11]. Since this tool set does not yet fully support all opcodes of the Fermi architecture, the remainder of this section will focus on decuda and therefore compute capability 1.x and CUDA SDK version 2.3. Support for OpenCL has been introduced with CUDA SDK version 4.0 and will not be described in this section.

2.3.1 Programming NVIDIA GPUs

Today's NVIDIA GPUs are SIMD architectures. They have been designed for data parallel applications—e.g. graphics processing. Each independent item of a data stream is assigned to a thread; many threads are executed in parallel on the execution units of the GPU. A certain number of threads (compute capability 1.x: 32 threads; compute capability 2.x: 64 threads) are executed in lock-step, i.e. a single instruction is fetched and executed by all of these threads. Such a group of threads is called a thread *warp*. Threads in a warp are allowed to take different branches but this results in a sequential execution of the branches that are taken by subsets of threads and therefore in a loss of efficiency. Several warps can be organized in a *thread block*. All threads in a thread block can exchange data via a fast data storage called *shared memory*. Several thread blocks can be executed concurrently on one GPU core and in parallel on different cores.

Each core has a large number of registers and a relatively small amount of shared memory, e.g. on compute capability 1.3 devices each core has 16384 registers and 16 KB shared memory. A set of registers is assigned to each thread and a fraction of shared memory to each thread block for the whole execution of the thread block. Therefore, the maximum number of threads that can be executed concurrently on one core depends on the number of registers each thread demands and the amount of shared memory each thread block requests.

Since each core of compute capability 1.x GPUs has 8 ALUs, one warp of 32 threads seems to be more than enough to keep all ALUs busy. Nevertheless, experiments showed that one single warp is scheduled only every second instruction issue cycle (which is four times longer than one ALU clock cycle), leaving the ALUs idle during one issue cycle in between. Consequently at least two warps should be executed per core. Furthermore, the instruction set can only address 128 registers, so to fully utilize all registers at least 128 threads (four warps) must be executed concurrently. In addition, instruction latencies and memory latencies lead to the recommendation of running at least 192 threads concurrently to get a high ALU utilization [NVI09a].

The threads have access to off-chip device memory, also called *global memory*; this memory is not cached close to the cores. Furthermore, there is a *constant memory* cache for read-only data access. The cores also have access to a texture memory which is not further investigated in this thesis. Communication between thread blocks is only possible via global memory. Compute capabilities 1.1 and above offer atomic operations on global memory for synchronization between thread blocks.

As stated above, NVIDIA graphics cards can be programmed with the help of the CUDA-SDK. The goal of CUDA is that programmers do not need to rewrite their whole application to profit from the computational power of GPUs. Instead, computationally intensive parts from an existing program are identified and delegated to the GPU. It is possible to mix host code and device code in the source code. To distinguish between the two different execution domains, NVIDIA introduced a small set of additional keywords to the C programming language [NVI09b].

Three *Function Type Qualifiers* specify in which context a function should be executed: A `__host__` function can only be called from and executed on the host. This is the standard qualifier, therefore it can be omitted. A `__global__` function is executed on the device and can be called from the host. Functions which are executed on the device and can only be called from the device are specified as `__device__` functions.

The memory location of data is defined by three *Variable Type Qualifiers.* Variables which are defined with the keyword `__device__` are located on the device. The additional keyword `__constant__` specifies data items that are not changed during kernel execution and can be cached in the constant data cache of the GPU cores. Variables that should reside in shared memory are defined as `__shared__`. Similar specifiers are available for PTX code to qualify global or shared memory accesses. The CUDA API offers functions to move data between host and device memory.

The CUDA-C compiler is called *nvcc.* It slightly modifies the host code to be compliant to standard C and passes this code to the default C compiler, e.g. gcc. The kernel code is extracted and compiled to PTX-code and machine-code if requested.

When the kernel is invoked through the CUDA API during program execution, the CUDA runtime tries to locate a kernel binary that fits to the actual GPU hardware. If no binary is available, either CUDA-C or PTX source code is used to compile the kernel. The binary can afterwards be stored in a code repository for future usage to avoid recompilation at each invocation.

Experiments showed that NVIDIA's compiler nvcc is not suitable for scientific workloads. In particular the register allocator of nvcc produced inefficient binary code for cryptanalytic kernels. The allocator seems to be optimized for small kernels (since most shader kernels are relatively small) and does not perform well on kernels that have several thousand lines of code. In some cases the compiler ran for several hours, eventually crashing when the system ran out of memory. When the compiler succeeded to produce assembly code, the program required a high number of registers; register spills occurred frequently. Since spilled registers are not cached close to the core but written to and read from global memory, this causes high instruction latencies and therefore often results in performance penalties.

These issues can be solved by writing cryptographic computing kernels by hand in assembly language using van der Laans assembler cudasm. The following section describes the toolkit *qashm-cudasm* that helps to implement and maintain kernels of several thousand lines of assembly code.

2.3.2 Assembly for NVIDIA GPUs

The key to programming kernels for NVIDIA GPUs in assembly language is the assembler cudasm available at [Laa07]. It translates a kernel written in assembler into a text file containing the byte code in a hexadecimal representation. The assembly language was defined by van der Laan to use the same mnemonics and a similar syntax as the PTX code of NVIDIA whenever possible.

The output file of cudasm can be put into the code repository of the CUDA application. The CUDA runtime transfers it to the graphics card for execution the same way as a kernel that was compiled from CUDA-C or PTX. The following tweaks and compiler flags are necessary to make the CUDA runtime digest the cudasm-kernel:

First a host application is written in CUDA-C. This application allocates device memory, prepares and transfers data to the graphics card and finally invokes the kernel as defined by the CUDA API. The device kernel can be implemented in CUDA-C for testing purposes; if this is omitted a function stub for the kernel must be provided. Eventually, the program is compiled using the following command:

```
nvcc -arch=architecture -ext=all -dir=repository sourcefiles -o outfile
```

The desired target architecture can be chosen by the flag `arch`; e.g. compute capability 1.0 is requested by `arch=sm_10`, compute capability 1.3 by `arch=sm_13`. The flag `ext=all` instructs the compiler to create a code repository for both PTX and assembly code in the directory *repository*. For each run nvcc will create a separate subdirectory in the directory of the code repository; the name of the subdirectory seems to be a random string or a time based hash value. The runtime will choose the most recent kernel that fits to the actual hardware. Deleting all previous subdirectories when recompiling the host code facilitates to keep track of the most recent kernel version.

The PTX kernel will be placed inside the newly created subdirectory in the file `compute_`*architecture*. The binary code resides in the file `sm_`*architecture*. The latter one can simply be replaced by the binary that was created by cudasm. The runtime library will load this binary at the next kernel launch.

Assembly programming

Commercial software development requires a fast, programmer friendly, efficient, and cheap work flow. Source code is expected to be modular, reusable, and platform independent. These demands are usually achieved by using high-level programming languages like C, C++, Java, C#, or Objective-C. There are only a few reasons for programming in assembly language:

- high demands on performance: In many cases hand-optimizing critical code sections gives better performance than compiler-optimized code. This has several reasons; details can be found in [Sch11a, Section 2.7].
- the need for direct access to the instruction set: Some instructions like the AES instruction set [Gue10] might not be accessible from high level programming languages. Operating systems need access to special control instructions.
- the requirement for full control over instruction order: For example the program flow of cryptographic applications can leak secret information to an attacker [Koc96]. This can be avoided by carefully scheduling instructions by hand.

- a questionable understanding of "having fun": Some people claim that it is fun to program in assembly language.

Only the latter one might allow the programmer to actually enjoy programming in assembly language. Assembly programming is a complex and error prone task. To reduce the complexity of manually writing GPU assembly code, this section introduces several tools that tackle key issues of assembly code generation.

The major pitfall of writing assembly code is to keep track of what registers contain which data items during program execution. On the x86 architecture which provides only 8 general purpose registers, this is already troublesome. Keeping track of up to 128 registers on NVIDIA GPUs pushes this effort to another level. However, the advantage to choose what register contains which data at a certain time when running the program is not necessary to write highly efficient assembly code and therefore is usually not the reason for writing assembly code in the first place. As long as the programmer can be sure that a register allocator finds a perfect register allocation provided that it exists and otherwise reports to the programmer, register allocation can be delegated to a programming tool. However, this tool should not be NVIDIA's compiler nvcc—since it neither finds a perfect register allocation nor reports on its failure.

Furthermore, there is no common syntax for different assembly languages; even for the same instruction set the syntax may be different for assemblers of different vendors: e.g. "`addl %eax, -4(%rbp)`" and "`add DWORD PTR [rbp-4], eax`" are the same x86 instruction in gcc assembler and Intel assembler, respectively. The instructions vary in mnemonic, register and addressing syntax, and even in operand order. Therefore switching between architectures or compilers is error prone and complicated. Furthermore, most assembly languages have not been designed for human interaction and are not easily readable. This again is not an inherent problem of assembly programming itself but a shortcoming of commonly used assembly programming tools.

Both of these pitfalls, register allocation and syntax issues, are addressed by Daniel J. Bernstein's tool qhasm.

qhasm

The tool qhasm is available at [Ber07b]. It is neither an assembler itself nor has it been designed for writing a complete program; qhasm is intended to be used for replacing critical functions of a program with handwritten assembly language. The job of qhasm is to translate an assembly-like source code, so called *qhasm code*, into assembly code. Each line in qhasm code is translated to exactly one assembly instruction. The syntax of the qhasm code is user-defined. Furthermore, qhasm allows using an arbitrary number of named register variables instead of the fixed number of architecture registers in the qhasm code. These register variables are mapped to register names by the register allocator of qhasm. If qhasm does not find a mapping, it returns with an error message; the programmer is in charge of spilling registers to memory. The order of instructions is not modified by qhasm.

Therefore, qhasm gives as much control to the programmer as writing assembly directly while eliminating the major pitfalls.

The input for qhasm is the qhasm code consisting of variable declarations and operations and a *machine description file*. This file defines the syntax of the qhasm code by providing a mapping of legitimate qhasm operations to assembly instructions. Furthermore, it contains information about the calling convention and the register set of the target environment.

This design allows the programmer to easily adapt qhasm to new machine architectures by providing an appropriate machine description file. Furthermore, qhasm code for different architectures can follow a unified, user defined syntax. In theory qhasm source code can be reused on other architectures if the instruction sets offer instructions of identical semantics and if a consistent machine description file is provided; nevertheless, it is likely that parts of the qhasm code need to be rewritten or reordered to take architectural differences into account and to achieve full performance.

qhasm-cudasm

Van der Laan's assembler cudasm combined with Bernstein's register allocator qhasm (together with a machine description file that was written for cudasm) gives a powerful tool for programming NVIDIA graphics cards on assembly level and can be easily used for small kernels. Nevertheless, for large kernels of several hundred or even thousand lines of code additional tools are necessary to gain a reasonable level of usability for the following reasons:

NVIDIAs compiler nvcc does not support linking of binary code; usually all functions that are qualified with the keyword `__device__` in CUDA-C are inlined before compiling to binary code. Therefore it is not possible to replace only parts of a CUDA-C kernel with qhasm code—the whole GPU kernel must be implemented in qhasm code. But qhasm was designed to replace small computationally intensive functions. There is no scope for register variables: the names of register variables are available in the whole qhasm code. This makes it complicated to keep track of the data flow. Furthermore, qhasm does not support the programmer in maintaining a large code base e.g. by splitting the code into several files.

Therefore, a modified version of the *m5* macro processor is used on top of qhasm to simplify the handling of large kernels. The original version of m5 by William A. Ward, Jr. can be found at [War01]. The following list gives an overview of some native m5 features:

- includes: An m5 source file can include other m5 files; the content of the files is inlined in the line where the include occurs.

- functions: Functions can be defined and called in m5; "call" in this case means that the content of the called function is inlined at the position where the call occurs.

- expressions: Macro variables can be defined in m5 and expressions on macro variables and constant values can be evaluated.
- loops: Loops over macro variables result in an "unrolled" m5 output.

The m5 macro processor was extended to support scoping of register variables. Register variables which are defined in a local scope are automatically prefixed by m5 with a scope-specific string. This guarantees locality on the qhasm level.

The m5 macro processor completes the qhasm-cudasm development toolkit. Each kernel is processed on three levels: The programmer provides a set of source files with the file extension ".mq". Each mq-file contains qhasm code enriched with m5 macro instructions. These files are flattened by m5 to a single qhasm file with the file extension ".q". The qhasm code is fed to qhasm together with the machine description file. The output of qhasm is an assembly file with the file extension ".s". Eventually the assembly file is processed by cudasm, giving the final binary file with the file extension ".o". This file is used to replace the kernel stub that was created by nvcc in the code repository directory.

The solution of using m5 for preprocessing is not perfect; in particular the syntax for declaring and using local register variables is error-prone. Therefore in the future several functionalities of m5 will be integrated into qhasm, in particular function inlining and the declaration of local register variables.

Furthermore, future versions of qhasm-cudasm will support up-to-date graphics hardware of NVIDIA and AMD, using either the upcoming assembler implementations of the open-source community or by contributing assemblers ourselves. As mentioned before the open-source project asfermi aims at providing an assembler for NVIDIA's Fermi architecture. Ádám Rák started to implement an assembler for AMD graphics cards as open-source software (available at [Rák11]). Furthermore, an initial version of an assembler for AMD graphics cards is provided on my website at [Nie11]. Since AMD uses a VLIW architecture, additional tools or extensions to qhasm are necessary for programming AMD graphics cards in assembly language.

The combination of qhasm and cudasm together with m5 was used for programming a large cryptographic kernel of over 5000 lines of code. This project is described in Chapter 3. Furthermore, various kernels for benchmarking NVIDIAs GeForce 200 Series GPUs were implemented using these tools.

3

Parallel implementation of Pollard's rho method

The elliptic-curve discrete-logarithm problem (ECDLP) is the number-theoretic problem behind elliptic-curve cryptography (ECC): the problem of computing a user's ECC secret key from his public key. Pollard's rho method solves this problem in $O(\sqrt{\ell})$ iterations, where ℓ is the largest prime divisor of the order of the base point. A parallel version of the algorithm due to van Oorschot and Wiener [OW99] provides a speedup by a factor of $\Theta(N)$ when running on N computers, if ℓ is larger than a suitable power of N. In several situations a group automorphism of small order m provides a further speedup by a factor of $\Theta(\sqrt{m})$. No further speedups are known for any elliptic curve chosen according to standard security criteria.

However, these asymptotic iteration counts ignore many factors that have an important influence on the cost of an attack. Understanding the hardness of a specific ECDLP requires a more thorough investigation. The publications summarized on [Gir11], giving recommendations for concrete cryptographic key sizes, all extrapolate from such investigations. To reduce extrapolation errors it is important to use as many data points as possible, and to push these investigations beyond the ECDLP computations that have been carried out before.

Certicom published a list of ECDLP challenges in 1997 at [Cer97] in order to "increase the cryptographic community's understanding and appreciation of the difficulty of the ECDLP". These challenges range from very easy exercises, solved in 1997 and 1998, to serious cryptanalytic challenges. The last Certicom challenge that was publicly broken was a 109-bit ECDLP in 2004. Certicom had already predicted the lack of further progress: it had stated in [Cer97, page 20] that the subsequent challenges were "expected to be infeasible against realistic software and hardware attacks, unless of course, a new algorithm for the ECDLP is discovered."

Since then new hardware platforms have become available to the attacker. Processor design has moved away from increasing the clock speed and towards increasing the number of cores. This means that implementations need to be parallelized in order to make full use of the processor. Running a serial implementation on a recent processor might take longer than five years ago, because the average clock speed has decreased, but if this implementation can be parallelized and occupy the entire processor then the implementation will run much faster.

Certicom's estimate was that ECC2K-130, the first "infeasible" challenge, would require (on average) 2 700 000 000 "machine days" of computation. The main result of this chapter is that a cluster of just 1066 NVIDIA GTX 295 graphics cards or 2636 PlayStation 3 gaming consoles would solve ECC2K-130 in just one year; therefore nowadays this challenge requires only 379 496 machine days on a graphics card or 938 416 machine days on a gaming console.

The research presented in this chapter is joint work with Bos, Kleinjung, and Schwabe published in [BKN$^+$10] as well as with Bernstein, Chen, Cheng, Lange, Schwabe, and Yang published in [BCC$^+$10]. It is also included in the PhD thesis of Schwabe [Sch11a, Chapter 6]. Furthermore it is part of a large collaborative project that has optimized ECDLP computations for several different platforms and that aims to break ECC2K-130. See [BBB$^+$09] and [Ano] for more information about the project. This research has been supported by the Netherlands National Computing Facilities foundation (NCF) as project MP-185-10.

This chapter is structured as follows: Section 3.1 and Section 3.2 briefly introduce Pollard's rho method and the iteration function of this method for ECC2K-130. Section 3.3 gives a detailed description of the implementation for the Cell processor and section 3.4 for the GTX 295 graphics card. Section 3.5 will conclude with a comparison of both implementations.

3.1 The ECDLP and the parallel version of Pollard's rho method

The standard method for solving the ECDLP in prime-order subgroups is Pollard's rho method [Pol78]. For large instances of the ECDLP, one usually uses a parallelized version of Pollard's rho method due to van Oorschot and Wiener [OW99]. This section briefly reviews the ECDLP and the parallel version of Pollard's rho method.

The ECDLP is the following problem: Given an elliptic curve E over a finite field $\mathbb{F}_q$ and two points $P \in E(\mathbb{F}_q)$ and $Q \in \langle P \rangle$, find an integer k such that $Q = [k]P$.

Let ℓ be the order of P, and assume in the following that ℓ is prime. The parallel version of Pollard's rho method uses a client-server approach in which each client does the following:

1. Generate a pseudo-random starting point R_0 as a known linear combination of P and Q: $R_0 = a_0P + b_0Q$;
2. apply a pseudo-random iteration function f to obtain a sequence $R_{i+1} = f(R_i)$, where f is constructed to preserve knowledge about the linear combination of P and Q;
3. for each i, after computing R_i, check whether R_i belongs to an easy-to-recognize set D, the set of distinguished points, a subset of $\langle P \rangle$;
4. if at some moment a distinguished point R_i is reached, send (R_i, a_i, b_i) to the server and go to step 1.

The server receives all the incoming triples (R, a, b) and does the following:

1. Search the entries for a *collision*, i.e., two triples $(R, a, b), (R', a', b')$ with $R = R'$ and $b \not\equiv b' \pmod{\ell}$;
2. obtain the discrete logarithm of Q to the base P as $k = \frac{a'-a}{b-b'}$ modulo ℓ.

The expected number of calls to the iteration function f is approximately $\sqrt{\pi\ell/2}$ assuming perfectly random behavior of f. Therefore the expected total runtime of the parallel version of Pollard's rho algorithm is $\sqrt{\pi\ell/2} \cdot t/N$ given the time t for executing one step of iteration function f and N computing devices.

The implementation of the server and the communication between the clients and the server has been implemented by Bernstein and is not part of this thesis. Details on this part of the ECC2K-130 computation can be found in [BBB$^+$09, Appendices C and D]. The server for collecting and processing the distinguished points that are computed by the clients is running on a cluster at the Eindhoven Institute for the Protection of Systems and Information (Ei/Ψ) at the University of Technology Eindhoven.

3.2 ECC2K-130 and the iteration function

The specific ECDLP addressed in this chapter is given in the list of Certicom challenges [Cer97] as Challenge ECC2K-130. The given elliptic curve is the Koblitz curve $E : y^2 + xy = x^3 + 1$ over the finite field $\mathbb{F}_{2^{131}}$; the two given points P and Q have order ℓ, where ℓ is a 129-bit prime. This section reviews the definition of distinguished points and the iteration function used in [BBB$^+$09]. For a more detailed discussion, an analysis of communication costs, and a comparison to other possible implementation choices, the interested reader is referred to [BBB$^+$09].

Definition of the iteration function. A point $R \in \langle P \rangle$ is *distinguished* if $\mathrm{HW}(x_R)$, the Hamming weight of the x-coordinate of R in normal-basis representation, is smaller than or equal to 34. The iteration function is defined as

$$R_{i+1} = f(R_i) = \sigma^j(R_i) + R_i,$$

where σ is the Frobenius endomorphism and $j = ((\mathrm{HW}(x_{R_i})/2) \bmod 8) + 3$.

The restriction of σ to $\langle P\rangle$ corresponds to scalar multiplication with a particular easily computed scalar r. For an input $R_i = a_iP + b_iQ$, the output of f will be $R_{i+1} = (r^j a_i + a_i)P + (r^j b_i + b_i)Q$.

Each walk starts by picking a random 64-bit seed s which is then expanded deterministically into a linear combination $R_0 = a_0P+b_0Q$. To reduce bandwidth and storage requirements, the client does not report a distinguished triple (R, a, b) to the server but instead transmits only s and a 64-bit hash of R. On the occasions that a hash collision is found, the server recomputes the linear combinations in P and Q for $R = aP + bQ$ and $R' = a'P + b'Q$ from the corresponding seeds s and s'. This has the additional benefit that the client does not need to keep track of the coefficients a and b or maintain counters for how often each Frobenius power is used. This speedup is particularly beneficial for highly parallel architectures such as GPUs and vector processors, which otherwise would need a conditional addition to each counter in each step.

Computing the iteration function. Computing the iteration function requires one application of σ^j and one elliptic-curve addition. Furthermore the x-coordinate of the resulting point is converted to normal basis, if a polynomial-basis representation is used, to check whether the point is a distinguished point and to obtain j.

Many applications use so-called inversion-free coordinate systems to represent points on elliptic curves (see, e.g., [HMV04, Sec. 3.2]) to speed up the computation of point multiplications. These coordinate systems use a redundant representation for points. Identifying distinguished points requires a unique representation. This is why affine Weierstrass representation is used to represent points on the elliptic curve. Elliptic-curve addition in affine Weierstrass coordinates on the given elliptic curve requires 2 multiplications, 1 squaring, 6 additions, and 1 inversion in $\mathbb{F}_{2^{131}}$ (see, e.g. [BL07]). Application of σ^j means computing the 2^j-th powers of the x- and the y-coordinate. In total, one iteration takes 2 multiplications, 1 squaring, 2 computations of the form r^{2^m}, with $3 \leq m \leq 10$, 1 inversion, 1 conversion to normal-basis, and 1 Hamming-weight computation. In the following computations of the form r^{2^m} will be called *m-squaring.*

A note on the inversion. Montgomery's trick [Mon87] can be used on a batch of inversions to speed up this relatively costly operation: m batched inversions can be computed with $3(m - 1)$ multiplications and one inversion. For example, $m = 64$ batched elliptic curve additions take $2 \cdot 64 + 3 \cdot (64 - 1) = 317$ multiplications, 64 squarings and 1 inversion. This corresponds to 4.953 multiplications, 1 squaring and 0.016 inversions for a single elliptic-curve addition.

3.3 Implementing ECC2K-130 on the Cell processor

The Cell processor is a hybrid multi-core processor consisting of a traditional PowerPC core for general-purpose tasks, in particular running the operating system, and up to eight vector cores with fully controlled caches. Since the Cell processor is a highly parallel architecture, it is well suited for the parallel workload of Pollard's rho method. The Cell processor powers the PlayStation 3 gaming console and therefore can be obtained at relatively low cost.

This section is structured as follows: First the Cell processor is introduced. Then general design questions for the implementation of the iteration function are stated. This is followed by a detailed description of the implementation of the iteration function and a close investigation of the memory management. Eventually the results of the implementation for the Cell processor are discussed.

3.3.1 A Brief Description of the Cell processor

The Cell Broadband Engine Architecture [Hof05] was jointly developed by Sony, Toshiba and IBM. By November 2011 there were two implementations of this architecture, the Cell Broadband Engine (Cell/B.E.) and the PowerXCell 8i. The PowerXCell 8i is a derivative of the Cell/B.E. and offers enhanced double-precision floating-point capabilities and an improved memory interface. Both implementations consist of a central *Power Processor Element* (PPE), based on the Power 5 architecture and 8 *Synergistic Processor Elements* (SPEs) which are optimized for high-throughput vector instructions. All units are linked by a high-speed ring bus with an accumulated bandwidth of 204 GB/s.

The Cell/B.E. can be found in the IBM blade servers of the QS20 and QS21 series, in the Sony PlayStation 3, and several acceleration cards like the Cell Accelerator Board from Mercury Computer Systems. The PowerXCell 8i can be found in the IBM QS22 servers. Note that the PlayStation 3 only makes 6 SPEs available to the programmer.

The code described in this section runs on the SPEs directly and does not interact with the PPE or other SPEs during core computation. The enhanced double-precision floating-point capabilities of the PowerXCell 8i are not required for this implementation of the iteration function. Therefore in the remainder of this section only those features of the SPE are described which are of interest for the implementation and are common to both the Cell/B.E. and the PowerXCell 8i. The Cell/B.E. and the PowerXCell 8i will be addressed jointly as the *Cell processor*. More detailed information on the Cell Broadband Engine Architecture can be found in [IBM08].

Each SPE consists of a *Synergistic Processor Unit* (SPU) as its computation unit and a *Memory Flow Controller* (MFC) which grants access to the ring bus and therefore in particular to main memory.

Architecture and instruction set. The SPU is composed of three parts: The execution unit is the computational core of each SPE. It is fed with data either by the register file or by the *local storage* that also feeds the instructions into the execution unit.

The register file contains 128 general-purpose registers with a width of 128 bits each. The execution unit has fast and direct access to the local storage but the local storage is limited to only 256 KB of memory. The execution unit does not have transparent access to main memory; all data must be transferred from main memory to local storage and vice versa explicitly by instructing the DMA controller of the MFC. Due to the relatively small size of the local storage and the lack of transparent access to main memory, the programmer has to ensure that instructions and the active data set fit into the local storage and are transferred between main memory and local storage accordingly.

The execution unit has a pure RISC-like SIMD instruction set encoded into 32-bit instruction words; instructions are issued strictly in order to two pipelines called *odd* and *even pipeline*, which execute disjoint subsets of the instruction set. The even pipeline handles floating-point operations, integer arithmetic, logical instructions, and word shifts and rotates. The odd pipeline executes byte-granularity shift, rotate-mask, and shuffle operations on quadwords, and branches as well as loads and stores.

Up to two instructions can be issued each cycle, one in each pipeline, given that alignment rules are respected (i.e., the instruction for the even pipeline is aligned to a multiple of 8 bytes and the instruction for the odd pipeline is aligned to a multiple of 8 bytes plus an offset of 4 bytes), that there are no interdependencies to pending previous instructions for either of the two instructions, and that there are in fact at least two instructions available for execution. Therefore, a careful scheduling and alignment of instructions is necessary to achieve best performance.

Accessing main memory. As mentioned before, the MFC is the gate for the SPU to reach main memory as well as other processor elements. Memory transfer is initiated by the SPU and afterwards executed by the DMA controller of the MFC in parallel to ongoing instruction execution by the SPU.

Since data transfers are executed in the background by the DMA controller, the SPU needs feedback about when a previously initiated transfer has finished. Therefore, each transfer is tagged with one of 32 tags. Later on, the SPU can probe either in a blocking or non-blocking way if a subset of tags has any outstanding transactions. The programmer should avoid reading data buffers for incoming data or writing to buffers for outgoing data before checking the state of the corresponding tag to ensure deterministic program behaviour.

Accessing local storage. The local storage is single ported and has a line interface of 128 bytes width for DMA transfers and instruction fetch as well as a quadword interface of 16 bytes width for SPU load and store. Since there is only one port, the access to the local storage is arbitrated using the highest priority for DMA transfers (at most every 8 cycles), followed by SPU load/store, and the lowest priority for instruction fetch.

Instructions are fetched in lines of 128 bytes, i.e., 32 instructions. In the case that all instructions can be dual issued, new instructions need to be fetched every 16 cycles. Since SPU loads/stores have precedence over instruction fetch, in case of high memory access there should be a `NOP` instruction for the odd pipeline every 16 cycles to avoid instruction starvation. If there are ongoing DMA transfers an `HBRP` instruction should be used giving instruction fetch explicit precedence over DMA transfers.

Determining performance. The instruction set of the Cell processor gives access to a decrementer (see [IBM08, Sec. 13.3.3]) for timing measurements. The disadvantage of this decrementer is that it is updated with the frequency of the so-called timebase of the processor. The timebase is usually much smaller than the processor frequency. The Cell processor (rev. 5.1) in the PlayStation 3 for example changes the decrementer only every 40 cycles, the Cell processor in the QS21 blades even only every 120 cycles. Small sections of code can thus only be measured on average by running the code several times repeatedly. All cycle counts reported in this chapter have been measured by running the code and reading the decrementer.

3.3.2 Approaches for implementing the iteration function

The following paragraphs discuss two main design decisions for the implementation of the iteration function: Is it faster to use bitslicing or a standard approach and is it better to use normal-basis or polynomial-basis representation for elements of the finite field.

Bitsliced or not bitsliced? Binary-field arithmetic was commonly believed to be more efficient than prime-field arithmetic for hardware but less efficient for software implementations. This is due to the fact that most common microprocessors spend high effort on accelerating integer- and floating-point multiplications. Prime-field arithmetic can benefit from those high-speed multiplication instructions, binary-field arithmetic cannot. However, Bernstein showed that for *batched* multiplications, binary fields can provide better performance than prime fields also in software [Ber09c]. In his implementation of batched Edwards-curve arithmetic the bitslicing technique is used to compute (at least) 128 binary-field multiplications in parallel on an Intel Core 2 processor.

Bitslicing is a matter of transposition: Instead of storing the coefficients of an element of $\mathbb{F}_{2^{131}}$ as sequence of 131 bits in 2 128-bit registers, 131 registers can be used to store the 131 coefficients of an element, one register per bit. Algorithms are then implemented by simulating a hardware implementation—gates become bit operations such as `AND` and `XOR`. For one element in 131 registers this is highly inefficient, but it may become efficient if all 128 bits of the registers are used for independent operations on 128 field elements. The number of field elements which are processed in parallel using the bitslicing technique is called *bitslicing width*. The lack of registers—most architectures including the SPU do not support 131 registers—can easily be compensated for by register spills, i.e. storing currently unused values on the stack and loading them when they are required.

The results of [Ber09c] show that for batched binary-field arithmetic on the Intel Core 2 processor bitsliced implementations are faster than non-bitsliced implementations. To determine whether this is also true for binary-field arithmetic on the Cell processor, the iteration function was implemented with both approaches, a bitsliced and a non-bitsliced version. It turned out that the bitsliced version is faster than the non-bitsliced version. Both versions are described in detail in [BKN$^+$10]. In this chapter only the faster bitsliced version will be explained.

Polynomial or normal basis? Another choice to make for both bitsliced and non-bitsliced implementations is the representation of elements of $\mathbb{F}_{2^{131}}$: Polynomial bases are of the form $(1, z, z^2, z^3, \ldots, z^{130})$, so the basis elements are increasing powers of some element $z \in \mathbb{F}_{2^{131}}$. Normal bases are of the form $(\alpha, \alpha^2, \alpha^4, \ldots, \alpha^{2^{130}})$, so each basis element is the square of the previous one.

Performing arithmetic in normal-basis representation has the advantage that squaring elements is just a rotation of coefficients. Furthermore there is no need for basis transformation before computing the Hamming weight in normal basis. On the other hand, implementations of multiplications in normal basis are widely believed to be less efficient than those of multiplications in polynomial basis.

In [GSS07], von zur Gathen, Shokrollahi and Shokrollahi proposed an efficient method to multiply elements in type-2 optimal-normal-basis representation (see also [Sho07]). The following gives a review of this multiplier as shown in [BBB$^+$09]:

An element of $\mathbb{F}_{2^{131}}$ in type-2 optimal-normal-basis representation is of the form

$$f_0(\zeta + \zeta^{-1}) + f_1(\zeta^2 + \zeta^{-2}) + f_2(\zeta^4 + \zeta^{-4}) + \cdots + f_{130}(\zeta^{2^{130}} + \zeta^{-2^{130}}),$$

where ζ is a 263rd root of unity in $\mathbb{F}_{2^{131}}$. This representation is first permuted to obtain coefficients of

$$\zeta + \zeta^{-1}, \zeta^2 + \zeta^{-2}, \zeta^3 + \zeta^{-3}, \ldots, \zeta^{131} + \zeta^{-131},$$

and then transformed to coefficients in polynomial basis

$$\zeta + \zeta^{-1}, (\zeta + \zeta^{-1})^2, (\zeta + \zeta^{-1})^3, \ldots, (\zeta + \zeta^{-1})^{131}.$$

Applying this transformation to both inputs makes it possible to use a fast polynomial-basis multiplier to retrieve coefficients of

$$(\zeta + \zeta^{-1})^2, (\zeta + \zeta^{-1})^3, \ldots, (\zeta + \zeta^{-1})^{262}.$$

Applying the inverse of the input transformation yields coefficients of

$$\zeta^2 + \zeta^{-2}, \zeta^3 + \zeta^{-3}, \ldots, \zeta^{262} + \zeta^{-262}.$$

Conversion to permuted normal basis just requires adding appropriate coefficients, for example ζ^{200} is the same as ζ^{-63} and thus $\zeta^{200} + \zeta^{-200}$ is the same as $\zeta^{63} + \zeta^{-63}$. The normal-basis representation can be computed by applying the inverse of the input permutation.

This multiplication still incurs overhead compared to modular multiplication in polynomial basis, but it needs careful analysis to understand whether this overhead is compensated for by the above-described benefits of normal-basis representation. Observe that all permutations involved in this method are free for hardware and bitsliced implementations while they are quite expensive in non-bitsliced software implementations. Nevertheless, it is not obvious which basis representation has better performance for a bitsliced implementation. Therefore all finite-field operations are implemented in both polynomial- and normal-basis representation. By this the overall runtime can be compared to determine which approach gives the better performance.

3.3.3 ECC2K-130 iterations on the Cell processor

This section describes the implementation of the speed-critical parts of the iteration function in detail. Due to bitslicing and the register width of 128 bits, all operations process 128 inputs in parallel. The cycle counts stated in this section therefore account for 128 parallel computations. The polynomial-basis implementation uses $\mathbb{F}_{2^{131}} \cong \mathbb{F}_2[z]/(z^{131} + z^{13} + z^2 + z + 1)$.

Multiplication. The smallest known number of bit operations required to multiply two degree-130 polynomials over $\mathbb{F}_2$ is 11961 [Ber09b]. However, converting the sequence of bit operations in [Ber09b] to C syntax and feeding it to the compiler did not succeed because the size of the resulting function exceeds the size of the local storage. After reducing the number of variables for intermediate results and using some more tweaks the compiler produced functioning code, which had a code size of more than 100 KB and required more than 20000 cycles to compute a multiplication.

To achieve better performance, each polynomial of 131 coefficients can be extended to 132 coefficients by adding a zero at the highest position. Now two levels of the Karatsuba multiplication technique [KO63] can easily be applied to perform the multiplication of two 132-coefficient polynomials as nine 32-coefficient polynomial multiplications with some additional operations. This increases the number of bit operations but results in better-scheduled and more efficient code. Furthermore improvements to classical Karatsuba can be used to combine the results of the 9 multiplications as described in [Ber09a].

With this approach, one 131-coefficient polynomial multiplication takes 14503 cycles in total. This includes 11727 cycles for nine 32-coefficient polynomial multiplications, cycles required for combination of the results, and function-call overhead. Reduction modulo the pentanomial $z^{131} + z^{13} + z^2 + z + 1$ takes 520 bit operations, the fully unrolled reduction function takes 590 cycles, so multiplication in $\mathbb{F}_{2^{131}}$ takes $14503 + 590 = 15093$ cycles.

The *normal-basis* multiplication uses the conversion to polynomial basis as described in Section 3.3.2. Both conversion of inputs to polynomial basis and conversion of the result to normal basis (including reduction) is performed by fully unrolled assembly functions. One input conversion takes 434 cycles; output conversion including reduction takes 1288 cycles; one normal-basis multiplication including all conversions and runtime overhead takes 16635 cycles.

Squaring. In *polynomial-basis* representation, squaring consists of two parts: Zero-bits are inserted between all bits of the input and this intermediate result must be reduced by modular reduction. The first part does not require any instructions in bitsliced representation because the additional zeros do not need to be stored anywhere. Instead, the change of the positions must be respected during the reduction. For squaring the reduction is cheaper than for multiplication because it is known that every second bit is zero. In total squaring needs 190 bit operations, hence, it is bottlenecked by loading 131 inputs and storing 131 outputs. One call to the squaring function takes 400 cycles.

In *normal-basis* representation a squaring is a cyclic shift of bits. This is performed by 131 loads and 131 stores to cyclically shifted locations. A call to the squaring function in normal-basis representation takes 328 cycles.

m-Squaring. For *polynomial-basis* representation m-squarings are implemented as a sequence of squarings. Fully unrolled code can hide most of the 131 load and 131 store operations between the 190 bit operations of a squaring. Implementing dedicated m-squaring functions for different values of m would mostly remove the overhead of $m-1$ function calls but on the other hand significantly increase the overall code size.

For the *normal-basis* representation separate m-squarings are used for all relevant values of m as fully unrolled functions. The only difference between these functions is the shifting distance of the store locations. Each m-squaring therefore takes 328 cycles, just like a single squaring.

Computation of σ^j. The computation of σ^j cannot be realized as a single m-squaring with $m = j$, because the value of $j = ((\mathrm{HW}(x_{R_i})/2) \bmod 8) + 3$ is most likely different for the 128 bitsliced values in one batch. For the three bits b_1, b_2, b_3 (skipping the least significant bit b_0) of x_{R_i}, j can be computed as $j = b_1 + 2b_2 + 4b_3 + 3$. Therefore the computation of $r = \sigma^j(x_{R_i})$ can be carried out using 1 unconditional and 3 conditional m-squarings as follows:

```
    $r \leftarrow x_{R_i}^{2^3}$
    if $b_1$ then $r \leftarrow r^2$
    if $b_2$ then $r \leftarrow r^{2^2}$
    if $b_3$ then $r \leftarrow r^{2^4}$
    return $r$
```

The computation of $\sigma^j(y_{R_i})$ is carried out in the same way.

When using bitsliced representation, conditional statements have to be replaced by equivalent arithmetic computations. The k-th bit of the result of a conditional m-squaring of r depending on a bit b is computed as

$$r_k \leftarrow (r_k \wedge \neg b) \oplus (r_k^{2^m} \wedge b).$$

The additional three bit operations per output bit can be interleaved with the loads and stores which are needed for squaring. In particular when using normal-basis representation (which does not involve any bit operations for squarings), this speeds up the computation: In case of *normal-basis* representation, computation of σ^j takes 1380 cycles.

For *polynomial-basis* representation a conditional m-squaring consists of an m-squaring followed by a conditional move. The conditional move function requires 262 loads, 131 stores and 393 bit operations and thus balances instructions on the two pipelines. One call to the conditional move function takes 518 cycles. In total, computing σ^j takes 10 squarings and 3 conditional moves summing up to 5554 cycles.

Addition. Addition is the same for *normal-basis* and *polynomial-basis* representation. It requires loading 262 inputs, 131 `XOR` instructions and storing of 131 outputs. Just as squaring, the function is bottlenecked by loads and stores rather than bit operations. One call to the addition function takes 492 cycles.

Inversion. For both *polynomial-basis* and *normal-basis* representation the inversion is implemented using Fermat's little theorem. It involves 8 multiplications, 3 squarings and 6 m-squarings (with $m = 2, 4, 8, 16, 32, 65$). It takes 173325 cycles using *polynomial-basis* representation and 135460 cycles using *normal-basis* representation. Observe that with a sufficiently large batch size for Montgomery inversion this does not have big impact on the cycle count of one iteration.

Conversion to normal basis. For *polynomial-basis* representation the x-coordinate must be converted to normal basis before it can be detected whether it belongs to a distinguished point. This basis conversion is generated using the techniques described in [Ber09c] and uses 3380 bit operations. The carefully scheduled code takes 3748 cycles.

Hamming-weight computation. The bitsliced Hamming-weight computation of a 131-bit number represented in *normal-basis* representation can be done in a divide-and-conquer approach (producing bitsliced results) using 625 bit operations. This algorithm was unrolled to obtain a function that computes the Hamming weight using 844 cycles.

Overhead. For both *polynomial-basis* representation and *normal-basis* representation there is additional overhead from loop control and reading new input points after a distinguished point has been found. This overhead accounts only for about 8 percent of the total computation time. After a distinguished point has been found, reading a new input point takes about 2 009 000 cycles. As an input point takes on average $2^{25.7} \approx 40\,460\,197$ iterations to reach a distinguished point, these costs are negligible and are ignored in the overall cycle counts for the iteration function.

3.3.4 Using DMA transfers to increase the batch size

As long as main memory is not used during computation of the iteration function, the batch size for Montgomery inversions is restricted by the size of the local storage. Recall that the program code, the stack, and the current working set of the data needs to fit into local storage during execution. In case of polynomial-basis representation the storage space is sufficient for a batch size of 12 bitsliced iterations. The average time for one bitsliced iteration is 113844 cycles. The normal-basis implementation requires less space for program code and therefore a maximum batch size of 14 iterations fits into local storage. One bitsliced iteration takes 99994 cycles.

For comparison, using only a batch size of 12 iterations for the normal-basis representation gives an average runtime of approximately 101607 cycles per bitsliced iteration. This is about 10% less than the time for polynomial-basis representation with the same batch size. Clearly, the overhead caused by the conversions for multiplications in the normal-basis implementation is redeemed by the benefits in faster m-squarings, conditional m-squarings, and the saved basis conversion before Hamming-weight computation. Therefore the normal-basis implementation was chosen to be further improved by storing data in main memory as well: A larger number for the batch size of Montgomery inversion can be achieved by taking advantage of DMA transfers from and to main memory. The batches are stored in main memory and are fetched into local storage temporarily for computation.

Since the access pattern to the batches is totally deterministic, it is possible to use multi-buffering to prefetch data while processing previously loaded data and to write back data to main memory during ongoing computations. At least three slots—one for outgoing data, one for computation, and one for incoming data—are required in local storage for the buffering logic. The slots are organized as a ring buffer. One DMA tag is assigned to each of the slots to monitor ongoing transactions.

Before the computation starts, the first batch is loaded into the first slot in local storage. During one step of the iteration function, the SPU iterates multiple times over the batches. Each time, first the SPU checks whether the last write back from the next slot in the ring-buffer has finished. This is done using a blocking call to the MFC on the tag assigned to the slot. When the slot is free for use, the SPU initiates a prefetch for the next required batch into the next slot. Now—again in a blocking manner—it is checked whether the data for the current batch already has arrived. If so, data is processed and finally the SPU initiates a DMA transfer to write changed data back to main memory.

Due to this access pattern, all data transfers can be performed with minimal overhead and delay. Therefore it is possible to increase the batch size to 512 improving the runtime per iteration for the normal basis implementation by about 5% to 94949 cycles. Measurements on IBM blade servers QS21 and QS22 showed that neither processor bus nor main memory is a bottleneck even if 8 SPEs are doing independent computations and DMA transfers in parallel.

3.3.5 Overall results on the Cell processor

From the two implementations described in this section it becomes evident that on the Cell processor normal-basis representation of finite-field elements outperforms polynomial-basis representation when using a bitsliced implementation. The cycle counts for all field operations are summarized in Table 3.1. The numbers are normalized to operations on a single input, i.e. the numbers mentioned previously for bitsliced measurements are divided by the bitslicing width of 128bits.

Using the bitsliced normal-basis implementation—which employs DMA transfers to main memory to support a batch size of 512 for Montgomery inversions—on all 6 SPUs of a Sony PlayStation 3 in parallel, 25.88 million iterations can be computed per second. The expected total number of iterations required to solve the ECDLP given in the ECC2K-130 challenge is $2^{60.9}$ (see [BBB$^+$09]). This number of iterations can be computed in one year using only the SPEs of 2636 PlayStation 3 gaming consoles.

building block	**polynomial basis cycles**	**normal basis cycles**
multiplication	117.91	129.96
squaring / m-squaring	$m \cdot 3.16$	2.56
computation of σ^j	43.39	10.78
Hamming-weight computation	6.60	6.60
addition	3.84	3.84
inversion	1354.10	1058.28
conversion to normal basis	29.28	—
full iteration:		
$B = 12$	889.41	793.80
$B = 14$	—	781.20
$B = 512$	—	741.79

Table 3.1: Cycle counts per input for all building blocks on one SPE of a 3192 MHz Cell Broadband Engine (rev. 5.1). Cycle counts for 128 bitsliced inputs are divided by 128. The value B in the last row denotes the batch size for Montgomery inversions.

At the time of this writing, the computations have not been finished so far. The Cell implementation has been running on the MariCel cluster at the Barcelona Supercomputing Center (BSC), on the JUICE cluster at the Jülich Supercomputing Centre, and on the PlayStation 3 cluster of the École Polytechnique Fédérale de Lausanne (EPFL).

3.4 Implementing ECC2K-130 on a GPU

GPUs are massively parallel computing architectures, even more so than the Cell processor. As explained in Section 2.3, they have initially been developed as highly specialized gaming devices but nowadays they are also used for general-purpose computing. Compared to the Cell processor GPUs offer a much higher number of ALUs but a smaller amount of low-latency storage per ALU. Therefore the GPU implementation of the iteration function diverges significantly from the Cell implementation.

This section is organized as follows: First the target platform for this implementation, NVIDIA's GTX 295 graphics card, is introduced. Then general design decisions for the GPU implementation are explained. This is followed by a detailed description of the polynomial multiplication on the GPU because this is the most expensive and complex operation. Afterwards all other operations for the iteration function are described briefly. This section is concluded with a brief overview of the performance results for the GPU implementation.

3.4.1 The GTX 295 graphics card

The most impressive feature of GPUs is their theoretical floating-point performance. Each of the 480 ALUs on a GTX 295 can dispatch a single-precision floating-point multiplication (with a free addition) every cycle at a clock frequency of 1.242 GHz. There are also 120 "special-function units" that can each dispatch another single-precision floating-point multiplication every cycle, for a total of 745 billion floating-point multiplications per second.

The most useful GPU arithmetic instructions for the ECC2K-130 computation are 32-bit logical instructions (`AND` and `XOR`) rather than floating-point multiplications. Logical instructions can be executed only by the 480 ALUs. Nevertheless, 596 billion 32-bit logical instructions per second are still much more impressive than, e.g., the 28.8 billion 128-bit logical instructions per second performed by a typical 2.4 GHz Intel Core 2 CPU with 4 cores and 3 128-bit ALUs per core.

However, the GPUs also have many bottlenecks that make most applications run slower, often one or two orders of magnitude slower, than the theoretical throughput figures would suggest. The most troublesome bottlenecks are discussed in the remainder of this section and include a heavy divergence penalty, high instruction latency, low SRAM capacity, high DRAM latency, and relatively low DRAM throughput per ALU.

The dispatcher. The 8 ALUs in a GPU core are fed by a single *dispatcher*. The dispatcher cannot issue more than one new instruction to the ALUs every 4 cycles. The dispatcher can send this one instruction to a *warp* containing 32 separate *threads* of computation, applying the instruction to 32 pieces of data in parallel and keeping all 8 ALUs busy for all 4 cycles; but the dispatcher cannot direct some of the 32 threads to follow one instruction while the remaining threads follow another.

Branching is allowed, but if threads within one warp take different branches, the threads taking one branch will no longer operate in parallel with the threads in the other branch; execution of the two branches is serialized and the time it takes to execute diverging branches is the sum of the time taken in all branches.

SRAM: registers and shared memory. Each core has 16384 32-bit registers; these registers are divided among the threads. For example, if the core is running 256 threads, then each thread is assigned 64 registers. If the core is running 128 threads, then each thread is assigned 128 registers, although access to the high 64 registers is somewhat limited: the architecture does not allow a high register as the second operand of an instruction. Even with fewer than 128 threads, only 128 registers are available per thread.

The core also has 16384 bytes of *shared memory*. This memory enables communication between threads. It is split into 16 banks, each of which can dispatch one 32-bit read or write operation every two cycles. To avoid bank conflicts, either each of the 16 threads of a half-warp must access different memory banks or the same memory address must be touched by all 16 threads. Otherwise accesses to the same memory bank are serialized; in the worst case, when all threads are requesting data from different addresses on the same bank, memory access takes 16 times longer than memory access without bank conflicts.

Threads also have fast access to an 8192-byte *constant cache*. This cache can broadcast a 32-bit value from one location to every thread simultaneously, but it cannot read more than one location per cycle.

DRAM: global memory and local memory. The CPU makes data available to the GPU by copying it into DRAM on the graphics card outside the GPU. The cores on the GPU can then load data from this *global memory* and store results in global memory to be retrieved by the CPU. Global memory is also a convenient temporary storage area for data that does not fit into shared memory. However, global memory is limited to a throughput of just one 32-bit load from each GPU core per cycle, with a latency of 400–600 cycles.

Each thread also has access to *local memory*. The name "local memory" might suggest that this storage is fast, but in fact it is another area of DRAM, as slow as global memory. Instructions accessing local memory automatically incorporate the thread ID into the address being accessed, effectively partitioning the local memory among threads without any extra address-calculation instructions.

There are no hardware caches for global memory and local memory. Programmers can, and must, set up their own schedules for moving data between shared memory and global memory.

Instruction latency. The ALUs execute the instruction stream strictly in order. NVIDIA does not document the exact pipeline structure but recommends running at least 192 threads (6 warps) on each core to hide arithmetic latency. If all 8 ALUs of a core are fully occupied with 192 threads then each thread runs every 24 cycles; evidently the latency of an arithmetic instruction is at most 24 cycles.

One might think that a single warp of 32 threads can keep the 8 ALUs fully occupied, if the instructions in each thread are scheduled for 24-cycle arithmetic latency (i.e., if an arithmetic result is not used until 6 instructions later). However, if only one warp is executed on one core, the dispatcher will issue instructions only every second dispatching cycle. Therefore at least 2 warps (64 threads) are necessary to exploit all ALU cycles. Furthermore, experiments showed that additional penalty is encountered when shared memory is accessed. This penalty can be hidden if enough warps are executed concurrently or if the density of memory accesses is sufficiently low. For instance the ALU can be kept busy in all cycles with 128 threads as long as fewer than 25% of the instructions include shared-memory access and as long as these instructions are not adjacent.

NVIDIA also recommends running many more than 192 threads to hide DRAM latency. This does not mean that one can achieve the best performance by simply running the maximum number of threads that fit into the core. Threads share the register bank and shared memory, so increasing the number of threads means reducing the amount of these resources available to each thread. The ECC2K-130 computation puts extreme pressure on shared memory, as discussed later in this section; to minimize this pressure, this implementation is using 128 threads, skirting the edge of severe latency problems.

3.4.2 Approaches for implementing the iteration function

Similar to the implementation for the Cell processor, two major design decisions must be made for the GPU implementation: Whether bitslicing should be used and what basis representation should be chosen. The experiences made for the Cell-processor can be taken into account for the GPU implementation:

Bitsliced or not bitsliced? The main task in optimizing multiplication, squaring, etc. in $\mathbb{F}_{2^{131}}$ is to decompose these arithmetic operations into the operations available on the target platform. Similarly to the Cell processor bitslicing is used on the GPU as it allows very efficient usage of logical operations in implementing binary-field operations.

The bitslicing technique has the disadvantage that it increases the size of the working set roughly by a factor of the bitslicing width. Since register spills to DRAM are expensive, this is particularly troublesome for architectures which offer a small amount of low-latency memory. In case of the GTX 295 this is the register bank and shared memory. Section 3.4.3 explains how GPU threads can be programmed to cooperate on the computation of the iteration function. Compared to an implementation where the threads process data independently, by this cooperation the overall working set of one processor core can be reduced while enabling usage of a large number of concurrent threads.

Polynomial or normal basis? From the Cell implementation of the iteration function it is known that normal-basis representation outperforms polynomial-basis representation for bitsliced implementations. Therefore the GPU implementation also uses Shokrollahi's type-2 optimal-normal-basis representation as explained in Section 3.3.2. Compared to the Cell implementation, the performance of the normal-basis representation has been improved by applying a technique introduced by Bernstein and Lange in [BL10]. Bernstein and Lange reduced the overhead that is introduced by basis conversion by combining the optimal normal basis with an *optimal polynomial basis*. This makes it possible to skip the conversion from normal basis to polynomial basis in cases where several multiplications are performed consecutively: The output of a polynomial multiplication is converted to a suitable input for a follow-up multiplication directly without fully converting to normal basis in between. This technique reduces the cost of multiplications for the ECC2K-130 iteration compared to the Cell processor implementation.

Therefore the GPU implementation uses two multiplication routines: In case a multiplication is followed by a squaring and therefore a conversion to normal basis is necessary, the function *PPN* is called that takes two inputs in polynomial-basis representation, performs a multiplication, and returns an output in normal basis representation as described in Section 3.3.2.

In case a multiplication is followed by another multiplication, the function *PPP* is called. This function takes two inputs in polynomial-basis representation and performs a multiplication. Now it keeps the lower half of the polynomial product in polynomial-basis representation and use the conversion routine only to compute the polynomial reduction, ending up in polynomial-basis representation. For full details on the conversion routines see [BL10].

If a value in normal-basis representation needs to be converted to polynomial basis as an input for one of these multiplication functions, a separate conversion function is called.

3.4.3 Polynomial multiplication on the GPU

The costs of either type of field multiplication, PPN or PPP, are dominated by the costs of the 131-bit polynomial multiplication. With optimal polynomial bases (see Section 3.4.2), each iteration involves slightly more (depending on the batch size for Montgomery inversion) than five 131-bit polynomial multiplications and only about 10000 extra bit operations. Up to now, there is no 131-bit polynomial multiplier using fewer than 11000 bit operations; in particular, Bernstein's multiplier [Ber09b] uses 11961 bit operations.

These figures show that polynomial multiplication consumes more than 80% of the bit operations in each iteration. Therefore optimization of the multiplication has a high priority. This section explains how this goal can be achieved.

The importance of avoiding DRAM. Consider an embarrassingly vectorized approach: T threads in a core work on $32T$ independent multiplication problems in bitsliced form. The $32T \times 2$ inputs are stored as 262 vectors of $32T$ bits each, and the $32T$ outputs are stored as 261 vectors of $32T$ bits.

The main difficulty with this approach is that, even if the outputs are perfectly overlapped with the inputs, even if no additional storage is required, the inputs cannot fit into SRAM. For $T = 128$ the inputs consume 134144 bytes, while shared memory and registers together have only 81920 bytes. Reducing T to 64 (and risking severe GPU under-utilization) would fit the inputs into 67072 bytes, but would also make half of the registers inaccessible (since each thread can access at most 128 registers), reducing the total capacity of shared memory and registers to 49152 bytes.

There is more than enough space in DRAM, even with very large T, but DRAM throughput then becomes a serious bottleneck. A single pass through the input vectors, followed by a single pass through the output vectors, keeps the DRAM occupied for $523T$ cycles (i.e., more than 16 cycles per multiplication), and any low-memory multiplication algorithm requires many such passes.

Several implementations of the complete iteration function using different multiplication algorithms achieved at most 26 million iterations per second on a GTX 295 with this approach. The remainder of this section describes a faster approach.

How to fit into shared memory. The SIMD programming model of GPUs highly relies on the exploitation of data-level parallelism. However, data-level parallelism does not require having each thread to work on completely independent computations: parallelism is also available within computations. For example, the addition of two 32-way-bitsliced field elements is nothing but a sequence of

131 32-bit `XOR` operations; it naturally contains 131-way data-level parallelism. Similarly, there are many ways to break 131-bit binary-polynomial multiplication into several smaller-degree polynomial multiplications that can be carried out in parallel.

Registers cannot be used for communication between threads. Having several threads cooperate on a single computation requires the active data for the computation to fit into shared memory. Furthermore, registers offer more space than shared memory; therefore during multiplication some registers can be used as spill locations for data not involved in the multiplication, reversing the traditional direction of data spilling from registers to memory.

The implementation carries out 128 independent 131-bit multiplications (i.e., four 32-way bitsliced 131-bit multiplications) inside shared memory and registers, with no DRAM access. This means that each multiplication has to fit within 1024 bits of shared memory. This is not a problem for schoolbook multiplication, but it is a rather tight fit for a fast Karatsuba-type multiplication algorithm (see below); more simultaneous multiplications would require compromises in the multiplication algorithm.

When using 128 threads, 32 threads are cooperating on each of the four 32-way bitsliced 131-bit multiplications. Experiments confirmed that this number of threads is enough to achieve almost 80% ALU occupancy during the multiplication which is the most time-consuming part of the iteration function. The 131-bit multiplication algorithm allows close to 32-way parallelization, as discussed below, although the parallelization is not perfect.

There would be higher ALU occupancy when using 192 or 256 threads. This would require either to handle more than 128 iterations in parallel and thus raise pressure on fast memory or to increase the parallelization degree within each multiplication. In the opposite direction, the memory demand or the parallelization degree could be reduced by running 96 or 64 threads; but below 128 threads the GPU performance drops drastically. Therefore using 128 threads to compute on 128 independent iterations seems to yield the best performance.

The main task is now to multiply 131-bit polynomials, at each step using 32 parallel bit operations to the maximum extent possible. The resulting algorithm is expanded to 128 independent, bitsliced inputs to obtain code for 128 cooperating threads: performing 128 separate multiplications of 131-bit polynomials, stored in bitsliced form as $4 \cdot 131$ 32-bit words, using 128 concurrent 32-bit operations to the maximum extent possible.

Vectorized 128-bit multiplication. First consider the simpler task of multiplying 128-bit polynomials. This can efficiently be performed by applying three levels of Karatsuba expansion. Each level uses $2n$ `XOR` instructions to expand a $2n$-bit multiplication into three n-bit multiplications, and then $5n-3$ `XOR` instructions to collect the results (with Bernstein's "refined Karatsuba" from [Ber09b]).

Three levels of Karatsuba result in 27 times 16-bit polynomial multiplications. The inputs to these multiplications occupy a total of 864 bits, consuming most but not all of the 1024 bits of shared memory available to each 131-bit multiplication.

The code from [Ber09b] for a 16-bit polynomial multiplication can be scheduled to fit into 67 registers. It is applied to the 27 multiplications in parallel, leaving 5 threads idle out of 32. In total $27 \cdot 4 = 108$ 16-bit polynomial multiplications on 32-bit words are carried out by 108 threads in this subroutine leaving 20 threads idle. Each thread executes 413 instructions (350 bit operations and 63 load/store instructions).

The initial expansion can be parallelized trivially. Operations on all three levels can be joined and performed together on blocks of 16 bits per operand using 8 loads, 19 `XOR` instructions, and 27 stores per thread.

Karatsuba collection is more work: On the highest level (level 3), each block of 3 times 32-bit results (with leading coefficient zero) is combined into a 64-bit intermediate result for level 2. This takes 5 loads (2 of these conditional), 3 `XOR` operations and 3 stores per thread on each of the 9 blocks. Level 2 operates on blocks of 3 64-bit intermediate results leading to 3 128-bit blocks of intermediate results for level 1. This needs 6 loads and 5 `XOR` operations for each of the 3 blocks. The 3 blocks of intermediate results of this step do not need to be written to shared memory and remain in registers for the following final step on level 1. Level 1 combines the remaining three blocks of 128 bits to the final 256-bit result by 12 `XOR` operations per thread.

Vectorized 131-bit multiplication. To multiply 131-bit polynomials, the inputs are split into a 128-bit low part and a 3-bit high part. The 128-bit multiplications of the two low parts are handled by a 128-bit multiplier as described above, the 3×3-bit product of the high parts and the two 3×128-bit mixed products are handled in a straightforward way: The 3×3-bit multiplication can be carried out almost for free by an otherwise idle 16-bit multiplication thread. The 3×128-bit multiplications can be implemented straightforwardly by schoolbook multiplication.

However, some of the additions to obtain the final result can be saved and the code can be shaped more streamlined by distributing the computations in the following way: The 5 most-significant bits of the final result only depend on the 5 most-significant bits of each input. Thus they can be obtained by computing the product of these bits (using the 16-bit multiplier) and by cutting off the 4 least-significant bits of the 9 resulting bits. Now the 2 most significant bits of the results from the 3×128-bit multiplications do not need to be computed and summed up anymore; the 128 least significant bits of the 3×128-bit multiplications can be obtained each by 6 loads for the 3 highest bits of each input, $3 \cdot 128$ combined load-`AND` instructions per input, and $2 \cdot 128$ `XOR` instructions (some of them masked for the 2 least-significant bits).

Overall the multiplier uses 13087 bit operations, and about 40% of the ALU cycles are spent on these bit operations rather than on loads, stores, address calculations, and other overhead. An extra factor of about 1.1 is lost from 32-way parallelization, since the 32 threads are not always all active. For comparison, the Toom-type techniques from [Ber09b] use only 11961 bit operations, saving about 10%, but appear to be more difficult to parallelize.

3.4.4 ECC2K-130 iterations on the GPU

This section discusses several aspects of the overhead in the ECC2K-130 computation. The main goal, as in the previous section, is to identify 32-way parallelism in the bit operations inside each 131-bit operation. This is often more challenging for the "overhead" operations than it is for multiplication, and in some cases the algorithms are modified to improve parallelism. All of these operations work entirely in shared memory.

Basis conversion. As explained in Section 3.4.2 most of the elements of $\mathbb{F}_{2^{131}}$ are represented in (permuted) normal basis. Before those elements are multiplied, they are converted from normal basis to polynomial basis. Consider an element a of $\mathbb{F}_{2^{131}}$ in (permuted) normal basis:

$$a = a_0(\zeta + \zeta^{-1}) + a_1(\zeta^2 + \zeta^{-2}) + \cdots + a_{130}(\zeta^{131} + \zeta^{-131}).$$

On the first two levels of the basis conversion algorithm the following sequence of operations is executed on bits a_0, a_{62}, a_{64}, a_{126}:

$$\begin{aligned} a_{62} &\leftarrow a_{62} + a_{64} \\ a_0 &\leftarrow a_0 + a_{126} \\ a_{64} &\leftarrow a_{64} + a_{126} \\ a_0 &\leftarrow a_0 + a_{62}. \end{aligned}$$

Meanwhile the same operations are performed on bits a_1, a_{61}, a_{65}, a_{125}; on bits a_2, a_{60}, a_{66}, a_{124}; and so on through a_{30}, a_{32}, a_{94}, a_{96}. These 31 groups of bits are assigned to 32 threads, keeping almost all of the threads busy.

Merging levels 2 and 3 and levels 4 and 5 works in the same way. This assignment keeps 24 out of 32 threads busy on levels 2 and 3, and 16 out of 32 threads busy on levels 4 and 5. This assignment of operations to threads also avoids almost all memory-bank conflicts.

Multiplication with reduction. Recall that the PPP multiplication produces a product in polynomial basis, suitable for input to a subsequent multiplication. The PPN multiplication produces a product in normal basis, suitable for input to a squaring.

The main work in PPN, beyond polynomial multiplication as described in Section 3.4.3, is a conversion of the product from polynomial basis to normal basis. This conversion is almost identical to basis conversion described above, except that it is double-size and in reverse order. The reduction PPP is a more complicated double-size conversion, with similar parallelization.

Squaring and m-squaring. Squaring and m-squaring are simply permutations in normal basis, costing 0 bit operations, but this does not mean that they cost 0 cycles.

The obvious method for 32 threads to permute 131 bits is for them to pick up the first 32 bits, store them in the correct locations, pick up the next 32 bits, store them in the correct locations, etc.; each thread performs 5 loads and 5 stores, with most of the threads idle for the final load and store. The addresses determined by the permutation for different m-squarings can be kept in constant memory. However, this approach triggers two GPU bottlenecks.

The first bottleneck is shared-memory bank throughput. Recall from Section 3.4 that threads in the same half-warp cannot simultaneously store values to the same memory bank. To almost completely eliminate this bottleneck a greedy search is performed that finds a suitable order to pick up 131 bits, trying to avoid all memory bank conflicts for both the loads and the stores. For almost all values of m, including the most frequently used ones, this approach finds a conflict-free assignment. For two values of m the assignment involves a few bank conflicts, but these values are used only in inversion, not in the main loop.

The second bottleneck is the throughput of constant cache. Constant cache delivers only one 32-bit word per cycle; this value can be broadcasted to all threads in a warp. If the threads load from different positions in constant memory, then these accesses are serialized. To eliminate this bottleneck, the loads are moved out of the main loop. Each thread reserves 10 registers to hold 20 load and 20 store positions for the 4 most-often used m-squarings, packing 4 1-byte positions in one 32-bit register. Unpacking the positions costs just one shift and one mask instruction for the two middle bytes, a mask instruction for the low byte, and a shift instruction for the high byte.

Addition. The addition of 128 bitsliced elements of $\mathbb{F}_{2^{131}}$ is decomposed into computing the `XOR` of two sets of $4 \cdot 131 = 524$ of 32-bit words. This can be accomplished by 128 threads using $2 \cdot 5$ loads, 5 `XOR` operations and 5 stores per thread where 2 loads, 1 `XOR` and 1 store are conditional and carried out by only 12 threads.

Hamming-weight computation. The subroutine for Hamming-weight computation receives elements of $\mathbb{F}_{2^{131}}$ in bitsliced normal-basis representation as input and returns the Hamming weight, i.e. the sum of all bits of the input value, as bitsliced output. More specifically, it returns 8 bits $h_0, \ldots, h_7$ such that the Hamming weight of the input value is $\sum_{i=0}^{7} h_i 2^i$.

The basic building block for the parallel computation is a full adder, which has three input bits b_1, b_2, b_3 and uses 5 bit operations to compute 2 output bits c_0, c_1 such that $b_1 + b_2 + b_3 = c_1 2 + c_0$. At the beginning of the computation all 131 bits have a weight of 2^0. When the full adder operates on bits of weight 2^0, output bit c_1 gets a weight of 2^1 and the bit c_0 a weight of 2^0. If three bits with a weight of 2^1 are input to a full adder, output bit c_1 will have the weight 2^2, bit c_0 weight 2^1. More generally: If three bits with a weight of 2^i enter a full adder, output bit c_1 will have a weight of 2^{i+1}, output bit c_0 a weight of 2^i. The full adder sums up bits of each weight until only one bit of each weight is left giving the final result of the computation.

Because there are many input bits, it is easy to keep many threads active in parallel. In the first addition round 32 threads perform 32 independent full-adder operations, 96 bits with weight 2^0 are transformed into 32 bits with weight 2^0 and

32 bits with weight 2^1. This leaves $131 - 96 + 32 = 67$ bits of weight 2^0 and 32 bits of weight 2^1. In the second round, 22 threads pick up 66 bits of weight 2^0 and produce 22 bits of weight 2^0 and 22 bits of weight 2^1. At the same time 10 other threads pick up 30 bits of weight 2^1 and produce 10 bits of weight 2^1 and 10 bits of weight 2^2. This leaves $67-66+22 = 23$ bits of weight 2^0, $32-30+22+10 = 34$ bits of weight 2^1, and 10 bits of weight 2^2.

Following this approach, it takes at least 13 rounds to compute the bits $h_0, \ldots, h_7$, i.e. 8 bits with weight $2^0, \ldots, 2^7$. The implementation actually uses a somewhat less parallel approach with 21 rounds, two of these rounds being half-adder operations which receive only 2 input bits and take only 2 bit operations. This has the benefit of simplifying computation of the input positions as a function of the thread ID.

Once the Hamming weight is computed, it can be tested whether the weight is below 34, i.e., whether a distinguished point has been reached. Furthermore, the Hamming weight is needed to compute j.

3.4.5 Overall results on the GPU

GPU code is organized into kernels called from the CPU. Launching a kernel takes several microseconds on top of any time needed to copy data between global memory and the CPU. To eliminate these costs, the kernel runs for several seconds. The kernel consists of a loop around a complete iteration; it performs the iteration repeatedly without contacting the CPU. Any distinguished points are masked out of subsequent updates; distinguished points are rare, so negligible time is lost computing unused updates.

The high costs for inversions are alleviated by using Montgomery's trick as described in Section 3.2: Only one inversion is used for a batch of iterations. Therefore, a batch of 128 iterations is streamed in a simple way between global memory and shared memory; this involves a small number of global-memory copies in each iteration. Spilling of any additional data to DRAM is totally avoided; in particular, the kernel refrains from using local memory.

All of this sounds straightforward but in fact required a complete redesign of NVIDIA's programming environment. As explained in Section 2.3, NVIDIA's register allocator was designed to handle small kernels consisting of hundreds of instructions. For medium-size kernels the code produced by NVIDIA's compiler involved frequent spills to local memory, dropping performance by an order of magnitude. For larger kernels the compiler ran out of memory and eventually crashed.

An early attempt to implement the iteration function was to write the code in NVIDIA's PTX language. As described in Section 2.3, this language does not use hardware registers but register variables. Register allocation is performed by NVIDIA's PTX compiler—and this compiler turns out to be the culprit in NVIDIA's register-allocation problems. To control register allocation, eventually the whole kernel was implemented using the reverse-engineered assembler cudasm by van der Laan [Laa07] and the new tool chain introduced in Section 2.3 with m5

building block	**normal basis cycles**
multiplication	
PPN	159.54
PPP	158.08
squaring / m-squaring	9.60
computation of σ^j	44.99
Hamming-weight computation	41.60
addition	4.01
inversion	1758.72
conversion to polynomial basis	12.63
full iteration:	
$B = 128$	1164.43

Table 3.2: Cycle counts per input for all building blocks on one core of a GTX 295. Cycle counts for 128 bitsliced inputs are divided by 128. The value B in the last row denotes the batch size for Montgomery inversions.

and qhasm on top of cudasm. Furthermore a function-call convention was designed on the assembly level. Using this convention, large stretches of instructions were merged into functions to reduce code size and thus instruction-cache misses.

Table 3.2 reports timings of all major building blocks on the GPU. These numbers were collected during a typical pass through the main iteration loop by reading the GTX 295's hardware cycle counter. As in case for the Cell processor, the cycle counts have been divided by the bitslicing width of 128 independent iterations.

The complete kernel uses 1164.43 cycles per iteration on average on a single core of a GTX 295 graphics card. Therefore the 60 cores of the whole card achieve about 64 million iterations per second at a clock speed of 1.242 GHz. The whole ECC2K-130 computation would be finished in one year (on average; Pollard's rho method is probabilistic) using 1066 GTX 295 graphics cards.

As stated in Section 3.3.5, the ECC2K-130 computation is still ongoing at time of this writing. The GPU implementation has been running on the GPU clusters Lincoln at the National Center for Supercomputing Applications (NCSA) and Longhorn at the Texas Advanced Computing Center (TACC) as part of the former TeraGrid initiative. Furthermore it has been running on the AC GPU cluster at NCSA and on the GPU Cluster at the SARA Computing and Networking Services of the Netherlands National Computing Facilities foundation (NCF).

3.5 Performance comparison

NVIDIA's graphics card GTX 295 computes about 64 million iterations per second. That is almost 2.5 times faster than IBM's Cell processor which performs about 25.88 million iterations per second. Nevertheless, judging from the raw instruction throughput, the graphics card should perform much better than that: The 60 cores of a GTX 295 deliver up to 596.16 billion logical operations on 32-bit words per second. This is more than 7.5 times the performance of the 6 SPEs of a Cell processor in the PlayStation 3 delivering 76.8 billion logical operations on 32-bit words per second. There are several reasons why the Cell processor performs more efficiently than the GPU:

- The SPEs of the Cell processor can execute one memory operation in parallel to each logical instruction while the GPU has to spend extra cycles on loading and storing data.
- The memory controller of the SPE can transfer data between local storage and main memory per DMA in parallel to the computation of the ALU; setting up data transfers costs only a fraction of an instruction per transferred byte. On the GPU, memory transfers between device memory and shared memory take a path through the ALU, consuming additional cycles.
- The local storage of the SPEs is large enough to keep the full working set and all code during computation while the instruction cache of the GPU is too small for the code size.
- For the GPU implementation, several threads are cooperating on the bit operations of one iteration. This requires synchronization and communication which adds more cycles to the overall cycle count.
- Due to the cooperation of threads during the computation, not all threads of the GPU can be kept busy all the time.
- Sequential computations like address calculations are more frequent on the GPU due to the thread-based programming model.
- More than 128 threads would need to run concurrently to hide all latency effects which arise on the GPU.

Some of these issues could have been avoided by increasing the number of concurrent threads on each GPU core. However, the multiplication which is the most expensive instruction requires an amount of resources that is only available when using at most 128 threads. Any reduction of the resources would result in less efficient multiplication algorithms and thus eat up all improvements achieved by a higher ALU occupancy.

To put the results into perspective, similar implementations of the iteration function are presented in [BBB$^+$09]: A Core 2 Extreme Q6850 CPU with 4 cores running at 3 GHz clock frequency achieves 22.45 million iterations/second. A previous, more straightforward GPU implementation that was not sharing work between the threads, carries out 25.12 million iterations/second on the same GTX 295 graphics cards. A Spartan-3 XC3S5000-4FG676 FPGA delivers 33.67 million iterations/second. A $16mm^2$ ASIC is estimated to achieve 800 million iterations/second.

4

Parallel implementation of the XL algorithm

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 [YCB$^+$07].

This chapter describes 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 [CKP$^+$00]; 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. To achieve a better understanding of the practical behaviour of XL, this chapter describes 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 8 nodes and shared memory systems of up to 48 cores.

The content of this chapter is joined work with Ming-Shing Chen, Chen-Mou Cheng, Tung Chou, Yun-Ju Huang, and Bo-Yin Yang. This research has been supported by the Netherlands National Computing Facilities foundation (NCF) as project MP-230-11. A paper about the implementation with more details on the comparison between XL and F_4/F_5 is going to be finished in 2012.

This chapter is structured as follows: The XL algorithm is introduced in Section 4.1. Section 4.2 explains Coppersmith's block Wiedemann algorithm which is used for solving the linearized system. Sections 4.3 and 4.4 introduce variations of the Berlekamp–Massey algorithm that are used as building block for Coppersmith's block Wiedemann algorithm: Section 4.3 describes Coppersmith's version and Section 4.4 introduces an alternative algorithm invented by Thomé. An implementation of XL using the block Wiedemann algorithm is described in Section 4.5. Section 4.6 gives runtime measurements and performance values that are achieved by this implementation for a set of parameters on several parallel systems.

Notations: In this chapter 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 chapter use the bound for simple matrix multiplication $O(a\cdot b\cdot c)$ as upper bound for the asymptotic time complexity of matrix multiplications.

4.1 The XL algorithm

The original description of XL for multivariate quadratic systems can be found in [CKP$^+$00]; 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 the 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 can be computed by writing each of these monomials $x^b = x_1^{b_1} x_2^{b_2} \ldots x_n^{b_n}$ as a string

$$\text{“}\underbrace{\text{foo foo} \ldots \text{foo}}_{b_1\text{-many}}\ x_1\ \underbrace{\text{foo foo} \ldots \text{foo}}_{b_2\text{-many}}\ x_2\ \ldots\ x_n\ \underbrace{\text{foo foo} \ldots \text{foo}}_{(D-2-|b|)\text{-many}}\text{”}.$$

This string has $n+(D-2)$ words, n times “x_i” and $D-2$ times “foo”. The number of all such strings is $\binom{n+(D-2)}{n} = \binom{n+(D-2)}{D-2}$. Thus the number $|\mathcal{T}^{(D-2)}|$ of all monomials with total degree less than or equal to $D-2$ is $\binom{n+(D-2)}{n}$. 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. The solution for this dilemma is given by Yang and Chen in [YC05] (see also Moh in [Moh01] and Diem in [Die04]): they show that for random systems the minimum degree D_0 required for the reliable termination of XL is given by $D_0 := \min\{D : ((1-\lambda)^{m-n-1}(1+\lambda)^m)[D] \leq 0\}$.

4.2 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 [Mon95] and the block Wiedemann algorithm [Cop94]. The block Wiedemann algorithm was proposed by Coppersmith in 1994 and is a generalization of the original Wiedemann algorithm [Wie86]. It 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 chapter. The following paragraphs give a review of these three steps on an operational level; for more details please see [Cop94].

BW1: Given an input matrix $B \in K^{N\times N}$, parameters $m, n \in \mathbb{N}$ with $m \geq n$, and $\kappa \in \mathbb{N}$ of size $N/m + N/n + O(1)$, the first step BW1 computes the first κ 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 this chapter the quotient $\lceil m/n \rceil$ is treated 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 "matrix Berlekamp–Massey" algorithm or "block Berlekamp–Massey" algorithm in analogy to the name "block Wiedemann".

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 $N/m + N/n + O(1)$ with coefficients in $K^{n\times m}$ as

$$a(\lambda) = \sum_{i=0}^{\kappa} a^{(i)}\lambda^i.$$

```
input  : $H^{(j)} \in K^{(m+n)\times m}$
         a list of nominal degrees $d^{(j)}$
output: $P^{(j)} \in K^{(m+n)\times(m+n)}$
        $E^{(j)} \in K^{(m+n)\times(m+n)}$
$M \leftarrow H^{(j)}$, $P \leftarrow I_{m+n}$, $E \leftarrow I_{m+n}$;
sort the rows of $M$ by the nominal degrees in decreasing order and apply
the same permutation to $P^{(j)}$ and $E^{(j)}$;
for $k = 1 \rightarrow m$ do
    for $i = (m+n+1-k)$ downto $1$ do
        if $M_{i,k} \neq 0$ then
            $v_{(M)} \leftarrow M_i$, $v_{(P)} \leftarrow P_i$, $v_{(E)} \leftarrow E_i$;
            break;
    for $l = i+1$ to $(m+n+1-k)$ do
        $M_{l-1} \leftarrow M_l$, $P_{l-1} \leftarrow P_l$, $E_{l-1} \leftarrow E_l$;
    $M_{(m+n+1-k)} \leftarrow v_{(M)}$, $P_{(m+n+1-k)} \leftarrow v_{(P)}$, $E_{(m+n+1-k)} \leftarrow v_{(E)}$;
    for $l = 1$ to $(m+n-k)$ do
        if $M_{l,k} \neq 0$ then
            $M_l \leftarrow M_l - v_{(M)} \cdot (M_{l,k}/v_{(M)k})$;
            $P_l \leftarrow P_l - v_{(P)} \cdot (M_{l,k}/v_{(M)k})$;
$P^{(j)} \leftarrow P$;
$E^{(j)} \leftarrow E$;
```

Algorithm 1: Gaussian elimination in Coppersmith's Berlekamp–Massey algorithm

The j-th iteration step receives two inputs from the previous iteration: One input is an $(m+n)$-tuple of polynomials $(f_1^{(j)}(\lambda), \dots, f_{m+n}^{(j)}(\lambda))$ 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* $(d_1^{(j)}, \dots, d_{m+n}^{(j)})$; 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 $f_{m+i}^{(j_0)}$, $1 \leq i \leq n$, 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 $f_{m+i}^{(j_0)}[k] = 0$, $k \neq j_0$. Repeat choosing the polynomials $f_1^{(j_0)}, \dots, f_m^{(j_0)}$ 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 $0 \leq i \leq (m+n)$.

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

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

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 $(f_1(\lambda), \dots f_n(\lambda))$ with coefficients in $K^{1\times n}$ and an n-tuple $d = (d_1, \dots, d_n)$ 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 $(f_1(\lambda), \dots f_n(\lambda))$ with coefficients in $K^{1\times n}$ and an n-tuple $d = (d_1, \dots, d_n)$ as input from BW2. For each $f_i(\lambda)$, $1 \leq i \leq n$, compute $w_i \in K^N$ as

$$\begin{aligned} w_i &= z(f_i[\deg(f_i)])^T + B^1 z(f_i[\deg(f_i) - 1])^T + \dots + B^{\deg(f_i)} z(f_i[0])^T \\ &= \sum_{j=0}^{\deg(f_i)} B^j z(f_i[\deg(f_i) - j])^T. \end{aligned}$$

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 $B^{(k_i+1)} w_i = 0$, $0 \leq k_i \leq (d_i - \deg(f_i))$. 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; Vil97b; Vil97a].

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:

$$W^{(j)} = \begin{cases} z \cdot (f[0])^T & \text{for } j = 0, \\ z \cdot (f[j])^T + B \cdot W^{(j-1)} & \text{for } 0 < j \leq \deg(f), \\ B \cdot W^{(j-1)} & \text{for } \deg(f) < j \leq \max(d). \end{cases}$$

The kernel vectors of B are found during the iterative computation of $W^{(\max(d))}$ by checking whether an individual column $i \in \{1, \dots, n\}$ 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 asymptotically time complexity

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

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 consist of up to n kernel vectors of B.

4.3 The block Berlekamp–Massey algorithm

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

4.3.1 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 of $\phi^{(j)}$ 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 $P^{(j)}$; some of these operations are permutations of rows. The composition of these permutations is $E^{(j)}$:

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

The algorithm by Coppersmith requires that the first n rows of $P^{(j)}H^{(j)}$ are all zero (see [Cop94, p. 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.

4.3.2 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

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

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 by at most 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.

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
6	{7,6}	{5,4}	{3,2}	{1,0}	7
...	...	...	...	...	...

Table 4.1: Example for the workload distribution over 4 nodes. Iteration 0 receives the distribution in the first line as input and computes the new distribution in line two as input for iteration 1.

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 approximate $\deg(f^{(j)})$. 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 $\max(d^{(j+1)}) = \max(d^{(j)}) + 1$, the number $i^{(j)}$ 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 4.1 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. $k \in S_{(i)}^{(j)}$ and $(k-1) \in S_{(i)}^{(j)}$. Thus, $f^{(j+1)}[k]$ can be computed locally without communication. In the example in Figure 4.1, 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 $k - 1 = \max(S_{i-1}^{(j+1)}) = \max(S_{i-1}^{(j)})$, $i \neq 1$, 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 4.1. This case occurs if in iteration $j+1$ no coefficient is reassigned to node $i-1$ due to load balancing.

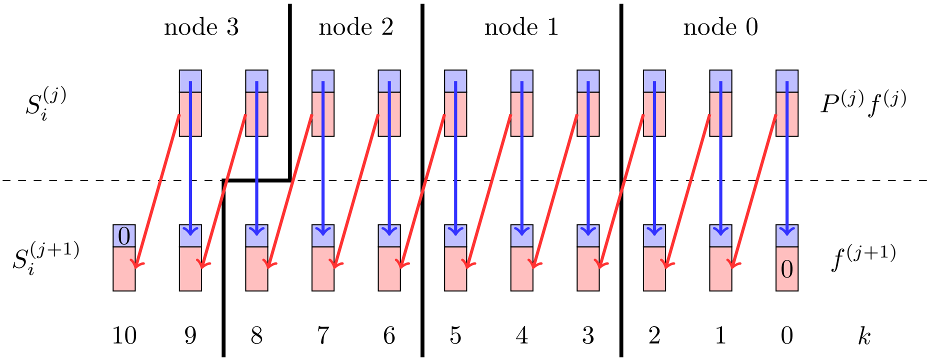

Figure 4.1: Example for the communication between 4 nodes. The top n rows of the coefficients are colored in blue, the bottom m rows are colored in red.

- In case $k = \min(S_i^{(j)}) = \max(S_{i-1}^{(j+1)})$, $i \neq 1$, the top n rows of $(P^{(j)}f^{(j)})[k]$ are sent from node i to node $i-1$. The example in Figure 4.1 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.

4.4 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.

4.4.1 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.

4.4.2 Parallelization of Thomé's Berlekamp–Massey algorithm

Thomé's Berlekamp–Massey algorithm uses multiplication of large matrix polynomials and Coppersmith's Berlekamp–Massey algorithm as building blocks. The parallelization of Coppersmith's version has already been explained. 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 the Macaulay matrix B is very large, the runtime of BW2 is 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.

4.5 Implementation of XL

This section gives an overview of the implementation of XL. Section 4.5.1 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 4.5.2 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 makes it possible to easily extend the implementation to arbitrary fields. Section 4.5.3 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 4.5.4 details how the Macaulay matrix multiplication in the stages BW1 and BW3 is performed efficiently, Section 4.5.5 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.

4.5.1 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 $O(N^2 w_B + Nmw_x)$.

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; Vil97b; Vil97a] do no longer apply.

4.5.2 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 $b_0, \ldots, b_3 \in \{0, 1\}$ where each field element $b \in \mathbb{F}_{16}$ is represented as $b = \sum_{i=0}^{3} b_i x^i$. 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 is implemented using a small number of bit-operations, e.g., multiplication by x is 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

```
INPUT  GF(16) vector A
OUTPUT A * x

mask_a3      = 10001000|10001000|10001000| ...
mask_a2a1a0 = 01110111|01110111|01110111| ...

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
```

Listing 4.1: Pseudocode for scalar multiplication by x.

in parallel using 7 bit-operations as shown in Listing 4.1. Similar computations give the multiplication by x^2 and x^3 respectively.

Multiplying a vector $a \in \mathbb{F}_{16}^{32}$ by an arbitrary scalar value $b \in \mathbb{F}_{16}$ is decomposed to adding up the results of $a \cdot x^i, i \in [0,3]$ for all bits b_i of b that are set to 1:

$$c = a \cdot b = \sum_{i=0}^{3} a \cdot x^i \cdot b_i, \quad c \in \mathbb{F}_{16}^{32}.$$

The number of bit-operations varies with the actual value of b since it is not necessary to explicitly compute $a \cdot x^i$ in case bit i of b is 0.

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. Using this instruction, scalar multiplication is 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 elements of $\mathbb{F}_{16}$, 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 vector register of the input contains 32 packed elements, two `PSHUFB` instructions are used. The input is unpacked using shift and mask operations accordingly as shown in Listing 4.2. Using the `PSHUFB` instruction, the scalar multiplication needs 7 operations for any input value b with the extra cost of accessing the lookup table. The lookup table consumes 256 bytes of memory.

```
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
```

Listing 4.2: Pseudocode for scalar multiplication using `PSHUFB`.

4.5.3 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.

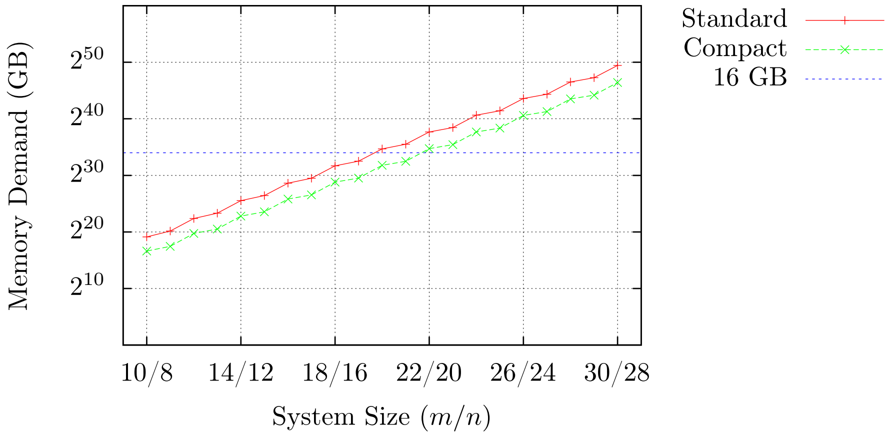

Figure 4.2: Memory demand of XL for several system sizes using $\mathbb{F}_{16}$ in standard and compact representation.

Therefore, it is possible to save memory 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. Finally, 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 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 makes it possible 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 chapter.

Figure 4.2 shows a graph of the memory demand for several systems with m equations and $n = m - 2$ variables 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 amount of memory is necessary to store precomputed column positions. As long as the target architecture offers a sufficient amount of memory, it is more efficient to store the values instead of recomputing them on demand.

```
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
```

Listing 4.3: Top-level iteration loop for BW1.

4.5.4 Macaulay matrix multiplication in XL

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

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

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 (see Section 4.2).

Listings 4.3 and 4.4 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

```
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
```

Listing 4.4: Top-level iteration loop for BW3.

up all row-vectors of t_{old} which are to be multiplied by the same field element and performing 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 ought to be 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 the matrix 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 [Sch11b, Statement 8], and for the addition of 0, see [Pet11b, 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 at 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 cheap (rather slightly cheaper) since the coefficients already have been decomposed into the polynomial components which gives low-cost SIMD multiplication code in either case.

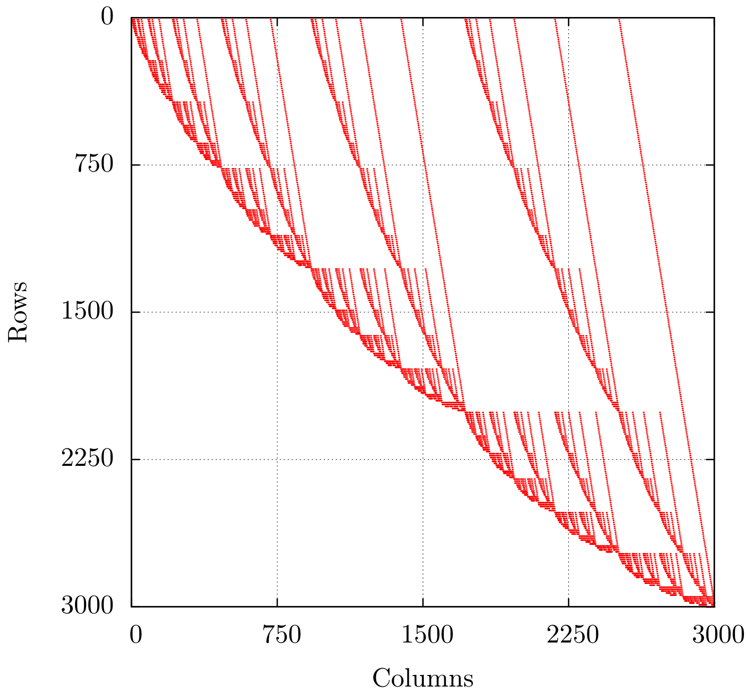

Figure 4.3: Plot of a Macaulay matrix over $\mathbb{F}_{16}$, 8 variables, 10 equations. Each row has 45 entries, 1947 rows have been dropped randomly to obtain a square matrix.

4.5.5 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 4.3 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.

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 makes it possible 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 row ranges to be split 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.

Parallelization for cluster systems: MPI and Infiniband verbs

On a cluster system, the workload is distributed similar to OpenMP by splitting the Macaulay matrix into blocks of rows. The computation on one row block of the Macaulay matrix depends on many rows of matrix t_{old}. A straightforward approach is to make the full matrix t_{old} available on all cluster nodes. This can be achieved by a blocking all-to-all communication step after each iteration step of stages BW1 and BW3.

If B were a dense matrix, such 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. Today's high-performance network interconnects are able to transfer data via direct memory access (DMA) without interaction with 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.

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 either avoid all communication during stages BW1 and BW3 or perform 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:

a) **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 sent 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 transferred, 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.

b) **Operating on two shared column blocks of t_{old} and t_{new}:**
One can avoid both small packet size and blocking communication steps 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} requires 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.

c) **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 B 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 c) might be the most efficient one although it loses efficiency due to the effect of the width of t_{old} and t_{new}. The performance of approach a) and approach b) depends heavily on the network configuration and the ratio between computation time and communication time. For these two approaches, also the structure of the Macaulay matrix accounts for a 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 number of rows to each cluster node.

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.

Approach b) is implemented in two ways: The first implementation uses MPI. After computation on one column block has been finished, the results are sent to all other nodes using the `MPI_Iallgather` function. During computation on the next column block, the `MPI_Test` function is called periodically 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 a performance penalty.

However, looking at the structure of the Macaulay matrix (an example is shown in Figure 4.3) one can observe that this communication scheme performs much more communication than necessary. For example on a cluster of four computing nodes, node 0 computes the top quarter of the rows of matrix $t_{new,0}$ and $t_{new,1}$. Node 1 computes the second quarter, node 2 the third quarter, and node 3 the bottom quarter. Node 3 does not require any row that has been computed by node 0 since the Macaulay matrix does not have entries in the first quarter of the columns for these rows. The obvious solution is that a node i sends only these rows to a node j that are actually required by node j in the next iteration step. Depending on the system size and the cluster size, this may require to send many separate data blocks to some nodes. This increases the communication overhead and requires to call `MPI_Test` even more often.

Therefore, to reduce the communication overhead and communication time, the second implementation of approach b) circumvents the MPI API and programs the network hardware directly. This implementation uses an InfiniBand network; the same approach can be used for other high-performance networks. The InfiniBand hardware is accessed using the InfiniBand verbs API. Programming the InfiniBand cards directly has several benefits: All data structures that are required for communication can be prepared offline; initiating communication requires only one call to the InfiniBand API. The hardware is able to perform all operations for sending and receiving data autonomously after this API call; there is no need for calling further functions to ensure communication progress as it is necessary when using MPI. Finally, complex communication patterns using scatter-gather lists for incoming and outgoing data do not have a large overhead. This implementation allows to send only such rows to the other nodes that are actually required for computation with a small communication overhead. This reduces communication to the smallest amount possible for the cost of only a negligibly small initialization overhead. One disadvantage of this approach is an unbalanced communication demand of the nodes. Another disadvantage is that the InfiniBand verbs API is much more difficult to handle than MPI.

Figure 4.4 shows the communication demand of each node for both implementations. The figure shows the values for a quadratic system of 18 equations and 16 variables; however, the values are qualitatively similar for different parameter choices. While for the MPI implementation the number of rows that are sent is the same for all nodes, it varies heavily for the InfiniBand verbs API implementation. The demand on communication is increased for large clusters for the MPI case; for the InfiniBand case, communication demand has a peak for 4 nodes and declines afterwards. However, the scalability of the InfiniBand approach depends on the ratio between computation time and communication time. Perfect scalability is only achieved as long as computation time is longer than communication time. While computation time is roughly halved when doubling the number of nodes, communication time decreases in a smaller slope. Therefore, at some point for a certain number of nodes, computation time is catching up with communication time. For moderate system sizes on a cluster with a fast InfiniBand network, the MPI implementation scales almost perfectly for up to four nodes while the InfiniBand verbs API implementation scales almost perfectly for up to eight nodes (details are discussed in the following Section 4.6). A better scalability for large problem size is not likely: On the one hand, larger systems have a higher row weight and therefore require more computation time per row. But on the other hand, a higher row weight also increases the amount of rows that need to be communicated.

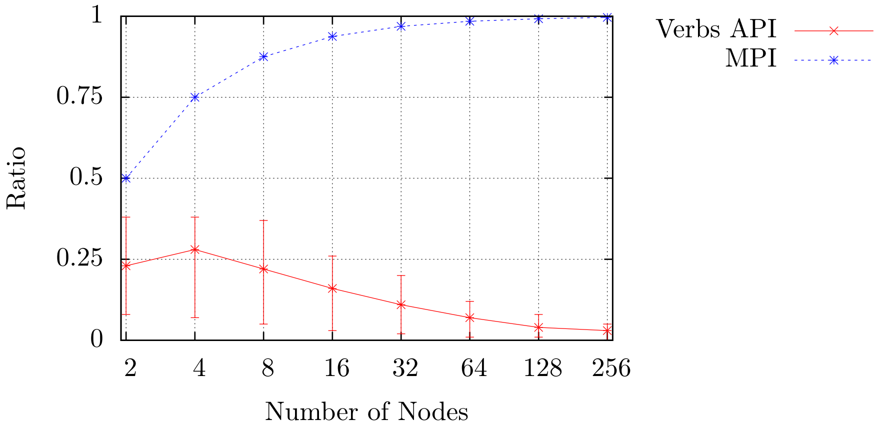

Figure 4.4: Ratio between the number of rows that each node sends in one iteration step and the total number of rows N. For MPI, the number of rows that are sent is equal for all cluster nodes. For InfiniBand, the number of rows varies; the maximum, minimum and average is shown.

Approach c) obviously does not profit from programming the InfiniBand cards directly—since it does not require any communication. Currently, there is only an MPI version of approach a). This approach would profit from a more efficient communication strategy; providing such an implementation is yet an open task.

All parallelization approaches stated above are based on the memory-efficient Macaulay matrix representation described in Section 4.5.3. Alternatively the compact data format can be dropped in 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, the Macaulay matrix could be partitioned using the *Mondriaan partitioner* of Bisseling, Fagginger Auer, Yzelman, and Vastenhouw available at [BFAY$^+$10]. Due to the low row-density, a repartitioning of the Macaulay matrix may reduce the communication demand and provide a better communication scheme. On the other hand, the Mondriaan partitioner performs well for random sparse matrices, but the Macaulay matrix is highly structured. Therefore, it is not obvious if the communication demand can be reduced by such an approach. First experiments with the Mondriaan partitioner did not yet yield lower communication cost. Another approach for reducing communication cost is to modify the Macaulay matrix systematically. Currently the terms of the linearized system are listed in graded reverse lexicographical (grevlex) monomial order. For XL, the choice of the monomial order is totally free. Other monomial orders may give a lower communication cost and improve caching effects. Analyzing the impact of the Mondriaan partitioner, choosing a different monomial order, and evaluating the trade-off between communication cost, memory demand, and computational efficiency is a major topic of future research.

	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
Number of cluster nodes	—	8
Total number of cores	48	64
Network interconnect	—	InfiniBand 40 GB/s
Memory		
Memory per CPU	32 GB	18 GB
Memory per cluster node	—	36 GB
Total memory	256 GB	288 GB

Table 4.2: Computer architectures used for the experiments.

4.6 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 an eight node InfiniBand cluster. Table 4.2 lists the key features of these systems. The computers are located at the Institute for Information Technology of the Academia Sinica in Taipei, Taiwan.

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

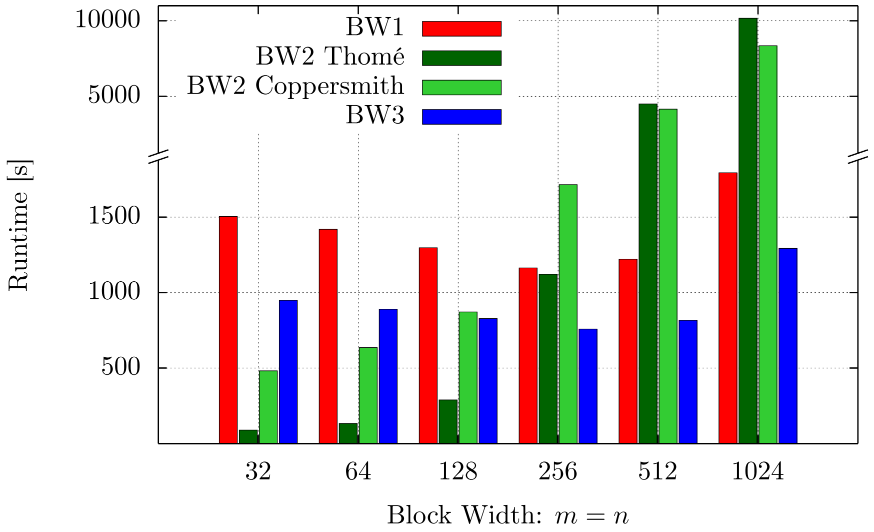

Figure 4.5: Runtime of XL 16-14 on one cluster node with two CPUs (8 cores in total) with different block sizes.

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.

4.6.1 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 4.5 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

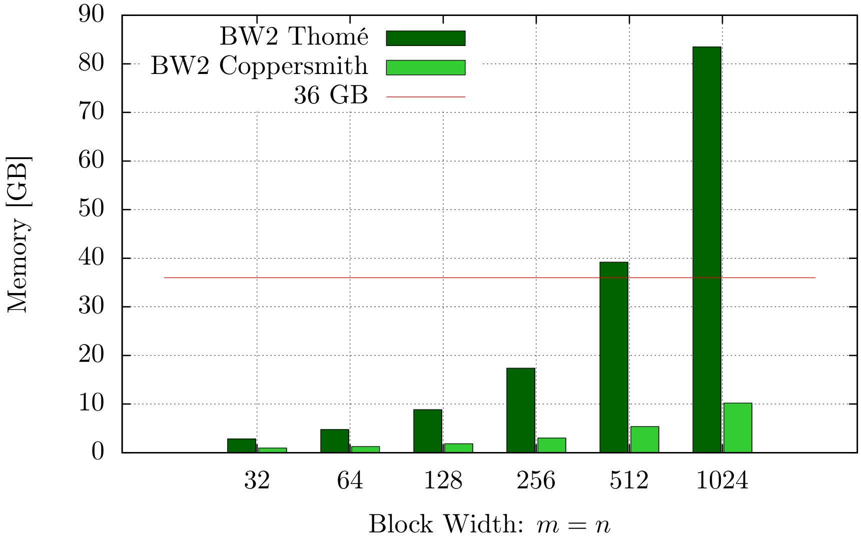

Figure 4.6: Memory consumption of XL 16-14 on a single cluster node with 36 GB RAM.

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 4.5 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 4.6 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 version of BW2.

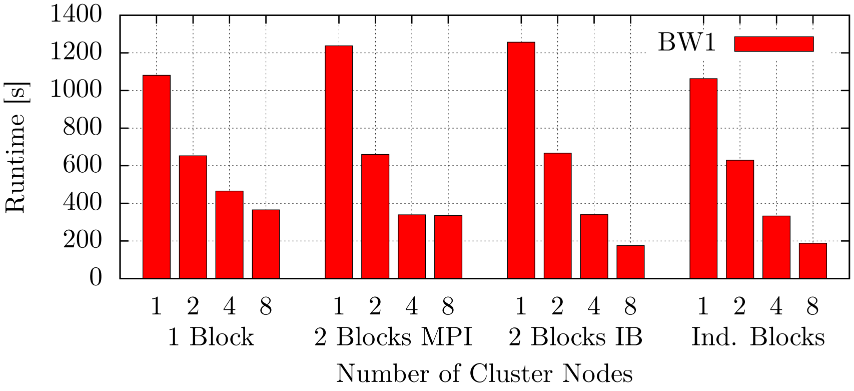

Figure 4.7: Runtime of stage BW1 for XL 16-14 on 1, 2, 4, and 8 cluster nodes using 8 cores per node.

4.6.2 Performance of the Macaulay matrix multiplication

This experiment investigates which of the approaches given in Section 4.5.5 for cluster parallelization of the Macaulay matrix multiplication gives the best performance. 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 computed as $E_p = T_1/pT_p$. Figure 4.7 shows the runtime of BW1 for each of the approaches on one to eight cluster nodes, Figure 4.8 shows parallel speedup and parallel efficiency. The same system size was used as in the previous experiment. Since this experiment is only concerned about the parallelization of stage BW1 and not about the performance of stage BW2, a block size of $m = n = 256$ was used for all these experiments. Similar results apply for stage BW3.

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 while computation continues. This approach is called “1 Block” in Figure 4.7 and 4.8. 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. For $n = 256$, each of the two column blocks has a width of 128 elements. This approach has two implementations. The first implementation uses MPI for communication; it is called “2 Blocks MPI” in the figures. The second implementation uses the InfiniBand verbs API; it is called “2 Blocks IB” in the figures. 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 128 for 2 cluster nodes, 64 for 4 cluster nodes, and 32 for 8 cluster nodes due to the fixed $n = 256$. This approach is called “Ind. Blocks” in the figures.

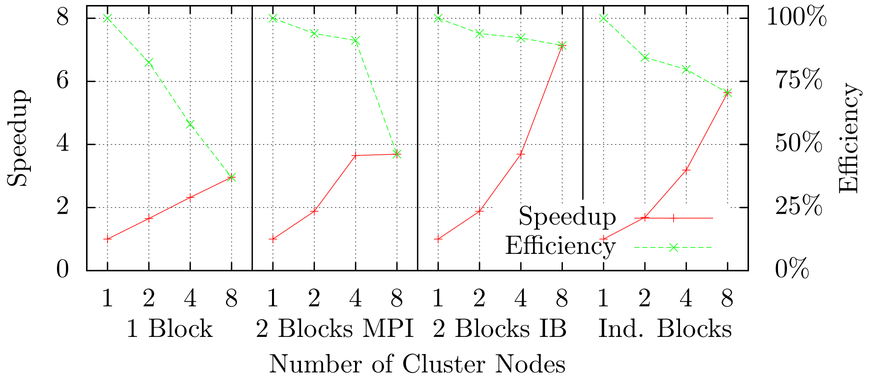

Figure 4.8: Speedup and efficiency of stage BW1 for XL 16-14 on 1, 2, 4, and 8 cluster nodes using 8 cores per node.

Since the approaches “2 Blocks MPI” and “2 Blocks IB” split the data into two column blocks independently of the number of computing nodes, they have some overhead when computing on a single cluster node. Therefore, for 1 cluster node the runtime is longer than for the other two approaches.

The approach “1 Block” does not scale very well for more than two nodes and thus is not appropriate for this cluster. The approach “2 Blocks MPI” scales almost perfectly for up to 4 nodes but takes about as long on 8 nodes as on 4 nodes. This is due to the fact that for 8 nodes communication takes twice as much time as computation. The approach “2 Blocks IB” scales almost perfectly for up to 8 nodes. For 8 nodes, communication time is about two thirds of the computation time and thus can entirely be hidden. Doubling the number of nodes to 16 decreases communication time by a factor of about 0.7 (see Figure 4.4) while computation time is halved. Therefore, this approach may scale very well for up to 16 nodes or even further. However, a larger cluster with a high-speed InfiniBand network is required to proof this claim. The approach “Ind. Blocks” scales well for up to 8 nodes. It looses performance due to the reduction of the block size per node. In case the per-node block size is kept constant by increasing the total block size n, this approach scales perfectly as well even for larger clusters—since no communication is required during the computation. However, increasing the total block size also increases runtime and memory demand of BW2 as described earlier.

The approach “2 blocks IB” was used for the scalability tests that are described in the next paragraphs since it has a good performance for up to 8 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 fixed total block size n than the third approach with several independent column blocks.

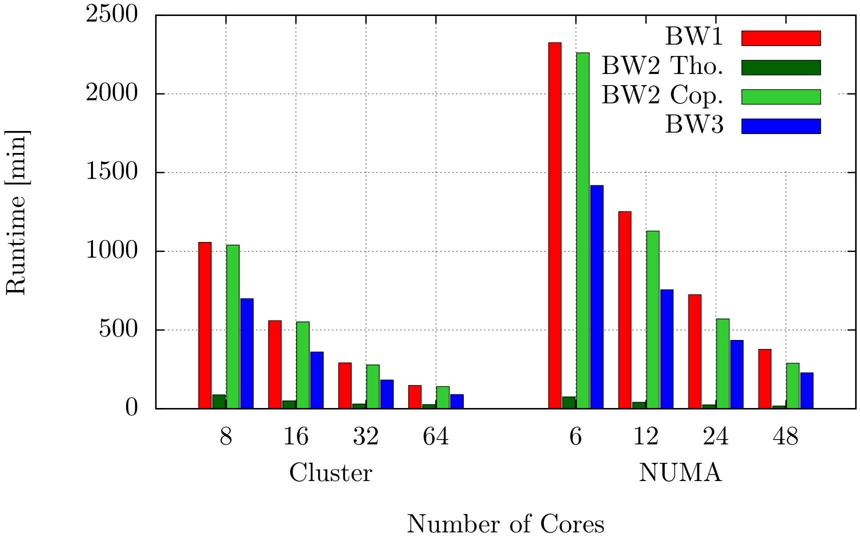

Figure 4.9: Runtime of XL 18-16 with Thomé's variant of BW2 on an 8 node cluster with 8 cores per node and on an 8 node NUMA systems with 6 cores per node.

4.6.3 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 $N = 5\,311\,735$ rows and columns; the row weight is $w_B = 153$.

For this experiment, the implementation of the block Wiedemann algorithm ran on 1, 2, 4, and 8 nodes of the cluster and on 1 to 8 CPUs of the NUMA system. Figure 4.9 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 node has more computing power than one NUMA node (see Table 4.2). At a first glance the implementation scales nicely: doubling of the core count roughly halves the runtime.

Figures 4.10 and 4.11 give a closer look on the parallel speedup and the parallel efficiency of BW1 and BW2; 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 an efficiency of at least 90% on 2, 4, and 8 cluster nodes. The efficiency of Thomé's BW2 is only around 75% on 4 nodes and drops to under 50% on 8 nodes. In particular the polynomial multiplications require a more efficient parallelization approach. However, Thomé's BW2 takes only a small part of the total runtime for this system size; for larger

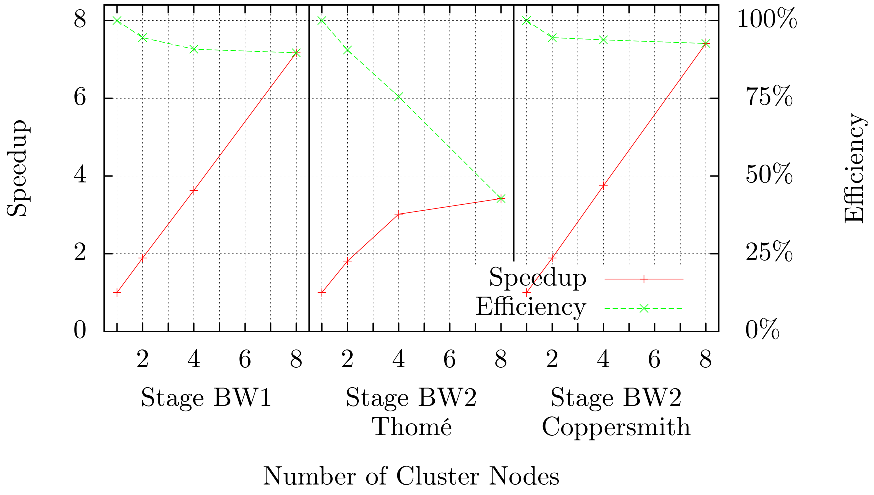

Figure 4.10: Speedup and efficiency of BW1 and BW2 on the cluster system.

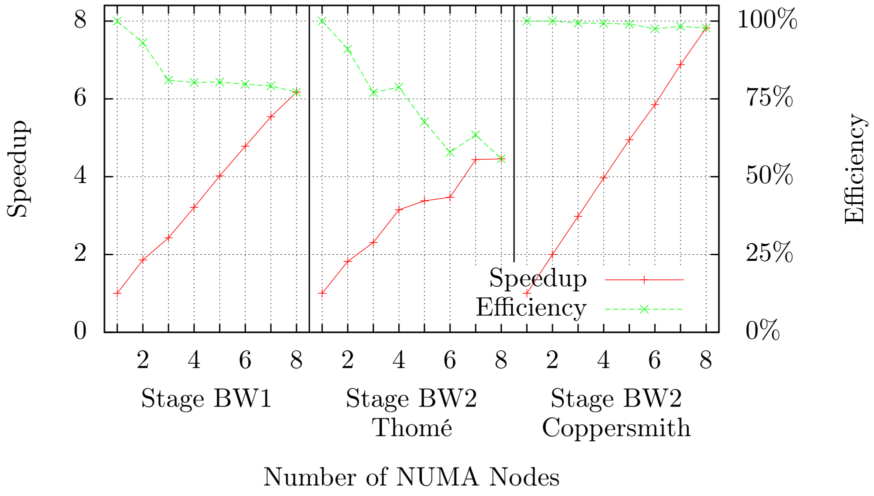

Figure 4.11: Speedup and efficiency of BW1 and BW2 on the NUMA system.

systems it is even smaller due its smaller asymptotic time complexity compared to steps BW1 and BW3. Thus, a lower scalability than BW1 and BW3 can be tolerated.

On the NUMA system, BW1 achieves an efficiency of over 75% 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 of over 90% 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 of around 55% for 8 NUMA nodes. 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.

5

Parallel implementation of Wagner's generalized birthday attack

Wagner's generalized birthday attack computes collisions for a cryptographic hash function. It is a generic attack that can be applied to a broad range of hash functions. Wagner's generalized birthday attack is relatively cheap in terms of computation but expensive in terms of memory demand. This puts a high pressure on capacity and bandwidth of mass storage.

This chapter describes a parallelized implementation of Wagner's generalized birthday attack that computes collisions for the compression function of the hash function FSB_{48}. This hash function is a toy version of the Fast Syndrome-Based hash function (FSB) proposed by Augot, Finiasz and Sendrier in [AFG$^+$08b] for the NIST SHA-3 competition. A straightforward implementation of Wagner's generalized birthday attack [Wag02a] would need 20 TB of storage. However, the attack was successfully performed on a small scale workstation-cluster of 8 nodes with a total memory of 64 GB RAM and 5.5 TB hard-disk storage. This chapter will give details about how to deal with this restricted storage capacity by applying and generalizing ideas described by Bernstein in [Ber07a].

This chapter is based on joint work with Bernstein, Lange, Peters, and Schwabe published in [BLN$^+$09] and as part of the PhD theses of Peters [Pet11a] and Schwabe [Sch11a]. The content of this chapter differs from [BLN$^+$09] only slightly in notation, structure and phrasing. The source code of the attack is publicly available and can be obtained from [NPS09].

Section 5.1 gives a short introduction to Wagner's generalized birthday attack and Bernstein's adaptation of this attack to storage-restricted environments. Section 5.2 describes the FSB hash function to the extent necessary to understand the attack methodology. Section 5.3 explains what strategy is used for the attack to make it fit on the restricted hard-disk space of the available computer cluster. Section 5.4 details the measures that have been applied to make the attack run as efficiently as possible. The overall cost of the attack is evaluated in Section 5.5 and cost estimates for a similar attack against full-size FSB are given in Section 5.6.

5.1 Wagner's generalized birthday attack

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

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

This section describes a solution to the generalized birthday problem due to Wagner [Wag02a]. Wagner also considered generalizations to operations other than xor, and to the case of k lists when k is not a power of 2.

5.1.1 Wagner's tree algorithm

Wagner's algorithm builds a binary tree of lists starting from the input lists. Let $L_{k,j}$ denote list j on level k of this tree. Therefore the input lists are $L_{0,0}, L_{0,1}, \dots, L_{0,2^{i-1}-1}$ (see Figure 5.1). The speed and success probability of the algorithm are analyzed under the assumption that each list contains $2^{B/i}$ elements chosen uniformly at random. The algorithm works as follows:

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

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

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

5.1.2 Wagner in storage-restricted environments

A 2007 paper [Ber07a] by Bernstein includes two techniques to mount Wagner's attack on computers which do not have enough memory to hold all list entries. Various special cases of the same techniques also appear in a 2005 paper [AFS05] by Augot, Finiasz, and Sendrier and in a 2009 paper [MS09] by Minder and Sinclair.

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

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

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

Note that clamping some bits through precomputation might be a good idea even if enough memory is available as the amount of data can be reduced in later steps. Therefore less data has to be moved and processed and thus it makes the computation more efficient.

After the precomputation step Wagner's tree algorithm is applied to lists containing bit strings of length B' where B' equals B minus the number of clamped bits. In the performance evaluation lists on level 0 will be considered only *after* clamping through precomputation. Therefore B instead of B' is used for the number of bits in these entries.

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

In the context of clamping through precomputation in each repetition the clamping values used during precomputation can be varied. If for some reason it is not possible to clamp any bits through precomputation the same idea of changing clamping values can be applied in an arbitrary merge step of the tree algorithm. Note that any solution to the generalized birthday problem can be found by some choice of clamping values.

Expected number of runs. Wagner's generalized birthday attack is a probabilistic algorithm. Without clamping through precomputation it produces an expected number of exactly one collision. However this does not mean that running the algorithm always gives a collision.

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

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

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

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

$$E(R) = \frac{1}{P_{\text{success}}}. \tag{5.1}$$

For larger values of $B - ib$ the expected number of runs is about 2^{B-ib}. The total runtime t_W of the attack can be modeled as being linear in the amount of data on level 0, i.e.,

$$t_W \in \Theta\left(2^{i-1} 2^{B-ib} 2^b\right). \tag{5.2}$$

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

Using Pollard iteration. If the number of uncontrolled bits is high because of memory restrictions, it may be more efficient to use a variant of Wagner's attack that uses Pollard iteration [Knu97, Chapter 3, exercises 6 and 7].

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

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

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

As Pollard iteration has square-root running time, the expected number of runs for this variant is $2^{B/2-(i-1)b/2}$, each taking time $2^{i-2}2^b$ (see (5.2)), so the expected running time is

$$t_{PW} \in \Theta\left(2^{i-2+B/2-(i-3)b/2}\right). \tag{5.3}$$

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

5.2 The FSB hash function

FSB was one of the 64 hash functions submitted to NIST's SHA-3 competition, and one of the 51 hash functions selected for the first round. However, FSB was significantly slower than most submissions, and was not selected for the second round.

This section briefly describes the construction of the FSB hash function. Details which are necessary for implementing the function but do not influence the attack are omitted. The second part of this section gives a rough description of how to apply Wagner's generalized birthday attack to find collisions of the compression function of FSB.

5.2.1 Details of the FSB hash function

The Fast Syndrome Based hash function (FSB) was introduced by Augot, Finiasz and Sendrier in 2003. See [AFS03], [AFS05], and [AFG⁺08b]. The security of FSB's compression function relies on the difficulty of the "Syndrome Decoding Problem" from coding theory.

The FSB hash function processes a message in three steps: First the message is converted by a so-called domain extender into suitable inputs for the compression function which processes the inputs in the second step. In the third and final step the Whirlpool hash function designed by Barreto and Rijmen [BR01] is applied to the output of the compression function in order to produce the desired length of output. Wagner's generalized birthday attack addresses the compression function of the second step. Therefore the domain extender, the conversion of the message to inputs for the compression function, and the last step involving Whirlpool will not be described.

The compression function. The main parameters of the compression function are called n, r and w: Consider n strings of length r which are chosen uniformly at random and can be written as an $r \times n$ binary matrix H. Note that the matrix H can be seen as the parity check matrix of a binary linear code. The FSB proposal [AFG⁺08b] specifies a particular structure of H for efficiency.

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

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

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

Finding preimages or collisions means solving two problems coming from coding theory: finding a preimage means solving the Regular Syndrome Decoding problem and finding collisions means solving the so-called 2-regular Null-Syndrome Decoding problem. Both problems were defined and proven to be NP-complete in [AFS05].

Parameters. Following the notation in [AFG$^+$08b] the term $\text{FSB}_{\texttt{length}}$ denotes the version of FSB which produces a hash value of length `length`. Note that the output of the compression function has r bits where r is considerably larger than `length`.

For the SHA-3 competition NIST demanded hash lengths of 160, 224, 256, 384, and 512 bits, respectively. Therefore the SHA-3 proposal contains five versions of FSB: FSB_{160}, FSB_{224}, FSB_{256}, FSB_{384}, and FSB_{512}. Table 5.1 gives the parameters for these versions.

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

A pseudo-random matrix. The attack against FSB_{48} uses a pseudo-random matrix H which is constructed as described in [AFG$^+$08b, Section 1.2.2]: H consists of 2048 submatrices, each of dimension 192×192. For the first submatrix consider a slightly larger matrix of dimension 197×192. Its first column consists of the first 197 digits of π where each digit is taken modulo 2. The remaining 191 columns of this submatrix are cyclic shifts of the first column. The matrix is then truncated to its first 192 rows which form the first submatrix of H. For the second submatrix consider digits 198 up to 394 of π. Again build a 197×192 bit matrix where the first column corresponds to the selected digits (each taken modulo 2) and the remaining columns are cyclic shifts of the first column. Truncating to the first 192 rows yields the second block matrix of H. The remaining submatrices are constructed in the same way.

This is one possible choice for the matrix H; the matrix may be defined differently. The attack described in this chapter does not make use of the structure of this particular matrix. This construction is used in the implementation since it is also contained in the FSB reference implementation submitted to NIST by the FSB designers. The reference implementation is available at [AFG+08a].

5.2.2 Attacking the compression function of FSB_{48}

Coron and Joux pointed out in [CJ04] that Wagner's generalized birthday attack can be used to find preimages and collisions in the compression function of FSB. The following paragraphs present a slightly streamlined version of the attack of [CJ04] in the case of FSB_{48}.

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

The attack uses 16 lists to solve this particular 48-sum problem. Each list entry will be the xor of three columns coming from one and a half blocks. This ensures that there are no overlaps, i.e., more than two columns coming from the same matrix block in the end. Applying Wagner's attack in a straightforward way means that each list has at least $2^{\lceil 192/5 \rceil}$ entries. By clamping away 39 bits in each step at least one collision is expected after one run of the tree algorithm.

Building lists. The 16 lists contain 192-bit strings each being the xor of three distinct columns of the matrix H. Each triple of three columns from one and a half blocks of H is selected in the following way:

List $L_{0,0}$ contains the sums of columns i_0, j_0, k_0, where columns i_0 and j_0 come from the first block of 2^{14} columns, and column k_0 is picked from the following block with the restriction that it is taken from the first half of it. Since there are no overlapping elements about 2^{27} sums of columns i_0 and j_0 are coming from the first block. These two columns are added to all possible columns k_0 coming from the first 2^{13} elements of the second block of the matrix H. In total there are about 2^{40} elements for $L_{0,0}$.

The second list $L_{0,1}$ contains sums of columns i_1, j_1, k_1, where column i_1 is picked from the second half of the second block of H and j_1 and k_1 come from the third block of 2^{14} columns. This again yields about 2^{40} elements. The lists $L_{0,2}, L_{0,3}, \ldots, L_{0,15}$ are constructed in the same way.

By splitting every second block in half several solutions of the 48-xor problem are neglected. For example, a solution involving two columns from the first half of the second block cannot be found by this algorithm. However, the alternative option of using fewer lists would require more storage and a longer precomputation phase to build the lists. This justifies the loss of possible solutions.

The proposed scheme generates more than twice the number of entries for each list than needed for a straightforward attack as explained in Section 5.1. About $2^{40}/4$ elements are likely to be zero on their least significant two bits. Clamping those two bits away should thus yield a list of 2^{38} bit strings. Note that since the two least significant bits of the list elements are known, they can be ignored and the list elements can be regarded as 190-bit strings. Therefore a straightforward application of Wagner's attack to 16 lists with about $2^{190/5}$ elements is expected to yield a collision after completing the tree algorithm.

5.3 Attack strategy

The computation platform for this particular implementation of Wagner's generalized birthday attack on FSB is an eight-node cluster of conventional desktop PCs. Each node has an Intel Core 2 Quad Q6600 CPU with a clock rate of 2.40 GHz and direct fully cached access to 8 GB of RAM. About 700 GB mass storage are provided by a Western Digital SATA hard disk with some space reserved for system and user data. This results in a total storage amount of less than $8 \cdot 700$ GB $=$ 5.5 TB for the full cluster. The nodes are connected via switched Gigabit Ethernet.

This section describes how to fit the attack on this cluster. Key issues are how to adapt the lists to the available storage and how to distribute data and computation over the nodes. Similar considerations can be drawn for a different target platform, in particular regarding number of computing nodes and available memory amount.

5.3.1 How large is a list entry?

The number of bytes required to store one list entry depends on how the entry is represented:

Value-only representation. The obvious way of representing a list entry is as a 192-bit string, the xor of columns of the matrix. Bits which are already known to be zero of course do not have to be stored, so on each level of the tree the number of bits per entry decreases by the number of bits clamped on the previous level. Ultimately the actual *value* of the entry is not of interest—for a successful attack it will be all-zero at the end—but the column positions in the matrix that lead to this all-zero value. However, Section 5.3.3 will show that computations only involving the *value* can be useful if the attack has to be run multiple times due to storage restrictions.

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

Value-and-clipped-positions representation. Instead of storing full positions alongside the value, storage can be saved by clipping the positions with loss of information. Positions can be clipped by, e.g., storing the position modulo 256. After the attack has successfully finished, the full position information can be computed by checking which of the possible positions lead to the respective intermediate results on each level.

Positions-only representation. If full positions are kept throughout the computation, the values do not need to be stored at all. Every time a value is required, it can be dynamically recomputed from the positions. In each level the size of a single entry doubles (because the number of positions doubles). The expected number of entries per list remains the same, but the number of lists halves. Therefore the total amount of data is the same on each level when using positions-only representation. As discussed in Section 5.2 there are 2^{40} possibilities to choose columns to produce entries of a list, so positions on level 0 can be encoded in 40 bits (5 bytes).

Observe that it is possible to switch between representations during computation if at some level another representation becomes more efficient: from value-and-positions representation to positions-only representation and back, from one of these two to value-and-clipped-positions representation, and from any other representation to value-only representation.

5.3.2 What list size can be handled?

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

As described in Section 5.2.2 the attack can start with 16 lists of size 2^{38}, each containing bit strings of length $r' = 190$. However, storing 16 lists with 2^{38} entries, each entry encoded in 5 bytes requires 20 TB of storage space which obviously does not fit the total amount of storage of 5.5 TB that is available on the cluster.

There are 16 lists on the first level, each list entry needs a minimum of 5 bytes for storage. Therefore at most $5.5 \cdot 2^{40}/2^4/5 = 1.1 \times 2^{36}$ entries per list fit into 5.5 TB storage. Some of the disk space is used for the operating system and user data. Therefore a straightforward implementation would use lists of size 2^{36}.

At most 2^{40} entries can be generated per list so following [Ber07a] by clamping 4 bits during list generation, each of the 16 lists has 2^{36} values. These values have a length of 188 bits represented through 5 bytes holding the positions from the matrix. Clamping 36 bits in each of the 3 steps leaves two lists of length 2^{36} with 80 non-zero bits. According to (5.1) the attack would be expected to run 256.5 times until a collision is found. The only way of increasing the list size to 2^{37} and thus reduce the number of runs is to use value-only representation on higher levels.

5.3.3 The strategy

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

Finding appropriate clamping constants. This task does not require storing the positions, since it only determines whether a certain set of clamping constants leads to a collision; it does not tell which matrix positions give this collision. Whenever storing the value needs less space than storing positions, the entries can be compressed by switching representation from positions to values. As a side effect this speeds up the computations because less data has to be loaded and stored.

Starting from lists $L_{0,0}, \ldots, L_{0,7}$, each containing 2^{37} entries first list $L_{3,0}$ (see Figure 5.1) is computed on 8 nodes. This list has entries with 78 remaining bits each. Section 5.4 will describe how these entries are presorted into 512 buckets according to 9 bits that therefore do not need to be stored. Another 3 bits are determined by the node holding the data (also see Section 5.4) so only 66 bits or 9 bytes of each entry have to be stored, yielding a total storage requirement of 1152 GB versus 5120 GB necessary for storing entries in positions-only representation.

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

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

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

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

With 37 bits clamped on every level and 3 clamped through precomputation there are only 4 uncontrolled bits left and therefore, according to (5.1), the algorithm is expected to yield a collision after 16.5 repetitions.

Computing the matrix positions of the collision. After the clamping constants which lead to a collision have been found, it is also known which value in the lists $L_{3,0}$ and $L_{3,1}$ yields a final collision. Now recompute lists $L_{3,0}$ and $L_{3,1}$ without compression to obtain the positions. For this task only positions are stored; values are obtained by dynamic recomputation. In total one half-tree computation requires 5120 GB of storage, hence, they can be performed one after the other on 8 nodes.

The (re-)computation of lists $L_{3,0}$ and $L_{3,2}$ is an additional time overhead over doing all computation on list positions in the first place. However, this cost is incurred only once, and is amply compensated for by the reduction of the number of repetitions compared to the straight forward attack.

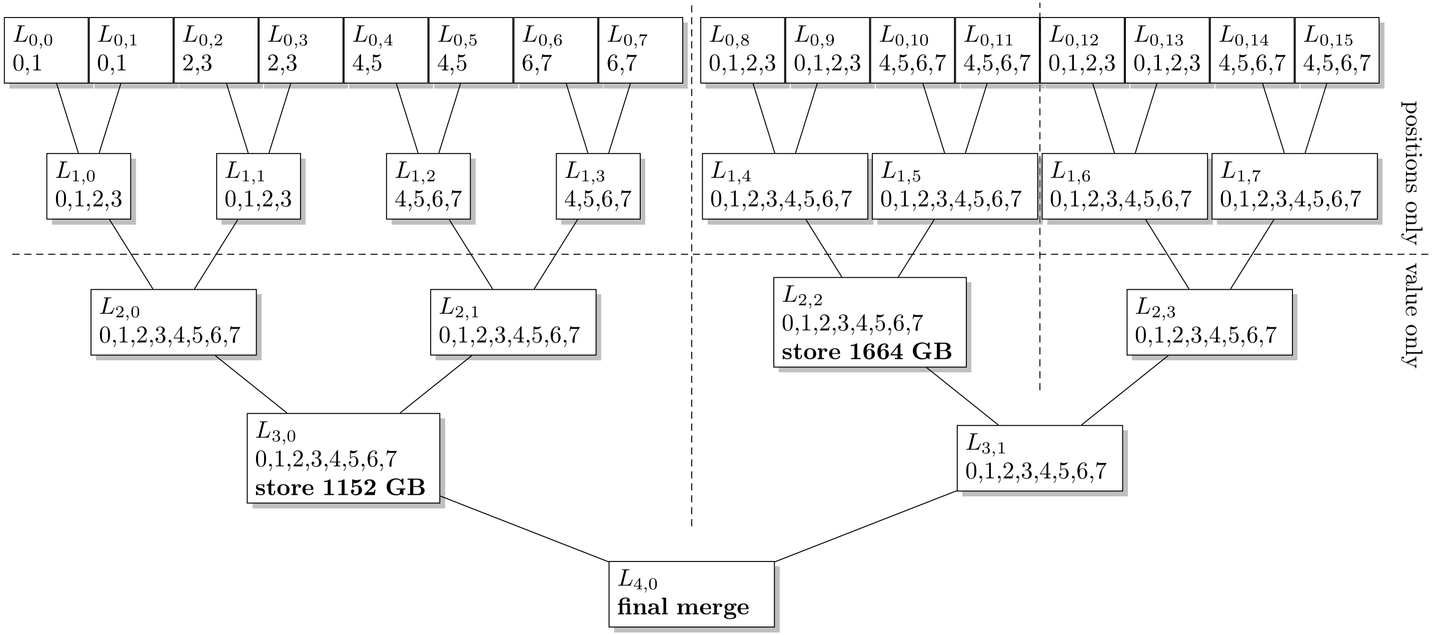

Figure 5.1: Structure of the attack: in each box the upper line denotes the list, the lower line gives the nodes holding fractions of this list

5.4 Implementing the attack

The communication between cluster nodes is accomplished by using MPICH2 [ANL] from the University of Chicago. MPICH2 is an implementation of the Message Passing Interface standard version 2.0. It provides an Ethernet-based back end for the communication with remote nodes and a fast shared-memory-based back end for local data exchange.

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

5.4.1 Parallelization

Most of the time in the attack is spent on determining the right clamping constants. As described in Section 5.3 this involves computations of several partial trees, e.g., the computation of $L_{3,0}$ from lists $L_{0,0}, \dots, L_{0,7}$ (half tree) or the computation of $L_{2,2}$ from lists $L_{0,8}, \dots, L_{0,11}$ (quarter tree). There are also computations which do not start with lists of level 0; the computation of list $L_{3,1}$ for example is computed from the (previously computed and stored) lists $L_{2,2}$ and $L_{2,3}$.

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

Distributing data over nodes. This algorithm is parallelized by distributing fractions of lists over the nodes in a way that each node can perform sort and merge locally on two lists. On each level of the computation, each node contains fractions of two lists. The lists on level j are split between n nodes according to $\lg(n)$ bits of each value. For example when computing the left half-tree, on level 0, node 0 contains all entries of lists $L_{0,0}$ and $L_{0,1}$ ending with a zero bit (in the bits not controlled by initial clamping), and node 1 contains all entries of lists $L_{0,0}$ and $L_{0,1}$ ending with a one bit.

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

The continuous dataflow of this implementation is depicted in Figure 5.2.

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

1. A whole fraction that exceeds the size of the memory can be sorted by sorting its presorted buckets independently.

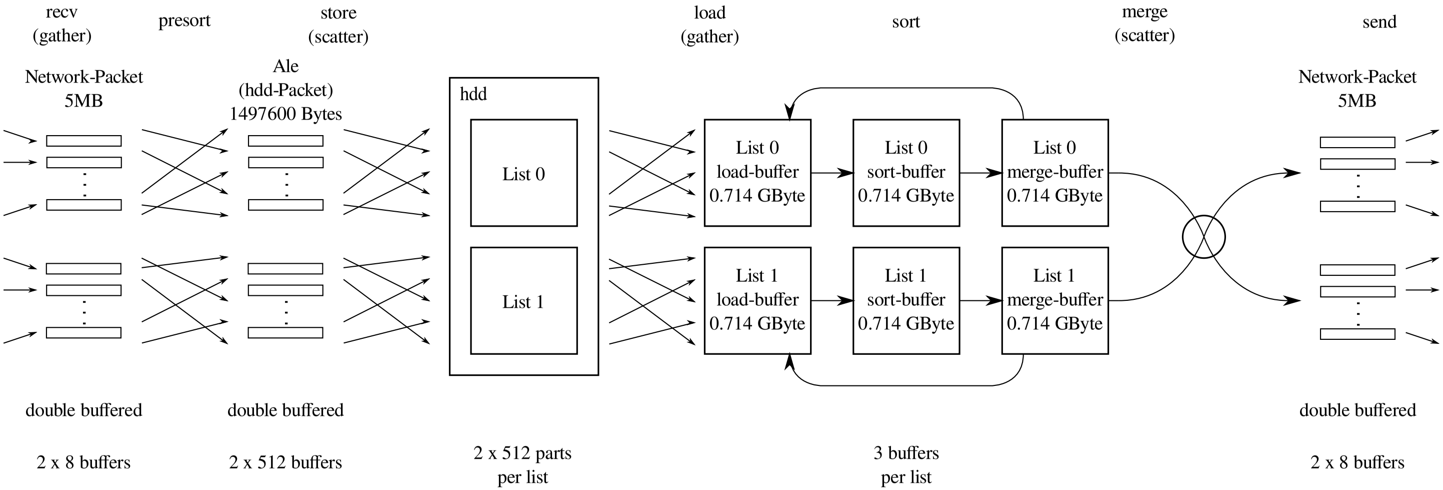

Figure 5.2: Data flow and buffer sizes during the computation.

2. Two adjacent buckets of the two lists on one node (with the same presort-bits) can be merged directly after they are sorted.
3. The 9 bits that determine the bucket for presorting do not need to be stored when entries are compressed to value-only representation.

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

5.4.2 Efficient implementation

Cluster computation imposes three main bottlenecks:

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

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

Finding bottlenecks. Hard-disk and network throughput of the cluster nodes are shown in Figure 5.3. Note that hard-disk throughput is measured directly on the device, circumventing the filesystem, to reach peak performance of the hard disk. Both sequential (seq) and randomized (rnd) access to the disk are shown.

The benchmarks reveal that, for sufficiently large packets, the performance of the system is mainly bottlenecked by hard-disk throughput. Since the throughput of MPI over Gigabit Ethernet is higher than the hard-disk throughput for packet sizes larger than 2^{16} bytes and since the same amount of data has to be sent that needs to be stored, no performance penalty is expected by the network for this size of packets.

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

Persistent data storage. To gain full hard-disk throughput a hand-written, throughput-optimized filesystem called *AleSystem* was implemented for the attack. The AleSystem provides fast and direct access to the hard disk and stores

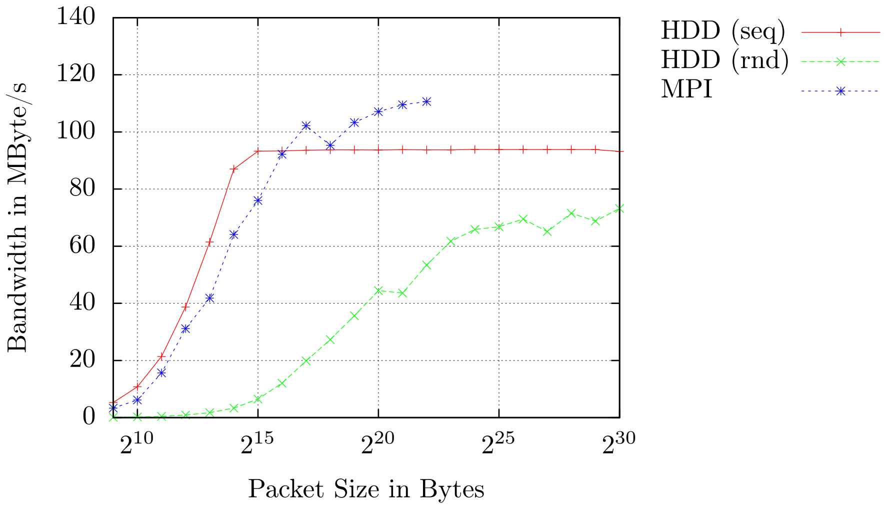

Figure 5.3: Benchmarks of hard-disk and network throughput. Hard-disk throughput is measured with sequential (seq) and random (rnd) access.

data in portions of *Ales*. Each cluster node has one large unformatted data partition, which is directly opened by the AleSystem using native Linux file I/O. After data has been written, it is not read for a long time and does not benefit from caching. Therefore caching is deactivated by using the open flag O_DIRECT. All administrative information is persistently stored as a file in the native Linux filesystem and mapped into the virtual address space of the process. On sequential access, the throughput of the AleSystem reaches about 90 MB/s which is roughly the maximum that the hard disk permits.

Tasks and threads. Since the cluster nodes are driven by quad-core CPUs, the speed of the computation is primarily based on multi-threaded parallelization. On the one side the tasks for receiving, presorting, and storing, on the other side the tasks for loading, sorting, merging, and sending are pipelined. Several threads are used for sending and receiving data and for running the AleSystem. The core of the implementation is given by five threads which process the main computation. There are two threads which have the task to presort incoming data (one thread for each list). Furthermore, sorting is parallelized with two threads (one thread for each list) and the merge task is assigned to another thread.

Memory layout. The benchmarks show that bigger buffers generally lead to higher throughput. However, the sum of all buffer sizes is limited by the size of the available RAM. Six buffers are needed for loading, sorting, and merging of two list buckets. Furthermore two times $2 \cdot 8$ network buffers are required for double-buffered send and receive, which results in 32 network buffers. Presorting entries of the two lists double-buffered into 512 buckets requires 2048 ales.

When a bucket is loaded from disk, its ales are treated as a continuous field of entries to avoid conditions and branches. Therefore, each ale must be completely filled with entries; no data padding at the end of each ale is allowed. Thus, the ales must have a size which allows them to be completely filled independent of the varying size of entries over the whole run of the program. Possible sizes of entries are 5, 10, 20, and 40 bytes when storing positions and 5, 10, 13, and 9 bytes when storing compressed entries. Furthermore, since the hard disk is accessed using DMA, the size of each ale must be a multiple of 512 bytes. Therefore the size of one ale must be a multiple of $5 \cdot 9 \cdot 13 \cdot 512$ bytes.

The size of network packets does not necessarily need to be a multiple of all possible entry sizes; if network packets happen not to be completely filled is is merely a small waste of network bandwidth.

In the worst case, on level 0 one list containing 2^{37} entries is distributed over 2 nodes and presorted into 512 buckets; thus the size of each bucket should be larger than $2^{37}/2/512 \cdot 5$ bytes = 640 MB. The actual size of each bucket depends on the size of the ales since it must be an integer multiple of the ale size.

Following these conditions the network packets have a size of $2^{20} \cdot 5$ bytes = 5 MB summing up to 160 MB for 32 buffers. The size of the ales is $5 \cdot 9 \cdot 13 \cdot 512 \cdot 5 = 1\,497\,600$ bytes (about 1.4 MB). Therefore 2.9 GB are necessary to store 2048 buffers for ales in memory. The buffers for the list buckets require $5 \cdot 9 \cdot 13 \cdot 512 \cdot 5 \cdot 512 = 766\,771\,200$ bytes (731.25 MB) each summing up to 4.3 GB for 6 buckets. Overall the implementation requires about 7.4 GB of RAM leaving enough space for the operating system and additional data, e.g., stack or data for the AleSystem.

Efficiency and further optimizations. The assignment of tasks to threads as described above results in an average CPU usage of about 60% and reaches a peak of up to 80%. The average hard-disk throughput is about 40 MB/s. The hard-disk benchmark (see Figure 5.3) shows that an average throughput between 45 MB/s and 50 MB/s should be feasible for packet sizes of 1.4 MB. Therefore further optimization of the sort task may make it possible to get closer to maximum hard-disk throughput.

5.5 Results

The implementation described in Sections 5.3 and 5.4 successfully computed a collision for the compression function of FSB_{48}. This section presents (1) the estimates, before starting the attack, of the amount of time that the attack would need; (2) measurements of the amount of time actually consumed by the attack; and (3) comments on how different amounts of storage would have changed the runtime of the attack.

5.5.1 Cost estimates

Finding appropriate clamping constants. As described before the first major step is to compute a set of clamping values which leads to a collision. In this first step entries are stored by positions on level 0 and 1 and from level 2 on list entries consist of values. Computation of list $L_{3,0}$ takes about 32 hours and list $L_{2,2}$ about 14 hours, summing up to 46 hours. These computations need to be done only once.

The time needed to compute list $L_{2,3}$ is about the same as for $L_{2,2}$ (14 hours), list $L_{3,1}$ takes about 4 hours and checking for a collision in lists $L_{3,0}$ and $L_{3,1}$ on level 4 about another 3.5 hours, summing up to about 21.5 hours. The expected number of repetitions of these steps is 16.5 and thus the expected runtime is about $16.5 \cdot 21.5 = 355$ hours.

Computing the matrix positions of the collision. Finally, computing the matrix positions after finding a collision requires recomputation with uncompressed lists. Entries of lists $L_{3,0}$ and $L_{3,1}$ need to be computed only until the entry is found that yields the collision. In the worst case this computation with uncompressed (positions-only) entries takes 33 hours for each half-tree, summing up to 66 hours.

Total expected runtime. Overall a collision for the FSB_{48} compression function is expected to be found in $46 + 355 + 66 = 467$ hours or about 19.5 days.

5.5.2 Cost measurements

Appropriate clamping constants were already found after only five iterations instead of the expected 16.5 iterations. In total the first phase of the attack took 5 days, 13 hours and 20 minutes.

Recomputation of the positions in $L_{3,0}$ took 1 day, 8 hours and 22 minutes and recomputation of the positions in $L_{3,1}$ took 1 day, 2 hours and 11 minutes. In total the attack took 7 days, 23 hours and 53 minutes.

Recall that the matrix used in the attack is the pseudo-random matrix defined in Section 5.2. The following column positions in this matrix have been found by the attack to yield a collision in the compression function: 734, 15006, 20748, 25431, 33115, 46670, 50235, 51099, 70220, 76606, 89523, 90851, 99649, 113400, 118568, 126202, 144768, 146047, 153819, 163606, 168187, 173996, 185420, 191473 198284, 207458, 214106, 223080, 241047, 245456, 247218, 261928, 264386, 273345, 285069, 294658, 304245, 305792, 318044, 327120, 331742, 342519, 344652, 356623, 364676, 368702, 376923, 390678.

5.5.3 Time-storage tradeoffs

As described in Section 5.3, the main restriction on the attack strategy was the total amount of background storage.

If 10496 GB of storage are available, lists of 2^{38} entries can be handled (again using the compression techniques described in Section 5.3). As described in Section 5.3 this would give exactly one expected collision in the last merge step and thus reduce the expected number of required runs to find the right clamping constants from 16.5 to 1.58. A total storage of 20 TB makes it possible to run a straightforward Wagner attack without compression which eliminates the need to recompute two half trees at the end.

Increasing the size of the background storage even further would eventually make it possible to store list entry values alongside the positions and thus eliminate the need for dynamic recomputation. However, the performance of the attack is bottlenecked by hard-disk throughput rather than CPU time so this measure is not likely to give any improvement.

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

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

5.6 Scalability analysis

The attack described in this chapter and the variants discussed in Section 5.5.3 are much more expensive in terms of time and especially memory than a brute-force attack against the 48-bit hash function FSB_{48}.

This section gives estimates of the power of Wagner's attack against the larger versions of FSB, demonstrating that the FSB design overestimated the power of the attack. Table 5.1 gives the parameters of all FSB hash functions.

A straightforward Wagner attack against FSB_{160} uses 16 lists of size 2^{127} containing elements with 632 bits. The entries of these lists are generated as xors of 10 columns from 5 blocks, yielding 2^{135} possibilities to generate the entries. Precomputation includes clamping of 8 bits. Each entry then requires 135 bits of storage so each list occupies more than 2^{131} bytes. For comparison, the largest available storage system in November 2011 offered 120 petabytes (less than 2^{57} bytes) of storage [Sim11].

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

Clamping 60 bits in each step leaves 273 bits uncontrolled so the Pollard variant of Wagner's algorithm (see Section 5.1.2) becomes more efficient than the plain attack. This attack generates 16 lists of size 2^{60}, containing entries which are the xor of 5 columns from 5 distinct blocks each. This gives the possibility to clamp 10 bits through precomputation, leaving $B = 630$ bits for each entry on level 0.

	n	w	r	lists	list size	bits / entry	total storage	time
FSB_{48}	3×2^{17}	24	192	16	2^{38}	190	$5 \cdot 2^{42}$	$5 \cdot 2^{42}$
FSB_{160}	7×2^{18}	112	640	16	2^{127}	632	$17 \cdot 2^{131}$	$17 \cdot 2^{131}$
				16*	2^{60}	630	$9 \cdot 2^{64}$	$9 \cdot 2^{224}$
FSB_{224}	2^{21}	128	896	16	2^{177}	884	$24 \cdot 2^{181}$	$24 \cdot 2^{181}$
				16*	2^{60}	858	$13 \cdot 2^{64}$	$13 \cdot 2^{343}$
FSB_{256}	23×2^{16}	184	1024	16	2^{202}	1010	$27 \cdot 2^{206}$	$27 \cdot 2^{206}$
				16*	2^{60}	972	$14 \cdot 2^{64}$	$14 \cdot 2^{386}$
				32*	2^{56}	1024	$18 \cdot 2^{60}$	$18 \cdot 2^{405}$
FSB_{384}	23×2^{16}	184	1472	16	2^{291}	1453	$39 \cdot 2^{295}$	$39 \cdot 2^{295}$
				32*	2^{60}	1467	$9 \cdot 2^{65}$	$18 \cdot 2^{618.5}$
FSB_{512}	31×2^{16}	248	1984	16	2^{393}	1962	$53 \cdot 2^{397}$	$53 \cdot 2^{397}$
				32*	2^{60}	1956	$12 \cdot 2^{65}$	$24 \cdot 2^{863}$

Table 5.1: Parameters of the FSB variants and estimates for the cost of generalized birthday attacks against the compression function. For Pollard's variant the number of lists is marked with a *. Storage is measured in bytes.

The time required by this attack is approximately 2^{224} (see (5.3)). This is substantially faster than a brute-force collision attack on the compression function, but is clearly much slower than a brute-force collision attack on the hash function, and even slower than a brute-force *preimage* attack on the hash function.

Similar statements hold for the other full-size versions of FSB. Table 5.1 gives rough estimates for the time complexity of Wagner's attack without storage restriction and with storage restricted to a few hundred exabytes (2^{60} entries per list). These estimates only consider the number and size of lists being a power of 2 and the number of bits clamped in each level being the same. The estimates ignore the time complexity of precomputation. Time is computed according to (5.2) and (5.3) with the size of level-0 entries (in bytes) as a constant factor.

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

Bibliography

[AFG$^+$08a] Daniel Augot, Matthieu Finiasz, Philippe Gaborit, Stéphane Manuel, and Nicolas Sendrier. "Fast Syndrome-Based hash function". 2008. URL: `http://www-rocq.inria.fr/secret/CBCrypto/index.php?pg=fsb` (cit. on p. 87).

[AFG$^+$08b] Daniel Augot, Matthieu Finiasz, Philippe Gaborit, Stéphane Manuel, and Nicolas Sendrier. "SHA-3 Proposal: FSB". Submission to NIST. 2008. URL: `http://www-rocq.inria.fr/secret/CBCrypto/fsbdoc.pdf` (cit. on pp. 81, 85, 86).

[AFI$^+$04] Gwénolé Ars, Jean-Charles Faugère, Hideki Imai, Mitsuru Kawazoe, and Makoto Sugita. "Comparison between XL and Gröbner basis algorithms". In: *Advances in Cryptology – ASIACRYPT 2004*. Ed. by Pil Lee. Vol. 3329. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2004, pp. 157–167. DOI: `10.1007/978-3-540-30539-2_24` (cit. on p. 47).

[AFS03] Daniel Augot, Matthieu Finiasz, and Nicolas Sendrier. "A fast provably secure cryptographic hash function". 2003. IACR Cryptology ePrint archive: 2003/230 (cit. on p. 85).

[AFS05] Daniel Augot, Matthieu Finiasz, and Nicolas Sendrier. "A family of fast syndrome based cryptographic hash functions". In: *Progress in Cryptology – Mycrypt 2005*. Ed. by Serge Vaudenay. Vol. 3715. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2005, pp. 64–83. DOI: `10.1007/11554868_6` (cit. on pp. 83, 85, 86).

[Amd67] Gene M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of AFIPS Spring Joint Computer Conference*. AFIPS 1967 (Spring). ACM, 1967, pp. 483–485. DOI: `10.1145/1465482.1465560` (cit. on p. 4).

[And04] David P. Anderson. "BOINC: a system for public-resource computing and storage". In: *5th IEEE/ACM International Workshop on Grid Computing*. IEEE Computer Society, 2004, pp. 4–10. DOI: `10.1109/GRID.2004.14` (cit. on p. 11).

[ANL] "MPICH2: High-performance and Widely Portable MPI". Argonne National Laboratory. URL: `http://www.mcs.anl.gov/research/projects/mpich2/` (cit. on pp. 14, 92).

[Ano] Anonymous. “Breaking ECC2K-130”. URL: `http://www.ecc-challenge.info` (cit. on p. 24).

[BBB$^+$09] 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 Gürkaynak, 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”. 2009. IACR Cryptology ePrint archive: 2009/541 (cit. on pp. 24, 25, 30, 34, 46).

[BBR10] Rajesh Bordawekar, Uday Bondhugula, and Ravi Rao. “Can CPUs Match GPUs on Performance with Productivity? Experiences with Optimizing a FLOP-intensive Application on CPUs and GPU”. Tech. rep. RC25033. IBM, 2010. URL: `http://domino.research.ibm.com/library/cyberdig.nsf/papers/EFE521AB23A0D28B85257784004DC9DD` (cit. on p. 16).

[BCC$^+$10] 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*. Ed. by Guang Gong and Kishan Chand Gupta. Vol. 6498. Lecture Notes in Computer Science. Document ID: `1957e89d79c5a898b6ef308dc10b0446`. Springer-Verlag Berlin Heidelberg, 2010, pp. 328–346. IACR Cryptology ePrint archive: 2012/002 (cit. on p. 24).

[BDM$^+$72] Wendell J. Bouknight, Stewart A. Denenberg, David E. McIntyre, J.M. Randall, Ahmed J. Sameh, and Daniel L. Slotnick. “The Illiac IV system”. In: *Proceedings of the IEEE* 60.4 (Apr. 1972), pp. 369–388. DOI: `10.1109/PROC.1972.8647` (cit. on p. 8).

[Ber07a] Daniel J. Bernstein. “Better price-performance ratios for generalized birthday attacks”. In: *Workshop Record of SHARCS 2007: Special-purpose Hardware for Attacking Cryptographic Systems*. Document ID: `7cf298bebf853705133a84bea84d4a07`. 2007. URL: `http://cr.yp.to/papers.html#genbday` (cit. on pp. 81, 83, 89).

[Ber07b] Daniel J. Bernstein. “qhasm: tools to help write high-speed software”. 2007. URL: `http://cr.yp.to/qhasm.html` (cit. on p. 20).

[Ber09a] Daniel J. Bernstein. “Batch binary Edwards”. In: *Advances in Cryptology – CRYPTO 2009*. Ed. by Shai Halevi. Vol. 5677. Lecture Notes in Computer Science. Document ID: `4d7766189e82c1381774dc840d05267b`. Springer-Verlag Berlin Heidelberg, 2009, pp. 317–336. DOI: `10.1007/978-3-642-03356-8_19` (cit. on p. 31).

[Ber09b] Daniel J. Bernstein. “Minimum number of bit operations for multiplication”. 2009. URL: `http://binary.cr.yp.to/m.html` (cit. on pp. 31, 39–41).

[Ber09c] Daniel J. Bernstein. “Optimizing linear maps modulo 2”. In: *Workshop Record of SPEED-CC: Software Performance Enhancement for Encryption and Decryption and Cryptographic Compilers.* Document ID: `e5c3095f5c423e2fe19fa072e23bd5d7`. 2009, pp. 3–18. URL: `http://binary.cr.yp.to/linearmod2-20091005.pdf` (cit. on pp. 29, 33).

[Ber66] Elwyn R. Berlekamp. “Nonbinary BCH decoding”. Institute of Statistics Mimeo Series No. 502. University of North Carolina, Dec. 1966. URL: `http://www.stat.ncsu.edu/information/library/mimeo.archive/ISMS_1966_502.pdf` (cit. on p. 50).

[BFAY+10] Rob H. Bisseling, Bas Fagginger Auer, Albert-Jan Yzelman, and Brendan Vastenhouw. “Mondriaan for sparse matrix partitioning”. Feb. 2010. URL: `http://www.staff.science.uu.nl/~bisse101/Mondriaan` (cit. on p. 72).

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Pat Hanrahan, Mike Houston, and Kayvon Fatahalian. “BrookGPU”. 2004. URL: `http://graphics.stanford.edu/projects/brookgpu/index.html` (cit. on p. 15).

[BGB10] Aydın Buluç, John R. Gilbert, and Ceren Budak. “Solving path problems on the GPU”. In: *Parallel Computing* 36.5–6 (June 2010), pp. 241–253. DOI: `10.1016/j.parco.2009.12.002` (cit. on p. 16).

[BKN+10] Joppe Bos, Thorsten Kleinjung, Ruben Niederhagen, and Peter Schwabe. “ECC2K-130 on Cell CPUs”. In: *Progress in Cryptology – AFRICACRYPT 2010.* Ed. by Daniel J. Bernstein and Tanja Lange. Vol. 6055. Lecture Notes in Computer Science. Document ID: `bad46a78a56fdc3a44fcf725175fd253`. Springer-Verlag Berlin Heidelberg, 2010, pp. 225–242. DOI: `10.1007/978-3-642-12678-9_14`. IACR Cryptology ePrint archive: 2010/077 (cit. on pp. 24, 29).

[BL07] Daniel J. Bernstein and Tanja Lange. “Explicit-Formulas Database”. 2007. URL: `http://www.hyperelliptic.org/EFD/` (cit. on p. 26).

[BL10] Daniel J. Bernstein and Tanja Lange. “Type-II optimal polynomial bases”. In: *Arithmetic of Finite Fields.* Ed. by M. Anwar Hasan and Tor Helleseth. Vol. 6087. Lecture Notes in Computer Science. Document ID: `90995f3542ee40458366015df5f2b9de`. Springer-Verlag Berlin Heidelberg, 2010, pp. 41–61. DOI: `10.1007/978-3-642-13797-6_4`. IACR Cryptology ePrint archive: 2010/069 (cit. on pp. 38, 39).

[BLN+09] Daniel J. Bernstein, Tanja Lange, Ruben Niederhagen, Christiane Peters, and Peter Schwabe. “FSBday: implementing Wagner’s generalized birthday attack against the SHA-3 round-1 candidate FSB”. In: *Progress in Cryptology – INDOCRYPT 2009.* Ed. by Bimal Roy and Nicolas Sendrier. Vol. 5922. Lecture Notes in Computer Science.

Document ID: `ded1984108ff55330edb8631e7bc410c`. Springer-Verlag Berlin Heidelberg, 2009, pp. 18–38. DOI: `10.1007/978-3-642-10628-6_2`. IACR Cryptology ePrint archive: 2009/292 (cit. on p. 81).

[BR01] Paulo S. L. M. Barreto and Vincent Rijmen. “The WHIRLPOOL Hash Function”. 2001. URL: `http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html` (cit. on p. 85).

[Cer97] Certicom. “Certicom ECC Challenge”. 1997. URL: `http://www.certicom.com/images/pdfs/cert_ecc_challenge.pdf` (cit. on pp. 23, 25).

[CJ04] Jean-Sébastien Coron and Antoine Joux. “Cryptanalysis of a provably secure cryptographic hash function”. 2004. IACR Cryptology ePrint archive: 2004/013 (cit. on p. 87).

[CKP$^+$00] Nicolas T. Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. “Efficient algorithms for solving overdefined systems of multivariate polynomial equations”. In: *Advances in Cryptology – EUROCRYPT 2000*. Ed. by Bart Preneel. Vol. 1807. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2000, pp. 392–407. DOI: `10.1007/3-540-45539-6_27` (cit. on pp. 47, 48).

[Cop94] Don Coppersmith. “Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm”. In: *Mathematics of Computation* 62.205 (1994), pp. 333–350. DOI: `10.1090/S0025-5718-1994-1192970-7` (cit. on pp. 49, 50, 55, 60).

[Cou03] Nicolas T. Courtois. “Higher order correlation attacks, XL algorithm and cryptanalysis of Toyocrypt”. In: *Information Security and Cryptology – ICISC 2002*. Ed. by Pil Lee and Chae Lim. Vol. 2587. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2003, pp. 182–199. DOI: `10.1007/3-540-36552-4_13` (cit. on p. 48).

[CP02] Nicolas T. Courtois and Josef Pieprzyk. “Cryptanalysis of block ciphers with overdefined systems of equations”. In: *Advances in Cryptology – ASIACRYPT 2002*. Ed. by Yuliang Zheng. Vol. 2501. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2002, pp. 267–287. DOI: `10.1007/3-540-36178-2_17`. IACR Cryptology ePrint archive: 2002/044 (cit. on p. 47).

[Die04] Claus Diem. “The XL-algorithm and a conjecture from commutative algebra”. In: *Advances in Cryptology – ASIACRYPT 2004*. Ed. by Pil Lee. Vol. 3329. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2004, pp. 146–159. DOI: `10.1007/978-3-540-30539-2_23` (cit. on p. 49).

[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.* ACM, 2002, pp. 75–83. DOI: `10.1145/780506.780516` (cit. on p. 47).

[Fau99] Jean-Charles Faugère. “A new efficient algorithm for computing Gröbner bases (F_4)”. In: *Journal of Pure and Applied Algebra* 139.1–3 (June 1999), pp. 61–88. DOI: `10.1016/S0022-4049(99)00005-5` (cit. on p. 47).

[Fis83] Joseph A. Fisher. “Very long instruction word architectures and the ELI-512”. In: *Proceedings of the 10th annual international symposium on Computer architecture — ISCA ’83.* Reprint at DOI:10.1109/MSSC.2009.932940. 1983, pp. 140–150 (cit. on p. 8).

[Fly66] Michael J. Flynn. “Very high-speed computing systems”. In: *Proceedings of the IEEE* 54.12 (Dec. 1966), pp. 1901–1909. DOI: `10.1109/PROC.1966.5273` (cit. on p. 3).

[Gir11] Damien Giry. “Keylength – Cryptographic Key Length Recommendation”. 2011. URL: `www.keylength.com` (cit. on p. 23).

[GSS07] Joachim von zur Gathen, Amin Shokrollahi, and Jamshid Shokrollahi. “Efficient multiplication using type 2 optimal normal bases”. In: *Arithmetic of Finite Fields.* Ed. by Claude Carlet and Berk Sunar. Vol. 4547. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2007, pp. 55–68. DOI: `10.1007/978-3-540-73074-3_6` (cit. on p. 30).

[Gue10] Shay Gueron. “Intel’s Advanced Encryption Standard (AES) Instructions Set”. White Paper. Version 3.0. Intel Mobility Group, Israel Development Center, Israel, Jan. 2010. URL: `http://software.intel.com/file/24917` (cit. on p. 19).

[Gus88] John L. Gustafson. “Reevaluating Amdahl’s law”. In: *Communications of the ACM* 31 (5 May 1988), pp. 532–533. DOI: `10.1145/42411.42415` (cit. on p. 5).

[Har03] Mark Jason Harris. “Real-Time Cloud Simulation and Rendering”. University of North Carolina Technical Report #TR03-040. PhD thesis. The University of North Carolina at Chapel Hill, 2003. URL: `ftp://ftp.cs.unc.edu/pub/publications/techreports/03-040.pdf` (cit. on p. 15).

[HMV04] Darrel R. Hankerson, Alfred J. Menezes, and Scott A. Vanstone. “Guide to Elliptic Curve Cryptography”. Springer-Verlag New York, 2004. DOI: `10.1007/b97644` (cit. on p. 26).

[Hof05] H. Peter Hofstee. "Power efficient processor architecture and the Cell processor". In: *11th International Symposium on High-Performance Computer Architecture — HPCA-11*. IEEE Computer Society, Feb. 2005, pp. 258–262. DOI: `10.1109/HPCA.2005.26` (cit. on p. 27).

[Hou11] Yunqing Hou. "asfermi: Assembler for the NVIDIA Fermi Instruction Set". 2011. URL: `http://code.google.com/p/asfermi/` (cit. on p. 16).

[HP07] John L. Hennessy and David A. Patterson. "Computer Architecture. A Quantitative Approach". 4th ed. Elsevier/Morgan Kaufmann Publishers, 2007 (cit. on pp. 6–8).

[IBM08] "Cell Broadband Engine Programming Handbook". Version 1.11. IBM DeveloperWorks, May 2008. URL: `https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1741C509C5F64B3300257460006FD68D` (cit. on pp. 27, 29).

[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". In: *Advances in Cryptology – CRYPTO 2010*. Ed. by Tal Rabin. Vol. 6223. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2010, pp. 333–350. DOI: `10.1007/978-3-642-14623-7_18`. IACR Cryptology ePrint archive: 2010/006 (cit. on p. 59).

[Kal95] Erich Kaltofen. "Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems". In: *Mathematics of Computation* 64.210 (1995), pp. 777–806. DOI: `10.1090/S0025-5718-1995-1270621-1` (cit. on pp. 53, 60).

[Knu97] Donald E. Knuth. "The Art of Computer Programming. Vol. 2, Seminumerical Algorithms". 3rd ed. Addison–Wesley, Nov. 1997 (cit. on p. 84).

[KO63] Anatolii Karatsuba and Yuri Ofman. "Multiplication of multidigit numbers on automata". In: *Soviet Physics Doklady* 7 (1963). Translated from Doklady Akademii Nauk SSSR, Vol. 145, No. 2, pp. 293–294, July 1962., pp. 595–596 (cit. on p. 31).

[Koc96] Paul C. Kocher. "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems". In: *Advances in Cryptology – CRYPTO '96*. Ed. by Neal Koblitz. Vol. 1109. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 1996, pp. 104–113. DOI: `10.1007/3-540-68697-5_9` (cit. on p. 19).

[KWA+01] Eric Korpela, Dan Werthimer, David Anderson, Jegg Cobb, and Matt Lebofsky. "SETI@home—massively distributed computing for SETI". In: *Computing in Science Engineering* 3.1 (Jan. 2001), pp. 78–83. DOI: `10.1109/5992.895191` (cit. on p. 11).

[Laa07] Wladimir J. van der Laan. “Cubin Utilities”. 2007. URL: `http://wiki.github.com/laanwj/decuda/` (cit. on pp. 16, 18, 44).

[Laz83] Daniel Lazard. “Gröbner-bases, Gaussian elimination and resolution of systems of algebraic equations”. In: *Proceedings of the European Computer Algebra Conference on Computer Algebra – EUROCAL ’83*. Ed. by J. A. van Hulzen. Vol. 162. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 1983, pp. 146–156. DOI: `10.1007/3-540-12868-9_99` (cit. on p. 47).

[LBL] “Berkeley UPC—Unified Parallel C”. Lawrence Berkeley National Laboratory and University of California, Berkeley. URL: `http://upc.lbl.gov/` (cit. on p. 14).

[LRD+90] Jed Lengyel, Mark Reichert, Bruce R. Donald, and Donald P. Greenberg. “Real-time robot motion planning using rasterizing computer graphics hardware”. In: *Proceedings of the 17th annual conference on Computer graphics and interactive techniques – SIGGRAPH ’90*. ACM, 1990, pp. 327–335. DOI: `10.1145/97879.97915` (cit. on p. 15).

[LS08] Calvin Lin and Larry Snyder. “Principles of Parallel Programming”. Addison-Wesley, Mar. 2008 (cit. on p. 3).

[Mas69] James L. Massey. “Shift-register synthesis and BCH decoding”. In: *IEEE Transactions on Information Theory* 15.1 (Jan. 1969), pp. 122–127. DOI: `10.1109/TIT.1969.1054260` (cit. on p. 50).

[MDS+11] Hans Meuer, Jack Dongarra, Erich Strohmaier, and Horst Simon, eds. “TOP500 Supercomputing Sites”. 2011. URL: `http://www.top500.org/` (cit. on p. 10).

[Moh01] Tzuong-Tsieng Moh. “On the method of XL and its inefficiency to TTM”. Jan. 2001. IACR Cryptology ePrint archive: 2001/047 (cit. on p. 49).

[Mon87] Peter L. Montgomery. “Speeding the Pollard and elliptic curve methods of factorization”. In: *Mathematics of Computation* 48.177 (1987), pp. 243–264. DOI: `10.1090/S0025-5718-1987-0866113-7` (cit. on p. 26).

[Mon95] Peter L. Montgomery. “A block Lanczos algorithm for finding dependencies over GF(2)”. In: *Advances in Cryptology — EUROCRYPT ’95*. Ed. by Louis Guillou and Jean-Jacques Quisquater. Vol. 921. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 1995, pp. 106–120. DOI: `10.1007/3-540-49264-X_9` (cit. on p. 49).

[MR02] Sean Murphy and Matthew J. B. Robshaw. “Essential algebraic structure within the AES”. In: *Advances in Cryptology – CRYPTO 2002*. Ed. by Moti Yung. Vol. 2442. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2002, pp. 1–16. DOI: `10.1007/3-540-45708-9_1` (cit. on p. 47).

[MR03] Sean Murphy and Matthew J. B. Robshaw. "Remarks on security of AES and XSL technique". In: *Electronics Letters* 39.1 (Jan. 2003), pp. 36–38. DOI: `10.1049/el:20030015` (cit. on p. 47).

[MS09] Lorenz Minder and Alistair Sinclair. "The extended k-tree algorithm". In: *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms – SODA 2009*. Ed. by Claire Mathieu. Society for Industrial and Applied Mathematics, 2009, pp. 586–595 (cit. on p. 83).

[Mun11] Aaftab Munshi, ed. "The OpenCL Specification". Version 1.2. Khronos OpenCL Working Group, Nov. 2011. URL: `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf` (cit. on p. 15).

[Nie11] Ruben Niederhagen. "calasm". 2011. URL: `http://polycephaly.org/projects/calasm/index.shtml` (cit. on p. 22).

[NPS09] Ruben Niederhagen, Christiane Peters, and Peter Schwabe. "FSBday, a parallel implementation of Wagner's generalized birthday attack". 2009. URL: `http://www.polycephaly.org/fsbday` (cit. on p. 81).

[NVI09a] "NVIDIA CUDA C Programming Best Practices Guide". NVIDIA. 2009. URL: `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf` (cit. on p. 17).

[NVI09b] "NVIDIA CUDA Programming Guide". Version 2.3. NVIDIA. 2009. URL: `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf` (cit. on p. 17).

[OMP] "The OpenMP API specification for parallel programming". OpenMP Architecture Review Board. URL: `http://openmp.org/` (cit. on p. 13).

[OMPI] "Open MPI: Open Source High Performance Computing". Indiana University and Indiana University Research and Technology Corporation. URL: `http://www.open-mpi.org/` (cit. on p. 14).

[OW99] Paul C. van Oorschot and Michael J. Wiener. "Parallel collision search with cryptanalytic applications". In: *Journal of Cryptology* 12.1 (Sept. 1999), pp. 1–28. DOI: `10.1007/PL00003816` (cit. on pp. 23, 24).

[Pet11a] Christiane Peters. "Curves, Codes, and Cryptography". PhD thesis. Eindhoven University of Technology, 2011. URL: `http://www2.mat.dtu.dk/people/C.Peters/thesis/20110510.diss.pdf` (cit. on p. 81).

[Pet11b] Christiane Peters. "Stellingen behorend bij het proefschrift *Curves, Codes, and Cryptography*". 2011. URL: `http://www2.mat.dtu.dk/people/C.Peters/thesis/stellingen.pdf` (cit. on p. 65).

[Pol78] John M. Pollard. "Monte Carlo methods for index computation (mod p)". In: *Mathematics of Computation* 32.143 (1978), pp. 918–924. DOI: `10.1090/S0025-5718-1978-0491431-9` (cit. on p. 24).

[Rák11] Ádám Rák. "AMD-GPU-Asm-Disasm". 2011. URL: `http://github.com/rakadam/AMD-GPU-Asm-Disasm` (cit. on p. 22).

[RRB$^+$08] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-Mei W. Hwu. "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA". In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming – PPoPP 2008*. ACM, 2008, pp. 73–82. DOI: `10.1145/1345206.1345220` (cit. on p. 16).

[Sch11a] Peter Schwabe. "High-Speed Cryptography and Cryptanalysis". PhD thesis. Eindhoven University of Technology, 2011. URL: `http://cryptojedi.org/users/peter/thesis/` (cit. on pp. 19, 24, 81).

[Sch11b] Peter Schwabe. "Theses accompanying the dissertation *High-Speed Cryptography and Cryptanalysis*". 2011. URL: `http://cryptojedi.org/users/peter/thesis/data/phdthesis-schwabe-statements.pdf` (cit. on p. 65).

[Sch77] Arnold Schönhage. "Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2". In: *Acta Informatica* 7.4 (1977), pp. 395–398. DOI: `10.1007/BF00289470` (cit. on p. 58).

[Sho07] Jamshid Shokrollahi. "Efficient implementation of elliptic curve cryptography on FPGAs". PhD thesis. Rheinische Friedrich-Wilhelms Universität Bonn, 2007. URL: `http://nbn-resolving.de/urn:nbn:de:hbz:5N-09601` (cit. on p. 30).

[Sim11] Tom Simonite. "IBM Builds Biggest Data Drive Ever". Technology Review. Aug. 25, 2011. URL: `http://www.technologyreview.com/computing/38440/` (cit. on p. 98).

[Tho02] Emmanuel Thomé. "Subquadratic computation of vector generating polynomials and improvement of the block Wiedemann algorithm". In: *Journal of Symbolic Computation* 33.5 (May 2002), pp. 757–775. DOI: `10.1006/jsco.2002.0533` (cit. on p. 58).

[Vil97a] Gilles Villard. "A study of Coppersmith's block Wiedemann algorithm using matrix polynomials". Rapport de Recherche 975 IM. Institut d'Informatique et de Mathématiques Appliquées de Grenoble, 1997 (cit. on pp. 53, 60).

[Vil97b] 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 ’97*. ACM. 1997, pp. 32–39. DOI: `10.1145/258726.258742` (cit. on pp. 53, 60).

[Wag02a] David Wagner. “A generalized birthday problem. Extended abstract”. In: *Advances in Cryptology – CRYPTO 2002*. Ed. by Moti Yung. Vol. 2442. Lecture Notes in Computer Science. See also newer version [Wag02b]. Springer-Verlag Berlin Heidelberg, 2002, pp. 288–304. DOI: `10.1007/3-540-45708-9_19` (cit. on pp. 81, 82, 110).

[Wag02b] David Wagner. “A generalized birthday problem. Full version”. See also older version [Wag02a]. 2002. URL: `http://www.cs.berkeley.edu/~daw/papers/genbday.html` (cit. on p. 110).

[War01] William A. Ward Jr. “The m5 macro processor”. 2001. URL: `http://www.factor3.de/research/m5/m5.html` (cit. on p. 21).

[Wie86] Douglas Wiedemann. “Solving sparse linear equations over finite fields”. In: *IEEE Transactions on Information Theory* 32.1 (Jan. 1986), pp. 54–62. DOI: `10.1109/TIT.1986.1057137` (cit. on p. 49).

[YC05] Bo-Yin Yang and Jiun-Ming Chen. “All in the XL family: theory and practice”. In: *Information Security and Cryptology – ICISC 2004*. Ed. by Choonsik Park and Seongtaek Chee. Vol. 3506. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2005, pp. 32–35. DOI: `10.1007/11496618_7` (cit. on p. 49).

[YCB$^+$07] Bo-Yin Yang, Owen Chia-Hsin Chen, Daniel J. Bernstein, and Jiun-Ming Chen. “Analysis of QUAD”. In: *Fast Software Encryption*. Ed. by Alex Biryukov. Vol. 4593. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2007, pp. 290–308. DOI: `10.1007/978-3-540-74619-5_19` (cit. on p. 47).

[YCC04] Bo-Yin Yang, Jiun-Ming Chen, and Nicolas Courtois. “On asymptotic security estimates in XL and Gröbner bases-related algebraic cryptanalysis”. In: *Information and Communications Security*. Ed. by Javier Lopez, Sihan Qing, and Eiji Okamoto. Vol. 3269. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2004, pp. 281–286. DOI: `10.1007/978-3-540-30191-2_31` (cit. on p. 48).

Summary

Parallel Cryptanalysis

Most of today's cryptographic primitives are based on computations that are hard to perform for a potential attacker but easy to perform for somebody who is in possession of some secret information, the key, that opens a back door in these hard computations and allows them to be solved in a small amount of time. To estimate the strength of a cryptographic primitive it is important to know how hard it is to perform the computation without knowledge of the secret back door and to get an understanding of how much money or time the attacker has to spend. Usually a cryptographic primitive allows the cryptographer to choose parameters that make an attack harder at the cost of making the computations using the secret key harder as well. Therefore designing a cryptographic primitive imposes the dilemma of choosing the parameters strong enough to resist an attack up to a certain cost while choosing them small enough to allow usage of the primitive in the real world, e.g. on small computing devices like smart phones.

This thesis investigates three different attacks on particular cryptographic systems: Wagner's generalized birthday attack is applied to the compression function of the hash function FSB. Pollard's rho algorithm is used for attacking Certicom's ECC Challenge ECC2K-130. The implementation of the XL algorithm has not been specialized for an attack on a specific cryptographic primitive but can be used for attacking some cryptographic primitives by solving multivariate quadratic systems. All three attacks are general attacks, i.e. they apply to various cryptographic systems; the implementations of Wagner's generalized birthday attack and Pollard's rho algorithm can be adapted for attacking other primitives than those given in this thesis.

The three attacks have been implemented on different parallel architectures. XL has been parallelized using the Block Wiedemann algorithm on a NUMA system using OpenMP and on an InfiniBand cluster using MPI. Wagner's attack was performed on a distributed system of 8 multi-core nodes connected by an Ethernet network. The work on Pollard's Rho algorithm is part of a large research collaboration with several research groups; the computations are embarrassingly parallel and are executed in a distributed fashion in several facilities with almost negligible communication cost. This dissertation presents implementations of the iteration function of Pollard's Rho algorithm on Graphics Processing Units and on the Cell Broadband Engine.

Curriculum Vitae

Ruben Niederhagen was born on August 10, 1980 in Aachen, Germany. After obtaining the German university-entrance qualification (Abitur) at the Goldberg Gymnasium Sindelfingen in 2000, he studied computer science at the RWTH Aachen University in Germany. In 2007 he graduated with the German degree "Diplom Informatiker" on the topic "Design and Implementation of a Secure Group Communication Layer for Peer-To-Peer Systems" and started his PhD studies at the Lehrstuhl für Betriebssysteme (research group for operating systems) of the Faculty of Electrical Engineering and Information Technology at the RWTH Aachen University. In 2009 he started a cooperation with the Coding and Cryptology group at Eindhoven University of Technology in the Netherlands and with the Fast Crypto Lab at the National Taiwan University in Taipei. He joined the Coding and Cryptology group in 2010 to continue his PhD studies under the supervision of Prof. Dr. Daniel J. Bernstein and Prof. Dr. Tanja Lange. During his PhD program, he commuted between the Netherlands and Taiwan on a regular basis to work as research assistant at the National Taiwan University and the Institute of Information Science at the Academia Sinica in Taipei under the supervision of Prof. Dr. Chen-Mou Cheng and Prof. Dr. Bo-Yin Yang. His dissertation contains the results of his work in the Netherlands and in Taiwan from 2009 to 2012.

His research interests are parallel computing and its application to cryptanalysis.