Accelerating Pre- and Post-Quantum Cryptography

Tung Chou

Accelerating Pre- and Post-Quantum Cryptography

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. F.T.P. Baaijens, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op maandag 27 juni 2016 om 16.00 uur

door

Tung Chou

geboren te Taipei, Taiwan

Contents

1 Introduction

Cryptography is the art of securing communication. The security comes with costs, however. There are computation costs for cryptographic operations such as encryption, decryption, signing, and verification. There are memory costs for storing the keys and carrying out cryptographic computations. There are also communication costs for parties involved in a protocol to exchange messages. In practice, these costs determine how widely a cryptographic scheme can be deployed. If the costs are too high, the users might not be able to afford it.

The works presented in this thesis focus on reducing computation costs of cryptographic software. There are many ways to accelerate cryptographic software. Similar to optimizing other types of software, there are different levels that one can work on. At a high level, one can perform platform-independent optimizations. This includes improving the existing algorithms or even designing new algorithms in order to reduce the number of operations (e.g., field multiplications) or even the complexity. At a low level, one can perform platform-dependent optimizations. This includes figuring out the best CPU instructions to use and instruction rescheduling for hiding instruction latencies.

Accelerating cryptographic software, however, is not the same as accelerating other types of software. The reason is that cryptographic implementations, in addition to carrying out the expected computation, need to be secure in a hostile environment. The adversary can often gain valuable information about the secret if the implementation itself leaks any information about it. Therefore, cryptographic computations should be leakage-free. Most of the implementations presented in this thesis aim to accelerate cryptographic software while being leakage-free.

In addition to accelerating cryptographic software, a small portion of this thesis is devoted to accelerating cryptanalytic software. Cryptanalytic software typically demands a large amount of computing resources, while cryptographic software is often required to run on very restricted platforms. Also, cryptanalytic software is not required to avoid leaking information about the computation since there is no secret. Optimizing cryptanalytic software is thus quite different from optimizing cryptographic software.

Post-quantum cryptography

In 1997, Shor showed in his seminal work [Sho97], that the most popular public-key cryptosystems nowadays, such as RSA and elliptic-curve-based systems, can be broken efficiently using large-scale quantum computers. Fortunately, there are a set of cryptosystems that withstand Shor's and all other known quantum algorithms, and the study of these cryptosystems is called *post-quantum cryptography.* The most studied post-quantum public-key systems are multivariate cryptosystems, code-based cryptosystems, lattice-based cryptosystems, and hash-based cryptosystems. Note that symmetric cryptosystems also belong to post-quantum cryptography.

Quantum computers are not an immediate threat, as it seems rather unlikely for large-scale quantum computers to be practical in 10 years. Unfortunately, post-quantum public-key cryptosystems are in general less usable than pre-quantum ones. For example, the first code-based encryption system, the original McEliece cryptosystem based on binary Goppa code [McE78], can be implemented efficiently and is quite confidence-inspiring, but it uses large keys of size up to a megabyte. The lattice-based encryption system NTRU [HPS98] is rather efficient in both key size and runtime, but it is not quite confidence-inspiring. It will take years for cryptographers to find satisfying post-quantum schemes, and it will take years to actually get these schemes deployed. If we want to be able to secure communication in the post-quantum world, it is essential to start now.

In February 2016, NIST announced their Call for Proposals for post-quantum cryptosystems at the Seventh International Conference on Post-Quantum Cryptography (PQCrypto). They announced that the deadline for submission will be in 2017, followed by an analysis phase of 3-5 years. This shows that post-quantum cryptography is no longer only a research area for the distant future; cryptography is actually moving towards the post-quantum world. In this thesis, I will present two works that are devoted to accelerating code-based encryption systems while being leakage-free.

Overview

The rest of this thesis is composed of four parts. The first part is the Preliminaries. Chapter 2 gives some background on implementation that is useful for the following chapters. Chapter 3 reviews the additive FFT proposed by Shuhong Gao and Todd Mateer in 2010. The works presented in Chapter 4 and 6 make use of this algorithm to speed up code-based and symmetric cryptosystems.

The second part covers binary-field cryptography. Chapter 4 presents a constant-time implementation for the McEliece cryptosystem using binary Goppa codes. This chapter is based on the CHES 2013 paper "McBits: fast constant-time code-based cryptography" [BCS13] with Daniel J. Bernstein and Peter Schwabe. The main difference between this chapter and the paper is that the paper contains a description for the Gao–Mateer additive FFT, which is now covered by Chapter 3.

Chapter 5 presents a constant-time implementation for the McEliece cryptosystem using QC-MDPC codes. This chapter is based on my preprint "QcBits: constant-time small-key code-based cryptography" [Cho16]. At this moment the preprint has been submitted to a conference, but it is not published yet.

Chapter 6 presents a high-security message authentication code which is designed for minimizing bit operations. This chapter is based on the SAC 2014 paper "Faster binary-field multiplication and faster binary-field MACs" [BC14] with Daniel J. Bernstein. Except for minor changes for formatting, there is essentially no difference between the chapter and the paper.

The third part covers elliptic-curve cryptography. Chapter 7 presents an optimized protocol for oblivious transfers, along with a constant-time implementation. This chapter is based on the Latincrypt 2015 paper "The simplest protocol for oblivious transfer" [CO15] with Claudio Orlandi. The main difference between the chapter and the paper is that the paper includes proofs for theorems and lemmas.

Chapter 8 presents a constant-time implementation for Curve25519. This chapter is based on my paper "Sandy2x: new Curve25519 speed records" [Cho15] at SAC 2015. Except for minor changes for formatting, there is essentially no difference between the chapter and the paper.

Chapter 9 presents an approach to manipulate curve standards. This chapter is based on the full version of the SSR 2015 paper "How to manipulate curve standards: a white paper for the black hat" [BCC+15] with Daniel J. Bernstein, Chitchanok Chuengsatiansup, Andreas Hülsing, Eran Lambooij, Tanja Lange, Ruben Niederhagen, and Christine van Vredendaal. Except for minor changes for formatting, there is essentially no difference between this chapter and the paper.

The fourth part covers polynomial-system solving. Chapter 10 presents an implementation for the extended linearization (XL) algorithm. This chapter is based on an extended version of the CHES 2012 paper "Solving quadratic equations with XL on parallel architectures" [CCN+12] with Chen-Mou Cheng, Ruben Niederhagen, and Bo-Yin Yang. In particular, Sections 10.2 and 10.3 go beyond the conference version.

Part I

Preliminaries

2 Cryptographic implementations

This chapter gives some background on implementation that is useful for the following chapters. Section 2.1 reviews vectorization, i.e., the usage of vector units. Section 2.2 reviews the concept of timing attacks and what programmers can do to avoid them. Section 2.3 reviews bitslicing, a technique of manipulating bit-transposed data. Section 2.4 reviews the development tool "qhasm" for assembly programming.

2.1 Vectorization

Most CPUs nowadays contain a set of general-purpose registers of width 32 or 64 bits. A large set of instructions are available for manipulating these registers, including integer arithmetic instructions and bitwise logical instructions. With the instructions for manipulating general-purpose registers, CPUs are able to carry out a wide variety of tasks efficiently. However, *vector units* made CPUs even more computationally powerful.

In 1999, Intel introduced the Streaming SIMD Extensions (SSE) to the x86 architecture. In particular, a set of 128-bit registers, called *XMM registers*, are introduced for the extension. SSE and its successors (SSE2, SSE3, etc.) provide instructions for manipulating the XMM registers. Typical instructions include entry-wise arithmetic operations (additions, subtractions, multiplications, etc.) by viewing each operand as a vector of 4 32-bit or 2 64-bit integers/floating-point numbers, and bitwise logical instructions. In 2008, Intel introduced the Advanced Vector Extensions (AVX). AVX (and its successor AVX2) instructions provide a similar functionality as SSE, but they operate on longer vectors.

With SSE/AVX instructions, several 32/64-bit operations can be carried out in one instruction. However, since these operations are carried out in parallel, the programmer needs to find independent 32/64-bit operations in the algorithm in order to fully exploit the power of vector instructions. Depending on the algorithm, this is not always an easy task (and sometimes this is not even possible). Furthermore, the values that need to be processed in parallel may not be in the required locations in the memory, or they may be in different registers. In these cases, we may need instructions such as “shuffling” or “unpacking” to move the values around, which adds overhead to the computation.

2.2 Timing attacks and constant-time implementations

Cryptographic schemes are often modeled as a set of algorithms. For example, a public-key encryption scheme is often modeled as a collection of three algorithms: key-pair generation, encryption, and decryption. Ideally, the parties that use a cryptographic protocol communicate with each other by exchanging the outputs of the algorithms. Of course, there can be malicious parties or an external entity interacting with honest parties, and the scheme is supposed to be secure even if this happens. In theoretical security proofs the information that an adversary obtains is often modeled as the data sent by the parties. However, in practice the adversaries are often able to exploit extra information leaked by the physical implementation of the algorithms. Attacks based on the knowledge of such information are called *side-channel attacks*.

In particular, *timing* is a common type of side-channel information when it depends on secret information. Attacks based on secret-dependent timings are called *timing attacks*. As a simple example, the double-and-add algorithm is vulnerable to timing attacks since the timing depends on the bits of the scalar which is usually a secret. Unfortunately, timing attacks are not just text-book material; they are actually used to break practical systems. For example, the well-known Lucky Thirteen attack [AP13] on TLS is a timing attack.

The obvious way to avoid timing attacks completely is to make sure the runtime is independent of the secret information. Implementations achieving this are called constant-time implementations. For cryptographic software, being constant-time means that secret values should not be used as operands of non-constant-time instructions. Note that hyperthreading allows adversarial software to measure the timing for several instructions. Being constant-time also means that there must not be any secret conditions and secret memory indices. A secret condition can lead to runtime differences when two branches have different amounts of computation. What makes the situation worse is the branch prediction mechanism implemented in essentially all CPUs nowadays. Secret memory indices can cause differences in runtime because memory accesses are slower in case of cache misses. Moreover, there are also timing variations inside cache; see [YGH16].

Consider the following C code that uses a secret condition:

```
if (cond) value = a;
else      value = b;
```

The code has at least two problems: the C-compiler might use branches to implement the if-statement, causing timing issues due to branch prediction. Furthemore, the memory access to `a` and `b` depends on the secret condition. The compiler might compile the code into conditional moves, which is then constant-time. However, this is not something we can always expect. In order to make the code constant-time, the trick is to convert the branch into arithmetic:

```
value = (a & mask) | (b & ~mask);
```

Here `mask` is $11...1_2$ if `cond` is True, and `mask` is 0 otherwise. How `mask` should be computed from `cond` in constant time depends on the data types, but this is quite easy in any case. Of course, in practice the situation can be much more complicated than this simple example, and making a program both constant-time and efficient can be a challenging task.

2.3 Bitslicing

The idea of bitslicing is to use bitwise logical operations to simulate n independent copies of a combinational circuit, where n is the width of the registers. Consider a simple 2-gate circuit

$$(a \oplus b) \odot c,$$

where $\oplus$ denotes the XOR gate and $\odot$ denotes the AND gate. The corresponding bit-sliced C code is simply

```
value = (A & B) | C;
```

Suppose `A`, `B`, and `C` are n-bit words, each bit position then simulates the simple circuit.

In some situations, bitslicing can be useful when we need to carry out some computations that are not well-supported by the CPU. For example, on many CPUs there are no proper instructions for binary-field arithmetic, and one can simply use bitslicing to simulate the arithmetic circuits. However, bitslicing also requires a lot of parallelism as n copies are run in parallel, so it does not always help to accelerate a program.

Since the number of inputs that are processed at the same time depends on the register width, it is often preferable to use wide registers for bitslicing. This is why bitslicing works well with vectorization.

2.4 qhasm

qhasm [Ber07b] is a development tool created by Daniel J. Bernstein to simplify assembly programming. It provides the following two very useful features for programmers.

The first feature is a customized instruction syntax, meaning that the user is able to define the instruction syntax for instructions in the assembly program. This works by using a so-called *machine description file* for the target platform. Each line of a description file defines the syntax for an instruction on the platform. The programmer, when writing qhasm code (in `.q` files), uses the syntax defined in the description file instead of the assembly syntax that is required by the platform specific assembler (in `.s` files). The qhasm code can then be compiled into `.s` files using qhasm such that each line of the qhasm code gets translated into the conventional assembly syntax. This feature is convenient, as the programmer does not have to memorize the conventional assembly syntax. The programmer can use a more friendly syntax as in high-level languages. Note that this also improves portability: If two platforms provide very similar instructions (e.g., unsigned 64-bit integer addition), by defining the instructions to have the same syntax we can make the qhasm code portable.

The second feature is automatic register allocation. This means that, when writing qhasm code, the programmer does not have to specify which registers are used as operands of the instructions. Instead, the programmer uses *register variables* which can be named meaningfully. These variables are pretty much like variables in high-level languages. The programmer declares which type of the registers should be used for each register variable at the beginning of the qhasm code. The programmer can then use the variables as operands for compatible instructions, using the self-defined syntax. When the qhasm code gets compiled, qhasm figures out the lifetime of register variables and assigns each register variable to a valid register for each instruction if possible. Note that qhasm does not perform register spilling, so the programmer still has to limit the working set.

Here is a line in a machine description file for architectures with YMM registers:

```
r = s ^ t:>r=reg256:<s=reg256:<t=reg256:asm/vxorpd <s,<t,>r:
```

The line can be viewed as several parts separated by colons. The last part

```
asm/vxorpd <s,<t,>r
```

indicates that the actual instruction being used is `vxorpd`, the 3-operand bitwise-XOR instruction for YMM registers. The first part

```
r = s ^ t
```

indicates the syntax that the programmer uses in qhasm code. The middle parts

```
>r=reg256:<s=reg256:<t=reg256
```

indicates that the operands are of type `reg256`, which means YMM registers, with `s` and `t` as inputs and `r` as output. To compute the XOR of two YMM registers y and z and store the results in another YMM register x, the programmer declares register variables `x,y,z` as

```
reg256 x
reg256 y
reg256 z
```

at the beginning of the qhasm code and then simply writes

```
x = y ^ z
```

in the `.q` file. The qhasm compiler then compiles this code into something like

```
vxorpd %ymm1, %ymm2, %ymm3
```

3
The Gao–Mateer additive FFT

Fast Fourier transforms (FFTs), in a broad sense, are special multi-point evaluation algorithms that take an essentially linear number of operations in the underlying ring. In the case of multiplicative FFTs, which are the most common type of FFTs, the evaluation points are the powers of a primitive root of unity. Multiplicative FFTs rely on the multiplicative structure of the evaluation points to perform efficient multi-point evaluations.

This chapter presents an "additive FFT" algorithm introduced in 2010 [GM10] by Gao and Mateer (improving upon previous algorithms by Wang and Zhu in [WZ88], Cantor in [Can89], and von zur Gathen and Gerhard in [GG96]). This algorithm evaluates a polynomial at every element of a characteristic-2 field $\mathbb{F}_q$, or more generally every element of an $\mathbb{F}_2$-linear subspace of $\mathbb{F}_q$.

Additive FFTs, instead of exploiting the multiplicative structure of the set of the evaluation points, make use of the additive structure. This nature makes additive FFTs more natural than multiplicative FFTs for characteristic-2 fields. The works presented in Chapter 4 and 6 rely heavily on the Gao–Mateer additive FFT.

Gao and Mateer's paper has three parts. The first part describes the "Taylor expansion", which is essentially a conversion of radix from x to $x^t + x$. The algorithm is reviewed in Section 3.3 and 3.4. Note that compared to the description in the Gao–Mateer paper, the description in the sections focuses more on how the algorithm can be implemented. The second part of the paper describes an additive FFT for arbitrary $\mathbb{F}_{2^m}$, which uses the radix conversion as a subroutine. The algorithm is reviewed in Section 3.1 and 3.2. The third part describes an additive FFT for $\mathbb{F}_{2^m}$, where m is a power of 2. Since the algorithm is not used in the works presented in this thesis, the reader may refer to the original paper if interested.

3.1 Additive FFT: overview

Let $f \in \mathbb{F}_q[x]$ where $q = 2^m$. The basic idea of the algorithm is to write $f \in \mathbb{F}_q[x]$ in the form $f^{(0)}(x^2+x)+xf^{(1)}(x^2+x)$ for two half-degree polynomials $f^{(0)}, f^{(1)} \in \mathbb{F}_q[x]$; this is handled efficiently by the "radix conversion" described below. Let $\alpha \in \mathbb{F}_q$. This form of f shows a large overlap between evaluating $f(\alpha)$ and evaluating $f(\alpha+1)$. Specifically, $(\alpha+1)^2 + (\alpha+1) = \alpha^2+\alpha$, so

$$\begin{aligned} f(\alpha) &= f^{(0)}(\alpha^2+\alpha) + \alpha f^{(1)}(\alpha^2+\alpha), \\ f(\alpha+1) &= f^{(0)}(\alpha^2+\alpha) + (\alpha+1) f^{(1)}(\alpha^2+\alpha). \end{aligned}$$

Evaluating both $f^{(0)}$ and $f^{(1)}$ at $\alpha^2+\alpha$ produces both $f(\alpha)$ and $f(\alpha+1)$ with just a few more field operations: multiply the $f^{(1)}$ value by α, add the $f^{(0)}$ value to obtain $f(\alpha)$, and add the $f^{(1)}$ value to obtain $f(\alpha+1)$.

The additive FFT applies this idea recursively. For example, if $\beta^2+\beta = 1$ then evaluating f at $\alpha, \alpha+1, \alpha+\beta, \alpha+\beta+1$ reduces to evaluating $f^{(0)}$ and $f^{(1)}$ at $\alpha^2+\alpha$ and $\alpha^2+\alpha+1$, which in turn reduces to evaluating four polynomials at $\alpha^4+\alpha$. One can handle any subspace by "twisting", as discussed below.

For comparison, a standard multiplicative FFT writes f in the form $f^{(0)}(x^2) + xf^{(1)}(x^2)$ (a simple matter of copying alternate coefficients of f), reducing the computation of both $f(\alpha)$ and $f(-\alpha)$ to the computation of $f^{(0)}(\alpha^2)$ and $f^{(1)}(\alpha^2)$. The problem in characteristic 2 is that α and $-\alpha$ are the same. The standard workaround is a radix-3 FFT, writing f in the form $f^{(0)}(x^3) + xf^{(1)}(x^3) + x^2 f^{(2)}(x^3)$, but this is considerably less efficient.

Note that the additive FFT, like the multiplicative FFT, is suitable for small hardware: it can easily be written as a highly structured iterative algorithm rather than a recursive algorithm, and at a small cost in arithmetic it can be written to use very few constants.

3.2 Additive FFT: detail

Consider the problem of evaluating a 2^m-coefficient polynomial f at all subset sums ($\mathbb{F}_2$-linear combinations) of $\beta_1, \ldots, \beta_m \in \mathbb{F}_q$: i.e., computing $f(0)$, $f(\beta_1)$, $f(\beta_2)$, $f(\beta_1 + \beta_2)$, etc. Gao and Mateer handle this problem as follows.

If $m = 0$ then the output is simply $f(0)$. Assume from now on that $m \geq 1$.

If $\beta_m = 0$ then the output is simply two copies of the output for $\beta_1, \ldots, \beta_{m-1}$. (The algorithm stated in [GM10] is slightly less general: it assumes that $\beta_1, \ldots, \beta_m$ are linearly independent, excluding this case.) Assume from now on that $\beta_m \neq 0$.

Assume without loss of generality that $\beta_m = 1$. To handle the general case, compute $g(x) = f(\beta_m x)$, and observe that the output for $f, \beta_1, \beta_2, \ldots, \beta_m$ is the same as the output for $g, \beta_1/\beta_m, \beta_2/\beta_m, \ldots, 1$. (This is the "twisting" mentioned above. Obviously the case $\beta_m = 1$ is most efficient; the extent to which this case can be achieved depends on how many powers of 2 divide $\lg q$ (lg denotes logarithm with base 2).)

Apply the radix conversion described below to find two 2^{m-1}-coefficient polynomials $f^{(0)}, f^{(1)} \in \mathbb{F}_q[x]$ such that $f = f^{(0)}(x^2+x) + xf^{(1)}(x^2+x)$. Recursively evaluate

$f^{(0)}$ at all subset sums of $\delta_1, \dots, \delta_{m-1}$, where $\delta_i = \beta_i^2 + \beta_i$. Also recursively evaluate $f^{(1)}$ at all subset sums of $\delta_1, \dots, \delta_{m-1}$.

Observe that each subset sum $\alpha = \sum_{i \in S} \beta_i$ with $S \subseteq \{1, 2, \dots, m-1\}$ has $\alpha^2 + \alpha = \gamma$ where $\gamma = \sum_{i \in S} \delta_i$. Compute $f(\alpha)$ as $f^{(0)}(\gamma) + \alpha f^{(1)}(\gamma)$, and compute $f(\alpha + 1)$ as $f^{(0)}(\gamma) + \alpha f^{(1)}(\gamma) + f^{(1)}(\gamma) = f(\alpha) + f^{(1)}(\gamma)$. Note that these evaluation points α and $\alpha + 1$ cover all subset sums of $\beta_1, \beta_2, \dots, \beta_m$, since $\beta_m = 1$.

3.3 Radix conversion: an example

The radix conversion subroutine converts a polynomial $f \in \mathbb{F}_q[x]$ from its radix-x representation

$$f_0 + f_1 x + f_2 x^2 + \cdots$$

into the radix-$(x^t + x)$ representation. Although the algorithm can work for any q, in this thesis it is assumed $q = 2^m$. Consider a polynomial f of 8 coefficients:

$$f(x) = f_0 + f_1 x + f_2 x^2 + f_3 x^3 + f_4 x^4 + f_5 x^5 + f_6 x^6 + f_7 x^7.$$

In the case of $t = 2$, the radix conversion aims to find $f'_0, f'_1, \cdots, f'_7$, such that

$$f(x) = (f'_0 + f'_1 x) + (f'_2 + f'_3 x)(x^2 + x) + (f'_4 + f'_5 x)(x^2 + x)^2 + (f'_6 + f'_7 x)(x^2 + x)^3.$$

In other words, f is represented as the coordinates $(f'_0, f'_1, \cdots, f'_7)$ with respect to the basis elements

$$1, x, x^2 + x, x(x^2 + x), (x^2 + x)^2, x(x^2 + x)^2, (x^2 + x)^3, x(x^2 + x)^3.$$

For the discussion below, it is convenient to view the basis elements as "subset products" of $\{x, x^2 + x, (x^2 + x)^2\}$. The radix conversion can then be viewed as a change of basis from $1, x, x^2, \dots, x^7$ to $1, x, x^2 + x, \dots, x(x^2 + x)^3$.

By viewing the radix conversion as a change of basis, with basic linear algebra it can be derived that

$$\begin{aligned}
f'_0 &= f_0, \\
f'_1 &= f_1 + f_2 + f_3 + f_4 + f_5 + f_6 + f_7, \\
f'_2 &= f_2 + f_3 + f_4 + f_5 + f_6 + f_7, \\
f'_3 &= f_3 + f_5 + f_7, \\
f'_4 &= f_4 + f_6, \\
f'_5 &= f_5 + f_6, \\
f'_6 &= f_6 + f_7, \\
f'_7 &= f_7.
\end{aligned}$$

Let v be an array of length 8 which is initialized such that $v[i] = f_i$. The algorithm described in [GM10] computes f'_i in 8 field additions:

$$\begin{aligned}
v[5] &\leftarrow v[5] \oplus v[7],\\
v[4] &\leftarrow v[4] \oplus v[6],\\
v[3] &\leftarrow v[3] \oplus v[5],\\
v[2] &\leftarrow v[2] \oplus v[4],\\
v[2] &\leftarrow v[2] \oplus v[3],\\
v[1] &\leftarrow v[1] \oplus v[2],\\
v[6] &\leftarrow v[6] \oplus v[7],\\
v[5] &\leftarrow v[5] \oplus v[6].
\end{aligned}$$

To see how this works, it is crucial to see that each

$$v[i] \leftarrow v[i] \oplus v[j]$$

performs a change of basis: the basis element corresponding to $v[i]$ is added to $v[j]$. Indeed, this holds since $c_i b_i + c_j b_j = (c_i + c_j)b_i + c_j(b_i + b_j)$, where $c_i, c_j \in \mathbb{F}_q$ and $b_i, b_j \in \mathbb{F}_q[x]$.

Therefore, after the first 4 field additions, the basis becomes

$$1, x, x^2, x^3, (x^4 + x^2), x(x^4 + x^2), x^2(x^4 + x^2), x^3(x^4 + x^2).$$

Recall that the target basis elements are subset products of $\{x, x^2 + x, (x^2 + x)^2\}$. The first 4 additions handles only the "$(x^2 + x)^2$-part" of the change of basis.

The second part of the algorithm handles the $(x^2 + x)$ part by performing size-4 radix conversions on the first half (elements that contain no $x^4 + x^2$) and the second half (elements that contain $x^4 + x^2$) of the basis. The resulting basis is

$$1, x, x^2 + x, x(x^2 + x), (x^4 + x), x(x^4 + x), (x^2 + x)(x^4 + x), x(x^2 + x)(x^4 + x),$$

which is exactly the target basis. Note that there is no need to handle the "x-part".

I would like to highlight two properties of the algorithm. The first one is that all the operations are done in-place. There is no need to allocate more space than the input/output size. The second property is that all operations performed are of the form $v[i] \leftarrow v[i] \oplus v[j]$ where $i < j$. This interesting property implies that there is no restriction on the polynomial length. For example, we can perform a radix conversion for polynomial length 6, by simply omitting the operations that involve array indices greater or equal to 6 for the above example. These two properties are in fact the properties of the algorithm, not just of this example.

3.4 The radix-conversion subroutine

Now we are ready to generalize the algorithm for arbitrary polynomial length n and radix $x^t + x$ ($t > 1$). Now the goal is, given

$$f_0 + f_1 x + \cdots + f_{n-1} x^{n-1},$$

```
def radix_conversion(L, off, n, t):

  if n <= t:
    return

  k = ceil(log(n/t, 2)) - 1

  for i in reversed(range(t*2^k, n)):
    L[off + i - (t-1)*2^k] += L[off + i]

  radix_conversion(L,         off,     t*2^k, t)
  radix_conversion(L, off + t*2^k, n - t*2^k, t)
```

Figure 3.1: Sage function for the radix conversion for polynomial $f(x)$. At the beginning of the function call, we have $f(x) = \sum_{i=0}^{n-1} \mathrm{L}[\mathrm{off} + i]x^i$. At the end of the function call, we have $f(x) = \sum_{i=0}^{\lceil n/t \rceil - 1} h_i(x)(x^t + x)^i$, where $h_i = \sum_{j=0}^{\min(t-1, n-1 \bmod t)} \mathrm{L}[\mathrm{off} + i \cdot t + j]x^j$.

compute $h_0, h_1, \ldots, h_{\lceil n/t \rceil - 1}$ such that

$$f(x) = \sum_{i=0}^{\lceil n/t \rceil - 1} h_i(x)(x^t + x)^i.$$

Note that we have $\deg(h_i) = n - 1$ if $i < \lceil n/t \rceil - 1$, and $\deg(h_{\lceil n/t \rceil - 1}) = n - 1 \bmod t$. In other words, this is a change of basis from $1, x, \ldots x^{n-1}$ to

$$1, x, \ldots, x^{n-1}, (x^t + x), x(x^t + x), \ldots x^{n-1}(x^t + x), \ldots, x^{n-1 \bmod t}(x^t + x)^{\lceil n/t \rceil - 1}.$$

In the example in Section 3.3, each target basis element is of the form

$$x^i \prod_{j=0}^{k} (x^2 + x)^{2^j e_j},$$

where $i \in \{0, 1\}$ and $e_j \in \{0, 1\}$. For the generic case, we have

$$x^i \prod_{j=0}^{k} (x^t + x)^{2^j e_j},$$

where $i \in \{0, 1, \ldots, t - 1\}$ and $e_j \in \{0, 1\}$. Following the idea in the example, the algorithm starts with handling the "$(x^t + x)^{2^k}$-part" with the largest possible k. Therefore, k is the unique number such that $t \cdot 2^k < n \leq t2^{k+1}$.

After finding k, we can proceed to handle $(x^t + x)^{2^k}$. This is done by performing

$$v[i - (t - 1)2^k] \leftarrow v[i - (t - 1)2^k] + v[i],$$

for i from $n-1$ down to $t2^k$. The basis then has two parts: the first $t2^k$ elements that do not contain $(x^t+x)^{2^k}$, and the remaining $n-t2^k$ elements that contain $(x^t+x)^{2^k}$. Following the idea of the example, we then perform two radix conversions: one for the first $t2^k$ elements and one for the remaining $n-t2^k$ elements. Note that no recursive calls will be triggered when $n \leq t$, since in this case the input to the algorithm is the same as the output. Figure 3.1 shows a Sage implementation for the algorithm.

Gao and Mateer use $t = 2$ for the additive FFT for arbitrary $\mathbb{F}_q = \mathbb{F}_{2^m}$. The additive FFT uses the radix conversion to compute $f^{(0)}$ and $f^{(1)}$ such that $f = f^{(0)}(x^2+x) + xf^{(1)}(x^2+x)$. It can be seen that the coefficients of $f^{(0)}$ and $f^{(1)}$ are coefficients of the target basis.

Part II

Binary-field Cryptography

4

McBits: fast constant-time code-based cryptography

This chapter presents new software speed records for public-key cryptography: for example, more than 400000 decryptions per second at a 2^{80} security level, or 200000 per second at a 2^{128} security level, on a \$215 4-core 3.4GHz Intel Core i5-3570 CPU. These speeds are fully protected against simple timing attacks, cache-timing attacks, branch-prediction attacks, etc.: all load addresses, all store addresses, and all branch conditions are public.

The public-key cryptosystem used here is a code-based cryptosystem with a long history, a well-established security track record, and even post-quantum security: namely, Niederreiter's dual form [Nie86] of McEliece's hidden-Goppa-code cryptosystem [McE78]. This cryptosystem is well known to provide extremely fast encryption and *reasonably* fast decryption. Our main contributions are new decryption techniques that are (1) much faster and (2) fully protected against timing attacks, including the attacks by Strenzke in [Str10], [Str11], and [Str12].

The main disadvantage of this cryptosystem is that public keys are quite large: for example, 64 kilobytes for the 2^{80} security level mentioned above. In some applications the benefits of very fast encryption and decryption are outweighed by the costs of communicating and storing these keys. We comment that our work allows a tradeoff between key size and decryption time: because decryption is so fast we can afford "combinatorial list decoding", using many trial decryptions to guess a few error positions, which allows the message sender to add a few extra error positions (as proposed by Bernstein, Lange, and Peters in [BLP08]), which increases security for the same key size, which allows smaller keys for the same security level.

We also present new speed records for generating signatures in the CFS code-based public-key signature system. Our speeds are an order of magnitude faster than previous work. This system has a much larger public key but is of interest for its short signatures and fast verification.

We will put all software described in this chapter into the public domain.

To bitslice, or not to bitslice. A basic fact in both hardware design and computational complexity theory is that every function from b bits to c bits can be expressed as a fully unrolled combinatorial circuit consisting solely of two-input NAND gates. Converting these NAND gates into "logical" CPU instructions produces fully timing-attack-protected software that computes the same function. One can save a small constant factor by paying attention to the actual selection of logical instructions on each CPU: for example, some CPUs offer single-cycle instructions as complex as a MUX gate. Even better, the same CPU instructions naturally operate on w-bit words in parallel, so in the same time they carry out w separate bitsliced copies of the same computation; common values of w on current CPUs include 8, 16, 32, 64, 128, and 256. For a server bottlenecked by cryptographic computations there is no trouble finding 256 separate computations to carry out in parallel.

However, this approach makes only very limited use of the instruction set of a typical CPU, and it is completely unclear that this approach can produce competitive speeds for typical cryptographic computations. Almost all cryptographic software takes advantage of more complicated CPU instructions: for example, the fastest previous software [BS08] for McEliece/Niederreiter decryption uses input-dependent table lookups for fast field arithmetic, uses input-dependent branches for fast root-finding, etc.

There are a few success stories for bitsliced cryptographic computations, but those stories provide little reason to believe that bitslicing is a good idea for code-based cryptography:

- Biham in [Bih97a] achieved a 2× speedup for DES on an Alpha CPU using bitslicing with $w = 64$. Non-bitsliced implementations of DES used only 64-entry table lookups and were slowed down by frequent bit-level rearrangements. For comparison, code-based cryptography naturally uses much larger tables without such obvious slowdowns.

- Bernstein in [Ber09a], improving upon previous work by Aoki, Hoshino, and Kobayashi in [AHK$^+$01], set speed records for binary-field ECC on a Core 2 CPU using bitslicing with $w = 128$. This paper took advantage of fast multiplication techniques for a large binary field, specifically $\mathbb{F}_{2^{251}}$. For comparison, code-based cryptography uses medium-size fields (such as $\mathbb{F}_{2^{11}}$) with relatively little possibility of savings from fast multiplication.

- Käsper and Schwabe in [KS09], improving upon previous work by Matsui and Nakajima in [MN07], achieved a 1.4× speedup for AES on a Core 2 CPU using bitslicing with $w = 128$. Non-bitsliced implementations of AES use only 256-entry table lookups.

- Bos, Kleinjung, Niederhagen, and Schwabe in [BKN$^+$10] optimized both bitsliced and non-bitsliced implementations of an ECC attack on a Cell CPU, and found that bitslicing was 1.5× faster. Like [Ber09a], this computation took advantage of fast multiplication techniques for a large binary field, specifically $\mathbb{F}_{2^{131}}$.

- Matsuda and Moriai in [MM12] reported fast bitsliced implementations of the PRESENT and Piccolo ciphers on recent Intel CPUs. Non-bitsliced implementations of these ciphers use only 16-entry table lookups.

To summarize, all of these examples of bitsliced speed records are for small S-boxes or large binary fields, while code-based cryptography relies on medium-size fields and seems to make much more efficient use of table lookups.

Despite this background we use bitslicing for the critical decoding step inside McEliece/Niederreiter decryption. Our central observation is that this decoding step is bottlenecked not by separate operations in a medium-size finite field, but by larger-scale polynomial operations over that finite field; state-of-the-art approaches to those polynomial operations turn out to interact very well with bitslicing. Our decoding algorithms end up using a surprisingly small number of bit operations, and as a result a surprisingly small number of cycles, setting new speed records for code-based cryptography, in some cases an order of magnitude faster than previous work.

The most important steps in our decoding algorithm are an “additive FFT” for fast root computation (Section 4.2) and a transposed additive FFT for fast syndrome computation (Section 4.3). It is reasonable to predict that the additive FFT will also reduce the energy consumed by *hardware* implementations of code-based cryptography. We also use a sorting network to efficiently simulate secret-index lookups in a large table (Section 4.4); this technique may be of independent interest for other computations that need to be protected against timing attacks.

Results: the new speeds. To simpify comparisons we have chosen to report benchmarks on a very widely available CPU microarchitecture, specifically the Ivy Bridge microarchitecture from Intel, which carries out one 256-bit vector arithmetic instruction per cycle. We emphasize, however, that our techniques are not limited to this platform. Older Intel and AMD CPUs perform two or three 128-bit vector operations per cycle; common tablet/smartphone ARMs with NEON perform one or two 128-bit vector operations per cycle (exploited by Bernstein and Schwabe in [BS12], although not with bitslicing); the same techniques will also provide quite respectable performance using 64-bit registers, 32-bit registers, etc.

Table 4.1 reports our decoding speeds for various code parameters. Decoding time here is computed as 1/256 of the total latency measured for 256 simultaneous decoding operations. Decryption time is slightly larger, because it requires hashing, checking a MAC, and applying a secret-key cipher; see Section 4.5. We comment that the software supports a separate secret key for each decryption (although many applications do not need this), and that the latency of 256 decryptions is so small as to be unnoticeable in typical applications.

We use the usual parameter notations for code-based cryptography: $q = 2^m$ is the field size, n is the code length, t is the number of errors corrected, and $k = n - mt$. “Bytes” is the public-key size $\lceil k(n-k)/8 \rceil$; the rows are sorted by this column. “Total” is our cycle count (measured by the Ivy Bridge cycle counter with

						Our speeds						
$q = 2^m$	n	t	k	bytes	sec	perm	synd	key eq	root	perm	total	[BS08]
2048	2048	27	1751	65006	81	3333	8414	3120	5986	3199	24051	
2048	1744	35	1359	65402	83	3301	9199	5132	6659	3145	27434	
2048	2048	32	1696	74624	87	3326	9081	4267	6699	3172	26544	445599
2048	2048	40	1608	88440	95	3357	9412	6510	6852	3299	29429	608172
4096	4096	21	3844	121086	87	8661	17496	2259	11663	8826	48903	288649
4096	2480	45	1940	130950	105	8745	21339	9276	14941	8712	63012	
4096	2690	56	2018	169512	119	8733	22898	14199	16383	8789	71000	
4096	4096	41	3604	221646	129	8622	20846	7714	14794	8520	60493	693822
8192	8192	18	7958	232772	91	23331	49344	3353	37315	23339	136679	317421
4096	3408	67	2604	261702	146	8983	24308	19950	17790	8686	79715	
8192	8192	29	7815	368282	128	22879	56336	7709	44727	22753	154403	540952
16384	16384	15	16174	424568	90	60861	99360	2337	79774	60580	302909	467818
8192	4624	95	3389	523177	187	22693	76050	70696	59409	22992	251838	
8192	6624	115	5129	958482	252	23140	83127	102337	65050	22971	296624	
8192	6960	119	5413	1046739	263	23020	83735	109805	66453	23091	306102	

Table 4.1: Number of cycles for decoding for various code parameters. See text for description.

Turbo Boost and hyperthreading disabled) for decoding, including overhead beyond vector operations. This cycle count is partitioned into five stages: “perm” for initial permutation (Section 4.4), “synd” for syndrome computation (Section 4.3), “key eq” for solving the key equation (standard Berlekamp–Massey), “root” for root-finding (Section 4.2), and “perm” again for final permutation.

Some of the parameters in this table are taken from [BLP08], which says that these parameters were designed to optimize security level subject to key sizes of 2^{16}, 2^{17}, 2^{18}, 2^{19}, and 2^{20} bytes. Some parameters are from [HG12]. Some parameters are from [BS08], and for comparison we repeat the Core 2 cycle counts reported in [BS08]. (We comment that the “cycles/byte” in [BS08] are cycles divided by $(k + \lfloor \lg \binom{n}{t} \rfloor)/8$.) Our speedups are much larger than any relevant differences between the Core 2 and the Ivy Bridge that we used for benchmarking; we will report Core 2 cycle counts for our software in a subsequent online update of this chapter. “Sec” is the approximate security level reported by the `https://bitbucket.org/cbcrypto/isdfq` script from Peters [Pet10], rounded to the nearest integer.

Some of the parameter choices from [BS08] are uninteresting in all of our metrics: they are beaten by other parameter choices in key size, speed, and security level. For these parameter choices we mark our cycle count in gray. Note that we have taken only previously published parameter sets; in particular, we have not searched for parameters that sacrifice key size to improve speed for the same security level.

Previous speeds for public-key cryptography. The eBATS benchmarking system [BL13] includes seven public-key encryption systems: `mceliece`, a McEliece implementation from Biswas and Sendrier (with $n = 2048$ and $t = 32$, slightly above a 2^{80} security level); `ntruees787ep1`, an NTRU implementation (2^{256} security) from Mark Etzel; and five sizes of RSA starting from `ronald1024` (2^{80} security). None of these implementations claim to be protected against timing attacks. On `h9ivy`, an Ivy Bridge CPU (Intel Core i5-3210M), the fastest encryption (for 59-byte messages) is 46940 cycles for `ronald1024` followed by 61440 cycles for `mceliece`, several more RSA results, and finally 398912 cycles for `ntruees787ep1`. The fastest decryption is 700512 cycles for `ntruees787ep1`, followed by 1219344 cycles for `mceliece` and 1340040 cycles for `ronald1024`.

A followup paper [BS08] by Biswas and Sendrier reports better decryption performance, 445599 cycles on a Core 2 for $n = 2048$ and $t = 32$. Sendrier says (private communication) that he now has better performance, below 300000 cycles. However, our speed of 26544 cycles for $n = 2048$ and $t = 32$ improves upon this by an order of magnitude, and also includes full protection against timing attacks.

eBATS also includes many Diffie–Hellman systems. One can trivially use Diffie–Hellman for public-key encryption; the decryption time is then the Diffie–Hellman shared-secret time plus some fast secret-key cryptography, and the encryption time is the same plus the Diffie–Hellman key-generation time. When we submitted this chapter the fastest Diffie–Hellman shared-secret time reported from `h9ivy` was 182632 cycles (side-channel protected), set by the `curve25519` implementation from Bernstein, Duif, Lange, Schwabe, and Yang in [BDL+11]. The fastest time now is 77468 cycles (not side-channel protected), set by `gls254` from Oliveira, López, Aranha, and Rodríguez-Henríquez; see [OLA+13a]. Our software takes just 60493 cycles (side-channel protected) for decryption with $n = 4096$ and $t = 41$ at the same 2^{128} security level.

We have found many claims that NTRU is orders of magnitude faster than RSA and ECC, but we have also found no evidence that NTRU can match our speeds. The fastest NTRU decryption report that we have found is from Hermans, Vercauteren, and Preneel in [HVP10]: namely, 24331 operations per second on a GTX 280 GPU.

Heyse and Güneysu in [HG12] report 17012 Niederreiter decryption operations per second on a Virtex6-LX240T FPGA for $n = 2048$ and $t = 27$. The implementation actually uses only 10% of the FPGA slices, so presumably one can run several copies of the implementation in parallel without running into place-and-route difficulties. A direct speed comparison between such different platforms does not convey much information, but we point out several ways that our decryption algorithm improves upon the algorithm used in [HG12]: we use an additive FFT rather than separate evaluations at each point ("Chien search"); we use a transposed additive FFT rather than applying a syndrome-conversion matrix; we do not even need to store the syndrome-conversion matrix, the largest part of the data stored in [HG12]; and we use a simple hash (see Section 4.5) rather than a constant-weight-word-to-bit-string conversion.

4.1 Field arithmetic

We construct the finite field $\mathbb{F}_{2^m}$ as $\mathbb{F}_2[x]/f$, where f is a degree-m irreducible polynomial. We use trinomial choices of f when possible. We use pentanomials for $\mathbb{F}_{2^{13}}$ and $\mathbb{F}_{2^{16}}$.

4.1.1 Addition

Addition in $\mathbb{F}_{2^m}$ is simply a coefficient-wise xor and costs m bit operations.

4.1.2 Multiplication

A field multiplication is composed of a multiplication in $\mathbb{F}_2[x]$ and reduction modulo f. We follow the standard approach of optimizing these two steps separately, and we use standard techniques for the second step. Note, however, that this two-step optimization is not necessarily optimal, even if each of the two steps is optimal.

For the first step we started from Bernstein's straight-line algorithms from `http://binary.cr.yp.to/m.html`. The mth algorithm is a sequence of XORs and ANDs that multiplies two m-coefficient binary polynomials. The web page shows algorithms for m as large as 1000; for McEliece/Niederreiter we use m between 11 and 16, and for CFS (Section 4.6) we use $m = 20$. These straight-line algorithms are obtained by combining different multiplication techniques as explained in [Ber09a]; for $10 \le m \le 20$ the algorithms use somewhat fewer bit operations than schoolbook multiplication. We applied various scheduling techniques (in some cases sacrificing some bit operations) to improve cycle counts.

4.1.3 Squaring

Squaring of a polynomial does not require any bit operations. The square of an m-coefficient polynomial $f = \sum_{i=0}^{m-1} a_i x^i$ is simply $f^2 = \sum_{i=0}^{m-1} a_i x^{2i}$. The only bit operations required for squaring in $\mathbb{F}_{2^m}$ are thus those for reduction. Note that half of the high coefficients are known to be zero; reduction after squaring takes only about half the bit operations of reduction after multiplication.

4.1.4 Inversion

We compute reciprocals in $\mathbb{F}_{2^m}$ as $(2^m - 2)$nd powers. For $\mathbb{F}_{2^{20}}$ we use an addition chain consisting of 19 squarings and 6 multiplications. For smaller fields we use similar addition chains.

4.2 Finding roots: the Gao–Mateer additive FFT

This section considers the problem of finding all the roots of a polynomial over a characteristic-2 finite field. This problem is parametrized by a field size $q = 2^m$ where m is a positive integer. The input is a sequence of coefficients $c_0, c_1, \ldots, c_t \in \mathbb{F}_q$ of a polynomial $f = c_0 + c_1 x + \cdots + c_t x^t \in \mathbb{F}_q[x]$ of degree at most t. The output is a sequence of q bits b_α indexed by elements $\alpha \in \mathbb{F}_q$ in a standard order, where $b_\alpha = 0$ if and only if $f(\alpha) = 0$.

4.2.1 Application to decoding

Standard decoding techniques have two main steps: finding an "error-locator polynomial" f of degree at most t, and finding all the roots of the polynomial in a specified finite field $\mathbb{F}_q$. In the McEliece/Niederreiter context it is traditional to take the field size q as a power of 2 and to take t on the scale of $q/\lg q$, typically between $0.1q/\lg q$

and $0.3q/\lg q$; a concrete example is $(q,t) = (2048, 40)$. In cases of successful decryption this polynomial will in fact have exactly t roots at the positions of errors added by the message sender.

4.2.2 Multipoint evaluation

In coding theory, and in code-based cryptography, the most common way to solve the root-finding problem is to simply try each possible root: for each $\alpha \in \mathbb{F}_q$, evaluate $f(\alpha)$ and then OR together the bits of $f(\alpha)$ in a standard basis, obtaining 0 if and only if $f(\alpha) = 0$.

The problem of evaluating $f(\alpha)$ for every $\alpha \in \mathbb{F}_q$, or more generally for every α in some set S, is called multipoint evaluation. Separately evaluating $f(\alpha)$ by Horner's rule for every $\alpha \in \mathbb{F}_q$ costs qt multiplications in $\mathbb{F}_q$ and qt additions in $\mathbb{F}_q$; if t is essentially linear in q (e.g., q or $q/\lg q$) then the total number of field operations is essentially quadratic in q. "Chien search" is an alternative method of evaluating each $f(\alpha)$, also using qt field additions and qt field multiplications.

There is an extensive literature on more efficient multipoint-evaluation techniques. Most of these techniques (for example, the "dcmp" method recommended by Strenzke in [Str12]) save at most small constant factors. Some of them are much more scalable: in particular, a 40-year-old FFT-based algorithm [BM74] by Borodin and Moenck evaluates an n-coefficient polynomial at any set of n points using only $n^{1+o(1)}$ field operations. On the other hand, the conventional wisdom is that FFTs are particularly clumsy for characteristic-2 fields, and in any case are irrelevant to the input sizes that occur in cryptography.

For multipoint evaluation we use the Gao–Mateer additive FFT algorithm decribed in Chapter 3. We show some new improvements below, which are spcialized for the context of decoding.

4.2.3 FFT improvement: 1-coefficient polynomials

Gao and Mateer show that for $q = 2^m$ this additive-FFT algorithm uses $2q\lg q - 2q + 1$ multiplications in $\mathbb{F}_q$ and $(1/4)q(\lg q)^2 + (3/4)q\lg q - (1/2)q$ additions in $\mathbb{F}_q$. The $\beta_m = 1$ optimization removes many multiplications when it is applicable.

We do better by generalizing from one parameter to two, separating the maximum polynomial degree t from the number 2^m of evaluation points. Our main interest is not in the case $t + 1 = 2^m$, but in the case that t is smaller than 2^m by a logarithmic factor.

The adjustments to the algorithm are straightforward. We begin with a polynomial having $t+1$ coefficients. If $t = 0$ then the output is simply 2^m copies of $f(0)$, which we return immediately without any additions or multiplications. If $t \geq 1$ then we continue as in the algorithm above; $f^{(0)}$ has $\lceil (t+1)/2 \rceil$ coefficients, and $f^{(1)}$ has $\lfloor (t+1)/2 \rfloor$ coefficients. Note that $t + 1$ and 2^m each drop by a factor of approximately 2 in the recursive calls.

It is of course possible to zero-pad a $(t+1)$-coefficient polynomial to a 2^m-coefficient polynomial and apply the original algorithm, but this wastes considerable time manipulating coefficients that are guaranteed to be 0.

4.2.4 FFT improvement: 2-coefficient and 3-coefficient polynomials

We further accelerate the case that t is considerably smaller than 2^m, replacing many multiplications with additions as follows.

Recall that the last step of the algorithm involves 2^{m-1} multiplications of the form $\alpha f^{(1)}(\gamma)$. Here α runs through all subset sums of $\beta_1, \beta_2, \ldots, \beta_{m-1}$, and $\gamma = \alpha^2 - \alpha$. The multiplication for $\alpha = 0$ can be skipped but all other multiplications seem nontrivial.

Now consider the case that $t \in \{1, 2\}$. Then $f^{(1)}$ has just 1 coefficient, so the recursive evaluation of $f^{(1)}$ produces 2^{m-1} copies of $f^{(1)}(0)$, as discussed above. The products $\alpha f^{(1)}(\gamma) = \alpha f^{(1)}(0)$ are then nothing more than subset sums of $\beta_1 f^{(1)}(0)$, $\beta_2 f^{(1)}(0)$, $\ldots$, $\beta_{m-1} f^{(1)}(0)$. Instead of $2^{m-1} - 1$ multiplications we use just $m - 1$ multiplications and $2^{m-1} - m$ additions.

4.2.5 Results

Table 4.2 displays the speed of the additive FFT, including these improvements, for an illustrative sample of field sizes $q = 2^m$ and degrees t taken from our applications to decoding.

4.2.6 Other algorithms

We briefly mention a few alternative root-finding algorithms.

In the standard McEliece/Niederreiter context, f is known in advance to have t distinct roots (for valid ciphertexts). However, in the signing context of Section 4.6 and the "combinatorial list decoding" application mentioned in Section 4.5, one frequently faces, and wants to discard, polynomials f that do not have t distinct roots. One can usually save time by checking whether $x^q - x \bmod f = 0$ before applying a root-finding algorithm. There are other applications where one wants all the roots of a polynomial f that has no reason to have as many as $\deg f$ distinct roots; for such applications it is usually helpful to replace f with $\gcd f, x^q - x$.

There are other root-finding techniques (and polynomial-factorization techniques) that scale well to very large finite fields $\mathbb{F}_q$ when t remains small, such as Berlekamp's trace algorithm [Ber70]. If t is as large as q then all of these techniques are obviously slower than multipoint evaluation with the additive FFT, but our experiments indicate that the t cutoff is above the range used in code-based signatures (see Section 4.6) and possibly within the range used in code-based encryption. Our main reason for not using these methods is that they involve many data-dependent conditional branches; as far as we can tell, all of these methods become much slower when the branches are eliminated.

There is a generalization of the additive FFT that replaces $x^2 - x$ with $x^t - x$ if q is a power of t. Gao and Mateer state this generalization only in the extreme case that $\lg q$ and $\lg t$ are powers of 2; we are exploring the question of whether the generalization produces speedups for other cases.

$m = 11$	t	27	32	35	40	53	63	69	79		
	adds	5.41	5.60	5.75	5.99	6.47	6.69	6.84	7.11		
	mults	1.85	2.12	2.13	2.16	2.40	2.73	2.77	2.82		
$m = 12$	t	21	41	45	56	67	81	89	111	133	
	adds	5.07	6.01	6.20	6.46	6.69	7.04	7.25	7.59	7.86	
	mults	1.55	2.09	2.10	2.40	2.64	2.68	2.70	2.99	3.28	
$m = 13$	t	18	29	35	57	95	115	119	189	229	237
	adds	4.78	5.45	5.70	6.44	7.33	7.52	7.56	8.45	8.71	8.77
	mults	1.52	1.91	2.04	2.38	2.62	2.94	3.01	3.24	3.57	3.64

Table 4.2: Number of field operations/point in the additive FFT for various field sizes $q = 2^m$ and various parameters t. The total number of field additions is q times "adds"; the total number of field multiplications is q times "mults". For comparison, Horner's rule uses qt additions and qt multiplications; i.e., for Horner's rule, "adds" and "mults" are both t. Chien search also uses qt additions and qt multiplications.

4.3 Syndrome computation: transposing the additive FFT

Consider the problem of computing the vector $(\sum_\alpha r_\alpha, \sum_\alpha r_\alpha \alpha, \dots, \sum_\alpha r_\alpha \alpha^d)$, given a sequence of q elements $r_\alpha \in \mathbb{F}_q$ indexed by elements $\alpha \in \mathbb{F}_q$, where $q = 2^m$. This vector is called a "syndrome". One can compute $\sum_\alpha r_\alpha \alpha^i$ separately for each i with approximately $2dq$ field operations. We do better in this section by merging these computations across all the values of i.

4.3.1 Application to decoding

The standard Berlekamp decoding algorithm computes the syndrome shown above, and then solves a "key equation" to compute the error-locator polynomial mentioned in Section 4.2. When Berlekamp's algorithm is applied to decoding Goppa codes using a degree-t polynomial g as described in Section 4.5, the inputs r_α are a received word divided by $g(\alpha)^2$, and d is $2t - 1$. Many other decoding algorithms begin with the same type of syndrome computation, often with d only half as large.

Note that there are only $n \leq q$ bits in the received word. The $(d + 1)m = 2tm$ syndrome bits are $\mathbb{F}_2$-linear functions of these n input bits. Standard practice in the literature is to precompute the corresponding $2tm \times n$ matrix (or a $tm \times n$ matrix for Patterson's algorithm), and to multiply this matrix by the n input bits to obtain the syndrome. These $2tmn$ bits are by far the largest part of the McEliece/Niederreiter secret key. Our approach eliminates this precomputed matrix, and also reduces the number of bit operations once t is reasonably large.

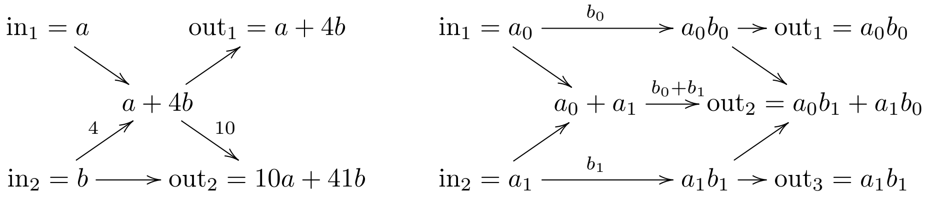

Figure 4.1: An $\mathbf{R}$-linear algorithm to compute $a, b \mapsto a + 4b, 10a + 41b$ using constants $4, 10$, and an $\mathbb{F}_{2^m}$-linear algorithm to compute $a_0, a_1 \mapsto a_0b_0, a_0b_1 + a_1b_0, a_1b_1$ using constants $b_0, b_0 + b_1, b_1$.

4.3.2 Syndrome computation as the transpose of multipoint evaluation

Notice that the syndrome $(c_0, c_1, \ldots, c_d)$ is an $\mathbb{F}_q$-linear function of the inputs r_α. The syndrome-computation matrix is a "transposed Vandermonde matrix": the coefficient of r_α in c_i is α^i.

For comparison, consider the multipoint-evaluation problem stated in the previous section, producing $f(\alpha)$ for every $\alpha \in \mathbb{F}_q$ given a polynomial $f = c_0 + c_1x + \cdots + c_dx^d$. The multipoint-evaluation matrix is a "Vandermonde matrix": the coefficient of c_i in $f(\alpha)$ is α^i.

To summarize, the syndrome-computation matrix is exactly the transpose of the multipoint-evaluation matrix. We show below how to exploit this fact to obtain a fast algorithm for syndrome computation.

4.3.3 Transposing linear algorithms

A *linear algorithm* expresses a linear computation as a labeled acyclic directed graph. Each edge in the graph is labeled by a constant (by default 1 if no label is shown), multiplies its incoming vertex by that constant, and adds the product into its outgoing vertex; some vertices without incoming edges are labeled as inputs, and some vertices without outgoing edges are labeled as outputs. Figure 4.1 displays two examples: a computation of $a+4b, 10a+41b$ given a, b, using constants 4 and 10; and a computation of $a_0b_0, a_0b_1 + a_1b_0, a_1b_1$ given a_0, a_1, using constants $b_0, b_0 + b_1, b_1$.

The *transposition principle* states that if a linear algorithm computes a matrix M (i.e., M is the matrix of coefficients of the inputs in the outputs) then reversing the edges of the linear algorithm, and exchanging inputs with outputs, computes the transpose of M. This principle was introduced by Bordewijk in [Bor56], and independently by Lupanov in [Lup56] for the special case of Boolean matrices. This reversal preserves the number of multiplications (and the constants used in those multiplications), and preserves the number of additions plus the number of nontrivial outputs, as shown by Fiduccia in [Fid73, Theorems 4 and 5] after preliminary work in [Fid72].

For example, Figure 4.2 displays the reversals of the linear algorithms in Figure 4.1.

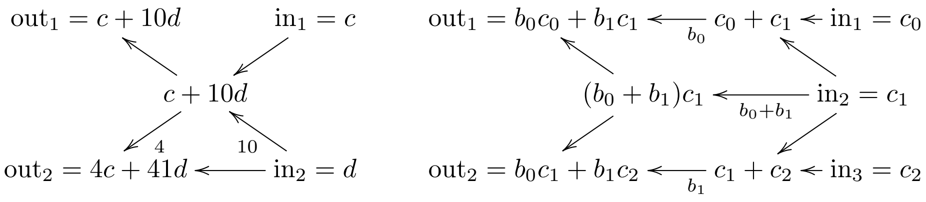

Figure 4.2: Transposing the algorithms in Figure 4.1.

The first reversal computes $c+10d, 4c+41d$ given c, d. The second reversal computes $b_0c_0 + b_1c_1, b_0c_1 + b_1c_2$ given c_0, c_1, c_2.

4.3.4 Transposing the additive FFT

In particular, since syndrome computation is the transpose of multipoint evaluation, reversing a fast linear algorithm for multipoint evaluation produces a fast linear algorithm for syndrome computation.

We started with our software for the additive FFT, including the improvements discussed in Section 4.2. This software is expressed as a sequence of additions in $\mathbb{F}_q$ and multiplications by various constants in $\mathbb{F}_q$. We compiled this sequence into a directed acyclic graph, automatically renaming variables to avoid cycles. We then reversed the edges in the graph and converted the resulting graph back into software expressed as a sequence of operations in $\mathbb{F}_q$, specifically C code with vector intrinsics.

This procedure produced exactly the desired number of operations in $\mathbb{F}_q$ but was unsatisfactory for two reasons. First, there were a huge number of nodes in the graph, producing a huge number of variables in the final software. Second, this procedure eliminated all of the loops and functions in the original software, producing a huge number of lines of code in the final software. Consequently the C compiler, `gcc`, became very slow as m increased and ran out of memory around $m = 13$ or $m = 14$, depending on the machine we used for compilation.

We then tried the `qhasm` register allocator [Ber07b], which was able to produce working code for larger values of m using the expected number of variables (essentially q), eliminating the first problem. We then wrote our own faster straight-line register allocator. We reduced code size by designing a compact format for the sequence of $\mathbb{F}_q$ operations and interpreting the sequence at run time. There was, however, still some performance overhead for this interpreter.

We considered more advanced compilation techniques to reduce code size: the language introduced in [FS10], for example, and automatic compression techniques to recognize repeated subgraphs of the reversed graph. In the end we eliminated the compiler, analyzed the interaction of transposition with the structure of the additive FFT, and designed a compact transposed additive FFT algorithm.

The original additive FFT algorithm A has steps of the form B, A_1, A_2, C, where A_1 and A_2 are recursive calls. The transpose $A^{\intercal}$ has steps $C^{\intercal}, A_2^{\intercal}, A_1^{\intercal}, B^{\intercal}$, preserving the recursions. The main loop in the additive FFT takes a pair of variables v, w (containing $f_0(\alpha^2-\alpha)$ and $f_1(\alpha^2-\alpha)$ respectively), operates in place on those variables (producing $f(\alpha)$ and $f(\alpha+1)$ respectively), and then moves on to the next pair of variables; transposition preserves this loop structure and simply transposes each operation. This operation replaces v by $v + w \cdot \alpha$ and then replaces w by $w + v$; the transposed operation replaces v by $v + w$ and then replaces w by $w + v \cdot \alpha$.

4.3.5 Improvement: transposed additive FFT on scaled bits

Recall that, in the decoding context, the inputs are not arbitrary field elements: r_α is a received bit divided by $g(\alpha)^2$. We take advantage of this restriction to reduce the number of bit operations in syndrome computation.

The first step of the transposed additive FFT operates on each successive pair of inputs v, w as described above: it replaces v by $v + w$ and then replaces w by $w + v \cdot \alpha$. Assume that before this v, w are computed as scaled bits $b_v \cdot s_v, b_w \cdot s_w$, where $b_v, b_w \in \mathbb{F}_2$ are variables and $s_v, s_w \in \mathbb{F}_q$ are constants. Computing $b_v \cdot s_v$ and $b_w \cdot s_w$ takes $2m$ bit operations; computing $w \cdot \alpha$ takes one field multiplication; computing $v + w \cdot \alpha$ takes m bit operations; computing $w + v$ takes m bit operations.

If the multiplication by α takes more than $2m$ bit operations then we do better by computing the final v and w directly as $b_v \cdot s_v + b_w \cdot s_w$ and $b_v \cdot s_v \cdot \alpha + b_w \cdot s_w \cdot (\alpha + 1)$ respectively. This takes just $6m$ bit operations: we precompute $s_v, s_w, s_v \cdot \alpha, s_w \cdot (\alpha+1)$.

The same idea can be used for more levels of recursion, although the number of required constants grows rapidly. Using this idea for all levels of recursion is tantamount to the standard approach mentioned earlier, namely precomputing a $2tm \times n$ matrix.

4.4 Secret permutations without secret array indices: odd-even sorting

Section 4.2 presented an algorithm that, given a polynomial f, outputs bits b_α for all $\alpha \in \mathbb{F}_q$ *in a standard order* (for example, lexicographic order using a standard basis), where $b_\alpha = 0$ if and only if $f(\alpha) = 0$. However, in the McEliece/Niederreiter context, one actually has the elements $(\alpha_1, \alpha_2, \ldots, \alpha_q)$ of $\mathbb{F}_q$ *in a secret order* (or, more generally, $(\alpha_1, \ldots, \alpha_n)$ for some $n \leq q$), and one needs to know for each i whether $f(\alpha_i) = 0$, i.e., whether $b_{\alpha_i} = 0$. These problems are not exactly the same: one must apply a secret permutation to the q bits output by Section 4.2. Similar comments apply to Section 4.3: one must apply the inverse of the same secret permutation to the q bits input to Section 4.3.

This section considers the general problem of computing a permuted q-bit string $b_{\pi(0)}, b_{\pi(1)}, \ldots, b_{\pi(q-1)}$, given a q-bit string $b_0, b_1, \ldots, b_{q-1}$ and a sequence of q distinct integers $\pi(0), \pi(1), \ldots, \pi(q-1)$ in $\{0, 1, \ldots, q-1\}$. Mapping the set $\{0, 1, \ldots, q-1\}$ to $\mathbb{F}_q$ in a standard order, and viewing α_{i+1} as either $\pi(i)$ or $\pi^{-1}(i)$, covers the problems stated in the previous paragraph.

The obvious approach is to compute $b_{\pi(i)}$ for $i = 0$, then for $i = 1$, etc. We require all load and store addresses to be public, so we cannot simply use the CPU's load instruction (with appropriate masking) to pick up the bit $b_{\pi(i)}$. Bitslicing can simulate this load instruction, essentially by imitating the structure of physical RAM hardware, but this is very slow: it means performing a computation involving every element of the array. We achieve much better bitslicing speeds by batching all of the required loads into a single large operation as described below.

4.4.1 Sorting networks

A "sorting network" uses a sequence of "comparators" to sort an input array S. A comparator is a data-independent pair of indices (i, j); it swaps $S[i]$ with $S[j]$ if $S[i] > S[j]$. This conditional swap is easily expressed as a data-independent sequence of bit operations: first some bit operations to compute the condition $S[i] > S[j]$, then some bit operations to overwrite $(S[i], S[j])$ with $(\min S[i], S[j], \max S[i], S[j])$.

There are many sorting networks in the literature. We use a standard "odd-even" sorting network by Batcher [Bat68], which uses exactly $(m^2 - m + 4)2^{m-2} - 1$ comparators to sort an array of 2^m elements. This is more efficient than other sorting networks such as Batcher's bitonic sort [Bat68] or Shell sort [She59]. The odd-even sorting network is known to be suboptimal when m is very large (see [AKS83]), but we are not aware of noticeably smaller sorting networks for the range of m used in code-based cryptography.

4.4.2 Precomputed comparisons

We treat this section's $b_{\pi(i)}$ computation as a sorting problem: specifically, we use a sorting network to sort the key-value pairs $(\pi^{-1}(0), b_0), (\pi^{-1}(1), b_1), \ldots$ according to the keys. Note that computing $(\pi^{-1}(0), \pi^{-1}(1), \ldots)$ from $(\pi(0), \pi(1), \ldots)$ can be viewed as another sorting problem, namely sorting the key-value pairs $(\pi(0), 0)$, $(\pi(1), 1)$, . . . according to the keys.

We do better by distinguishing between the b-dependent part of this computation and the b-independent part of this computation: we precompute everything b-independent before b is known. In the context of code-based cryptography, the permutations π and π^{-1} are known at key-generation time and are the same for every use of the secret key. The only computations that need to be carried out for each decryption are computations that depend on b.

Specifically, all of the comparator conditions $S[i] > S[j]$ depend only on π, not on b; the conditional swaps of π values also depend only on π, not on b. We record the $(m^2 - m + 4)2^{m-2} - 1$ comparator conditions obtained by sorting π, and then apply those conditional swaps to the b array once b is known. Conditionally swapping $b[i]$ with $b[j]$ according to a bit c uses only 4 bit operations ($y \leftarrow b[i] \oplus b[j]$; $y \leftarrow cy$; $b[i] \leftarrow b[i] \oplus y$; $b[j] \leftarrow b[j] \oplus y$), for a total of $4((m^2 - m + 4)2^{m-2} - 1)$ bit operations. Note that applying the same conditional swaps in reverse order applies the inverse permutation.

4.4.3 Permutation networks

A "permutation network" (or "rearrangeable permutation network" or "switching network") uses a sequence of conditional swaps to apply an arbitrary permutation to an input array S. Here a conditional swap is a data-independent pair of indices (i, j) together with a permutation-dependent bit c; it swaps $S[i]$ with $S[j]$ if $c = 1$.

A sorting network, together with a permutation, produces a limited type of permutation network in which the condition bits are computed by data-independent comparators; but there are other types of permutation networks in which the condition bits are computed in more complicated ways. In particular, the Beneš permutation network [Ben65] uses only $2^m(m - 1/2)$ conditional swaps to permute 2^m elements for $m \geq 1$.

The main challenge in using the Beneš permutation network is to compute the condition bits in constant time; see Section 4.5 for further discussion of timing-attack protection for key generation. We have recently completed software for this condition-bit computation but have not yet integrated it into our decoding software. We will report the details of this computation, and the resulting speeds, in an online update of this chapter.

4.4.4 Alternative: random condition bits

In code-based cryptography we choose a permutation at random; we then compute the condition bits for a permutation network, and later (during each decryption) apply the conditional swaps. An alternative is to first choose a random sequence of condition bits for a permutation network, then compute the corresponding permutation, and later apply the conditional swaps.

This approach reduces secret-key size but raises security questions. By definition a permutation network can reach every permutation, but perhaps it is much more likely to reach some permutations than others. Perhaps this hurts security. Perhaps not; perhaps a nearly uniform distribution of permutations is unnecessary; perhaps it is not even necessary to reach all permutations; perhaps a network half the size of the Beneš network would produce a sufficiently random permutation; but these speculations need security analysis. Our goals in this chapter are more conservative, so we avoid this approach: we are trying to reduce, not increase, the number of questions for cryptanalysts.

4.5 A complete code-based cryptosystem

Code-based cryptography is often presented as encrypting fixed-length plaintexts. McEliece encryption multiplies the public key (a matrix) by a k-bit message to produce an n-bit codeword and adds t random errors to the codeword to produce a ciphertext. The Niederreiter variant (which has several well-known advantages, and which we use) multiplies the public key by a weight-t n-bit message to produce an $(n - k)$-bit ciphertext. If the t-error decoding problem is difficult for the public code then both of these encryption systems are secure against passive attackers who intercept valid ciphertexts for random plaintexts.

What users want, however, is to be able to encrypt *non-random* plaintexts of *variable length* and to be secure against *active* attackers who observe the receiver's responses to *forged* ciphertexts. The literature contains several different ways to convert the McEliece encryption scheme into this more useful type of encryption scheme, with considerable attention paid to

- the ciphertext overhead (ciphertext length minus plaintext length) and
- the set of attacks that are proven to be as difficult as the t-error decoding problem (e.g., generic-hash IND-CCA2 attacks in [KI01]).

However, much less attention has been paid to

- the cost in encryption time,
- the cost in decryption time, and
- security against timing attacks.

The work described in previous sections of this chapter, speeding up t-error decoding and protecting it against timing attacks, can easily be ruined by a conversion that is slow or that adds its own timing leaks. We point out, for example, that straightforward implementations of any of the decryption procedures presented in [KI01] would abort if the "$\mathcal{D}^{\mathrm{McEliece}}$" step fails; the resulting timing leak allows all of the devastating attacks that [KI01] claims to eliminate.

This section specifies a fast code-based public-key encryption scheme that provides high security, including security against timing attacks. This section also compares the scheme to various alternatives.

4.5.1 Parameters

The system parameters are positive integers m, q, n, t, k such that $n \le q = 2^m$, $k = n - mt$, and $t \ge 2$. For example, one can take $m = 12$, $n = q = 4096$, $t = 41$, and $k = 3604$.

4.5.2 Key generation

The receiver's secret key has two parts: first, a sequence $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ of distinct elements of $\mathbb{F}_q$; second, a squarefree degree-t polynomial $g \in \mathbb{F}_q[x]$ such that $g(\alpha_1)g(\alpha_2)\cdots g(\alpha_n) \ne 0$. These can of course be generated dynamically from a much smaller secret.

The receiver computes the $t \times n$ matrix

$$\begin{pmatrix} 1/g(\alpha_1) & 1/g(\alpha_2) & \cdots & 1/g(\alpha_n) \\ \alpha_1/g(\alpha_1) & \alpha_2/g(\alpha_2) & \cdots & \alpha_n/g(\alpha_n) \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{t-1}/g(\alpha_1) & \alpha_2^{t-1}/g(\alpha_2) & \cdots & \alpha_n^{t-1}/g(\alpha_n) \end{pmatrix}$$

over $\mathbb{F}_q$. The receiver then replaces each entry in this matrix by a column of m bits in a standard basis of $\mathbb{F}_q$ over $\mathbb{F}_2$, obtaining an $mt \times n$ matrix H over $\mathbb{F}_2$. The kernel of H, i.e., the set of $c \in \mathbb{F}_2^n$ such that $Hc = 0$, is a vector space of dimension at least $n - mt = k$, namely the Goppa code $\Gamma = \Gamma_2(\alpha_1, \ldots, \alpha_n, g)$.

At this point one can compute the receiver's public key K by applying Gaussian elimination (with partial pivoting) to H. Specifically, K is the result of applying a sequence of elementary row operations to H (adding one row to another row), and is the unique result in systematic form, i.e., the unique result whose left $tm \times tm$ submatrix is the identity matrix. One can trivially compress K to $(n - mt)mt = k(n-k)$ bits by not transmitting the identity matrix; this compression was introduced by Niederreiter in [Nie86], along with the idea of using a systematic parity-check matrix for Γ instead of a random parity-check matrix for Γ. If Gaussian elimination fails (i.e., if the left $tm \times tm$ submatrix of H is not invertible) then the receiver starts over, generating a new secret key; approximately 3 tries are required on average.

The standard approach to Gaussian elimination is to search for a 1 in the first column (aborting if there is no 1), then swap that row with the first row, then subtract that row from all other rows having a 1 in the first column, then continue similarly through the other columns. This approach has several timing leaks in the success cases. (It also takes variable time in the failure cases, but those cases are independent of the final secret.) We eliminate the timing leaks in the success cases as follows, with only a small constant-factor overhead. We add $1 - b$ times the second row to the first row, where b is the first entry in the first row; and then similarly (with updated b) for the third row etc. We then add b times the first row to the second row, where b is the first entry in the second row; and then similarly for the third row etc. We then continue similarly through the other columns.

An alternate strategy is to first apply a reasonably long sequence of elementary row operations to H, using a public sequence of rows but secret random multiples. Here "reasonably long" is chosen so that the output is negligibly different from a uniform random parity-check matrix for the same code. That parity-check matrix can safely be made public, so one can feed it to any Gaussian-elimination routine to obtain K, even if the Gaussian-elimination routine leaks information about its input through timing.

One can argue that key generation provides the attacker only a single timing trace (for the secret key that ends up actually being used), and that this single trace is not enough information to pinpoint the secret key. However, this argument relies implicitly on a detailed analysis of how much information the attacker actually obtains through timing. By systematically eliminating all timing leaks we eliminate the need for such arguments and analyses.

4.5.3 Encryption

To encrypt a variable-length message we generate a random 256-bit key for a stream cipher and then use the cipher to encrypt the message. AES-CTR has fast constant-time implementations for some platforms but not for others, so we instead choose Salsa20 [Ber08b] as the stream cipher. To eliminate malleability we also generate a random 256-bit key for the Poly1305 MAC [Ber05], which takes time dependent only on the message length, and use this MAC to authenticate the ciphertext.

To generate these two secret keys we generate a random weight-t vector $e \in \mathbb{F}_2^n$ and then hash the vector to 512 bits. For the moment we use SHA-512 as the hash function; according to [BDP$^+$13] it is still not yet clear exactly which Keccak variants will be specified for SHA-3. All of these hash functions take constant time for fixed n.

To transmit the vector e to the receiver we compute and send $w = Ke \in \mathbb{F}_2^{tm}$. The ciphertext overhead is tm bits for w, plus 128 bits for the authenticator.

Note that we are following Shoup's "KEM/DEM" approach (see [Sho01]) rather than the classic "hybrid" approach. The hybrid approach (see, e.g., [OS09, Section 5.1]) is to first generate random secret keys, then encode those secret keys (with appropriate padding) as a weight-t vector e. The KEM/DEM approach is to first generate a weight-t vector e and then hash that vector to obtain random secret keys. The main advantage of the KEM/DEM approach is that there is no need for the sender to encode strings injectively as weight-t vectors, or for the receiver to decode weight-t vectors into strings. The sender does have to generate a random weight-t vector, but this is relatively easy since there is no requirement of injectivity.

A security proof for Niederreiter KEM/DEM appeared very recently in Persichetti's thesis [Per12]. The proof assumes that the t-error decoding problem is hard; it also assumes that a decoding failure for w is indistinguishable from a subsequent MAC failure. This requires care in the decryption procedure; see below.

4.5.4 Decryption

A ciphertext has the form (a, w, c) where $a \in \mathbb{F}_2^{128}$, $w \in \mathbb{F}_2^{tm}$, and $c \in \mathbb{F}_2^*$. The receiver decodes w (as discussed below) to obtain a weight-t vector $e \in \mathbb{F}_2^n$ such that $w = Ke$, hashes e to obtain a Salsa20 key and a Poly1305 key, verifies that a is the Poly1305 authenticator of c, and finally uses Salsa20 to decrypt c into the original plaintext.

Our decoding procedure is a constant-time sequence of bit operations and always outputs a vector e, even if w does not actually have the form Ke. With a small extra cost we also compute, in constant time, an extra bit indicating whether decoding succeeded. We continue through the hashing and authenticator verification in all cases, mask the authenticator-valid bit with the decoding-succeeded bit, and finally return failure if the result is 0. This procedure rejects all forgeries with the same sequence of bit operations; there is no visible distinction between decoding failures and authenticator failures.

Finding a weight-t vector e given $w = Ke$ is the problem of syndrome decoding for K. We follow one of the standard approaches to syndrome decoding: first compute *some* vector $v \in \mathbb{F}_2^n$ such that $w = Kv$, and then find a codeword at distance t from v; this codeword must be $v - e$, revealing e. We use a particularly simple choice of v, taking advantage of K having systematic form: namely, v is w followed by $n - mt$ zeros. (This choice was recommended to us by Nicolas Sendrier; we do not know where it was first used in code-based cryptography.) This choice means that the receiver does not need to store K. We also point out that some of the conditional swaps in Section 4.4 are guaranteed to take $0, 0$ as input and can therefore be skipped.

There are two standard methods to find a codeword at distance t from v: Berlekamp's method [Ber68] and Patterson's method [Pat75]. To apply Berlekamp's method one first observes that $\Gamma = \Gamma_2(\alpha_1, \ldots, \alpha_n, g^2)$, and then that Γ is the $\mathbb{F}_2$-subfield subcode of the generalized Reed–Solomon code $\Gamma_q(\alpha_1, \ldots, \alpha_n, g^2)$. Berlekamp's method decodes generalized Reed–Solomon codes by computing a syndrome

(Section 4.3), then using the Berlekamp–Massey algorithm to compute an error-locator polynomial, then computing the roots of the error-locator polynomial (Section 4.2).

Many authors have stated that Patterson's method is somewhat faster than Berlekamp's method. Patterson's method has some extra steps, such as computing a square root modulo g, but has the advantage of using g instead of g^2, reducing some computations to half size. On the other hand, Berlekamp's method has several advantages. First, as mentioned at the beginning of this Chapter, combinatorial list-decoding algorithms decode more errors, adding security for the same key size, by guessing a few error positions; in this case most decoding attempts fail (as in Section 4.6), and the analysis in [LS12b] suggests that this makes Berlekamp's method faster than Patterson's method. Second, Berlekamp's method generalizes to *algebraic* list-decoding algorithms more easily than Patterson's method; see, e.g., [Ber11]. Third, Berlekamp's method is of interest in a wider range of applications. Fourth, Berlekamp's method saves code size. Finally, Berlekamp's method is easier to protect against timing attacks.

4.6 New speed records for CFS signatures

CFS is a code-based public-key signature system proposed by Courtois, Finiasz, and Sendrier in [CFS01]. The main drawbacks of CFS signatures are large public-key sizes and inefficient signing; the main advantages are short signatures, fast verification, and post-quantum security. This section summarizes the CFS signature system and reports our CFS speeds.

4.6.1 Review of CFS

System parameters are m, q, n, t, k as in Section 4.5, with two extra requirements: $n = q$, and g is irreducible. Key generation works as in the encryption scheme described in Section 4.5.

The basic idea of signing is simple. To sign a message M, first hash this message to a syndrome. If this syndrome belongs to a word at distance $\leq t$ from a codeword, use the secret decoding algorithm to obtain the error positions and send those positions as the signature. The verifier simply adds the columns of the public-key matrix indexed by these positions and checks whether the result is equal to the hash of M.

Unfortunately, a random syndrome has very low chance of being the syndrome of a word at distance $\leq t$ from a codeword. CFS addresses this problem using combinatorial list decoding: guess δ error positions and then proceed with decoding. If decoding fails, guess a different set of δ error positions. Finding a decodable syndrome requires many guesses; as shown in [CFS01] the average number of decoding attempts is very close to $t!$. The decoding attempts for different guesses are independent; we can thus make efficient use of bitslicing in a *single* signature computation.

We actually use parallel CFS, a modification of CFS proposed by Finiasz in [Fin11]. The idea is to compute λ different hashes of the message M and compute a CFS signature for each of these hashes. This increases the security level of CFS against a 2004 Bleichenbacher attack; see generally [OS09] and [Fin11].

4.6.2 Previous CFS speeds

Landais and Sendrier in [LS12b] describe a software implementation of parallel CFS with various parameters that target the 80-bit security level. Their best performance is for parameters $m = 20$, $t = 8$, $\delta = 2$ and $\lambda = 3$. With these parameters they compute a signature in 1.32 seconds on average on an Intel Xeon W3670 (Westmere microarchitecture) running at 3.2GHz, i.e., $4.2 \cdot 10^9$ cycles per signature on average.

4.6.3 New CFS software

Our CFS software uses the same set of parameters. For most of the computation we also use the same high-level algorithms as the software described in [LS12b]: in particular, we use the Berlekamp–Massey algorithm to compute the error-locator polynomial f, and we test whether this polynomial splits into linear factors by checking whether $x^{2^m} \equiv x \pmod{f}$.

The most important difference in our implementation is the bitsliced field arithmetic. This has two advantages: it is faster and it does not leak timing information. Some parts of the computation are performed on only one stream of data (since we sign one message at a time), but even in those parts we continue using constant-time field arithmetic rather than the lookup-table-based arithmetic used in [LS12b].

We do not insist on the entire signing procedure taking constant time, but we do guarantee that the signing time (and all lower-level timing information) is independent of all secret data. Specifically, to guarantee that an attacker has no information about the guessed error positions that did not allow successful decoding, we choose $\delta = 2$ *random* elements of $\mathbb{F}_{2^m}$ and compute the corresponding public-key columns, rather than running through guesses in a predictable order. These columns are at *some* positions in the public key; we compute these positions (in constant time) if decoding is successful.

There are three main bottlenecks in generating a signature:

- pick $e_1, e_2 \in \mathbb{F}_{2^m}$ at random and compute the corresponding public-key columns;
- use Berlekamp–Massey to obtain an error-locator polynomial f;
- test whether $x^{2^m} \equiv x \pmod{f}$.

Once such a polynomial f has been found, we multiply it by $(x - e_1)(x - e_2)$ to obtain a degree-10 error-locator polynomial. We then find all roots of this polynomial and output the set of corresponding support positions as the signature. We split the root-finding problem into 256 separate 2^{12}-point evaluation problems, again allowing fast constant-time bitsliced arithmetic for a *single* signature.

4.6.4 New CFS speeds

Our software signs in than $0.425 \cdot 10^9$ Ivy Bridge cycles on average; the median is $0.391 \cdot 10^9$ Ivy Bridge cycles. This cycle count is an order of magnitude smaller than the cycle count in [LS12b]. We measured this performance across 100000 signature computations on random 59-byte messages on one core of an otherwise idle Intel Core i5-3210M with Turbo Boost and hyperthreading disabled.

It is common to filter out variations in cycle counts by reporting the median cycle count for many computations. Note, however, that the average is noticeably higher than the median for this type of random process. Similar comments apply to, e.g., RSA key generation.

Most of the $0.425 \cdot 10^9$ cycles are used by the three steps described above:

- picking e_1 and e_2 and computing the corresponding columns takes 52792 cycles for a batch of 256 iterations;
- the Berlekamp–Massey step takes 189900 cycles for a batch of 256 iterations;
- testing whether $x^{2^m} \equiv x \pmod{f}$ takes 436008 cycles for a batch of 256 iterations.

These computations account for $(52792+189900+436008)(t!\lambda+128)/256 \approx 0.32 \cdot 10^9$ cycles on average. Root-finding, repeated λ times, accounts for another $0.05 \cdot 10^9$ cycles. A small number of additional cycles are consumed by hashing, converting to bitsliced form, multiplying the degree-8 error-locator polynomial f by $(x-e_1)(x-e_2)$, et al.

We also have extremely fast software for signature verification, taking only 2176 cycles. This count is obtained as the median of 1000 signature verifications for 59-byte messages. Furthermore we have software for Intel and AMD processors that do not feature the AVX instruction set and that instead uses SSE instructions on 128-bit vectors. This software generates a signature in $0.658 \cdot 10^9$ cycles on average and verifies a signature in only 2790 cycles on one core of an Intel Core 2 Quad Q6600 CPU.

5 QcBits: constant-time small-key code-based cryptography

In 2012, Misoczki, Tillich, Sendrier, and Barreto proposed to use QC-MDPC code for code-based cryptography [MTS$^+$13]. The main benefit of using QC-MDPC codes is that they allow small key-sizes, as opposed to using binary Goppa codes as proposed in the original McEliece paper [McE78]. Since then, implementation papers for various platforms have been published; see [HMG13; MG14a] (for FPGA and AVR), [MG14b; MHG16] (for Cortex-M4), and [MOG15] (for Haswell, includes results from [HMG13; MG14a; MG14b]).

One problem of QC-MDPC code is that the most widely used decoding algorithm, when implemented naively, leaks information about secrets through timing. Even though decoding is only used for decryption, the same problem can also rise if the key-generation and encryption are not constant-time. Unfortunately, the only previous paper that addresses the timing-attack issue is [MG14b]. [MG14b] offers constant-time encryption and decryption on a platform without caches (for writable-memory).

This chapter presents `QcBits` (pronounced "quick-bits"), a fully constant-time implementation of a QC-MDPC-code-based encryption scheme. `QcBits` provides constant-time key-pair generation, encryption, and decryption for a wide variety of platforms, including platforms with caches. The scheme implemented in `QcBits` is the Niederreiter KEM/DEM encryption scheme proposed in [Per12; Per13]. As a property of the KEM/DEM encryption scheme, the software is protected against adaptive chosen ciphertext attacks, as opposed to the plain McEliece or Niederreiter [Nie86] encryption scheme. The code is written in C, which makes it easy to understand and verify. Moreover, `QcBits` outperforms the performance results achieved by all previous implementation papers; see below. `QcBits` will soon be in the public domain.

The reader should be aware that `QcBits`, for now, uses a 2^{80}-security param-

platform	key-pair	encrypt	decrypt	reference	implementation	scheme
Haswell	784 192	82 732	1 560 072	**(new)** `QcBits`	`clmul`	KEM/DEM
	20 339 160	225 948	2 425 516	**(new)** `QcBits`	`ref`	KEM/DEM
	*14 234 347	*34 123	*3 104 624	[MOG15]		McEliece
Sandy Bridge	2 497 276	151 204	2 479 616	**(new)** `QcBits`	`clmul`	KEM/DEM
	44 180 028	307 064	3 137 088	**(new)** `QcBits`	`ref`	KEM/DEM
Cortex-A8	61 544 763	1 696 011	16 169 673	**(new)** `QcBits`	`ref`	KEM/DEM
Cortex-M4	140 372 822	2 244 489	14 679 937	**(new)** `QcBits`	`no-cache`	KEM/DEM
	*63 185 108	*2 623 432	*18 416 012	[MHG16]		KEM/DEM
	*148 576 008	7 018 493	42 129 589	[MG14b]		McEliece

Table 5.1: Performance results for `QcBits`, [MHG16], [MG14b], and the vectorized implementation in [MOG15]. The "key-pair" column shows cycle counts for generating a key pair. The "encrypt" column shows cycle counts for encryption. The "decrypt" column shows cycle counts for decryption. For performance numbers of `Qcbits`, 59-byte plaintexts are used to follow the eBACs [BL13] convention. For [MHG16] 32-byte plaintexts are used. Cycle counts labeled with * means that the implementation for the operation is not constant-time on the platform, which means that the worst-case performance can be much worse (especially for decryption). Note that all the results are for 2^{80} security.

eter set from [MTS+13]. Also, the key space used is smaller than the one described in [MTS+13]. However, this is also true for all previous implementation papers [HMG13; MG14a; MG14b; MHG16; MOG15]. These design choices are made to reach a low decoding failure rate; see Section 5.1.1 and 5.5 for more discussions.

Performance results. The performance results of `QcBits` are summarized in Table 8.1, along with the results for [MHG16], [MG14b], and the vectorized implementation in [MOG15]. In particular, the table shows performance results of the implementations contained in `Qcbits` for different settings. The implementation "`ref`" serves as the reference implementation, which can be run on all reasonable 64/32-bit platforms. The implementation "`clmul`" is a specialized implementation that relies on the `PCLMULQDQ` instruction, i.e., the $64 \times 64 \rightarrow 128$-bit carry-less multiplication instruction. The implementation "`no-cache`" is similar to `ref` except that it does not provide full protection against cache-timing attacks. Both "`ref`" and "`clmul`" are constant-time, even on platforms with caches. "`no-cache`" is constant-time only on platforms that do not have cache for *writable* memory.

Regarding previous works, both the implementations in [MOG15] for Haswell and [MHG16] for Cortex-M4 are not constant-time. [MG14b] seems to provide constant-time encryption and decryption, even though the paper argues about resistance against simple-power analysis instead of being constant-time.

On the Haswell microarchitecture, `QcBits` is about twice as fast as [MOG15] for decryption and an order of magnitude faster for key-pair generation, even though the implementation of [MOG15] is not constant-time. `QcBits` takes much more cycles on encryption. This is mainly because `QcBits` uses a slow source of randomness; see 5.2.1 for more discussions. A minor reason is that KEM/DEM decryption requires intrinsically some more operations than McEliece decryption, e.g., hashing.

platform	key-pair	encrypt	decrypt	reference	implementation	scheme
Haswell	5 824 028	196 836	1 363 948	(new) QcBits	clmul	KEM/DEM
	*54 379 733	*106 871	*18 825 103	[MOG15]		McEliece
Cortex-M4	750 584 383	6 353 732	7 436 655	(new) QcBits	no-cache	KEM/DEM
	*251 288 544	*13 725 688	*80 260 696	[MHG16]		KEM/DEM

Table 5.2: Performance results for QcBits, [MHG16], and the vectorized implementation in [MOG15] for 128-bit security. The cycle counts for QcBits decryption are underlined to indicate that these are cycle counts for one decoding iteration.

For tests on Cortex-M4, STM32F407 is used for QcBits and [MG14b], while STM32F417 is used for [MHG16]. Note that there is no cache for writable memory (SRAM) on these devices. QcBits is faster than [MHG16] for encryption and decryption. The difference is even bigger when compared to [MG14b]. [MHG16] reports a flash usage of 16 124 bytes, while the implementation no-cache uses 62 kilo-bytes of flash. Note that [MHG16] uses an AES crypto co-processor on STM32F417 to facilitate symmetric encryption and authentication, which presumably results in less flash memory consumption than using a self-implemented AES. In contrast, QcBits does not use AES for symmetric operations so it cannot save flash memory in the same way. However, this also means QcBits is more portable than the implementation in [MHG16]. Regarding SRAM usage, the numbers reported in [MHG16] do not seem to include run-time stack-memory usage, and I am not aware of a good way to measure it.

It is important to note that, since the decoding algorithm is probabilistic, each implementation of decryption comes with a failure rate. For QcBits no decryption failure occured in 10^8 trials. It is unclear how many decoding iterations are required to achieve the same level of failure rate at a 2^{128} security level, which is why Table 8.1 only includes results for 80-bit security. For [MOG15], no decryption failure occured in 10^7 trials. For [MHG16] the failure rate is not indicated, but the decoder seems to be the same as [MOG15]. It is unclear what level of failure rate [MG14b] achieves. See section 5.5 for more discussions about failure rates.

Table 5.2 shows performance results for 128-bit security. Note that [MOG15] and [MHG16] did not specify the failure rates they achieved for 128-bit security.

QcBits versus McBits. In 2013, Bernstein, Chou, and Schwabe introduced McBits (cf. [BCS13]), a constant-time implementation for the KEM/DEM encryption scheme using binary Goppa code. At a 2^{128} security level, the software takes only 60493 Ivy Bridge cycles to decrypt. It might seem that QcBits is far slower than McBits. However, the reader should keep in mind that McBits relies on external parallelism to achieve such speed: the cycle count is the results of running 256 decryption instances in parallel and then divided by 256. The speed of QcBits relies only on internal parallelism: the timings presented in Table 8.1 are all results of running only one instance. See Section 5.4.3 for more details on how QcBits makes use of bitslicing.

It is also worth noticing that using QC-MDPC code allows much smaller key sizes. [MTS+13] reports a public-key size of 601 bytes for a 80-bit security parameter set and 1233 bytes for 128-bit security parameter set, while [BCS13] reports 74624 bytes for a 80-bit security parameter set and 221646 bytes for a 128-bit security parameter set.

Binary Goppa code and QC-MDPC code are also quite different in their history in cryptography. The usage of binary Goppa code was proposed by McEliece in [McE78] in 1978, along with the McEliece cryptosystem. After almost 40 years, nothing has really changed the practical security of the system. QC-MPDC-code-based cryptosystems, however, are still quite young and thus require some time to gain confidence from the public.

5.1 Preliminaries

This section presents preliminaries for the following sections. Section 5.1.1 gives a brief review on the definition of QC-MDPC codes. Section 5.1.2 describes the "bit-flipping" algorithm for decoding QC-MDPC codes. Section 5.1.3 gives a specification of the KEM/DEM encryption scheme implemented by `QcBits`.

5.1.1 QC-MDPC codes

"MDPC" stands for "moderate-density-parity-check". As the name implies, an MDPC code is a linear code with "moderate" number of non-zero entries in a parity-check matrix H. For ease of discussion, in this chapter it is assumed $H \in \mathbb{F}_2^{n \times N}$ where $N = 2n$, even though some parameter sets in [MTS⁺13] use $N = 3n$ or $N = 4n$. H is viewed as the concatenation of two square matrices, i.e., $H = H^{(0)}|H^{(1)}$, where $H^{(i)} \in \mathbb{F}_2^{n \times n}$.

"QC" stands for "quasi-cyclic". Being quasi-cyclic means that each $H^{(i)}$ is "cyclic". For ease of discussion, one can think this means

$$H^{(k)}_{(i+1) \bmod n, (j+1) \bmod n} = H^{(k)}_{i,j},$$

even though the original paper allows a row permutation on H. Note that being quasi-cyclic implies that H has a fixed row weight w. The following is a small parity-check matrix with $n = 5, w = 4$:

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

Usually a QC-MDPC code is said to be able to correct t errors. Since there is no good way to figure out the minimum distance for a given QC-MDPC code, t is usually merely an estimated value.

`Qcbits` uses $n = 4801$, $w = 90$, and $t = 84$ to match a 2^{80}-security parameter set proposed in [MTS⁺13]. However, `Qcbits` further requires that $H^{(0)}$ and $H^{(1)}$ to have the same row weight, namely $w/2$. This is not new, however, as all the previous implementation papers [HMG13; MG14a; MG14b; MHG16; MOG15] also restrict H

in the same way. For QcBits this is a decision for achieving low failure rate; see Section 5.5 for more discussions on this issue. Previous implementation papers did not explain why they restrict H in this way.

5.1.2 Decoding (QC-)MDPC codes

As opposed to many other linear codes that allows efficient decoding, the most popular decoder for (QC-)MDPC code, the "bit-flipping" algorithm, is a probabilistic one. The bit-flipping algorithm shares the same idea with so-called "statistical decoding" [Ove06; Jab01]. (The term "statistical decoding" historically comes later than "bit-flipping", but "statistical decoding" captures way better the idea behind the algorithm.)

Given a vector that is at most t errors away from a codeword, the algorithm aims to output the codeword (or equivalently, the error vector) in a sequence of iterations. Each iteration decides statistically which of the N positions of the input vector v might have higher chance to be in error and flips the bits at those positions. The flipped vector then becomes the input to the next iteration. In the simplest form of the algorithm, the algorithm terminates when Hv becomes zero.

The presumed chance of each position being in error is indicated by the count of unsatisfied parity checks. The higher the count is, the higher presumed chance a position is in error. In other words, the chance of position i being in error is indicated by

$$u_i = |\{i \mid H_{i,j} = (Hv)_i = 1\}|.$$

In this chapter the syndrome Hv will be called the *private syndrome*.

Now the remaining problem is, which bits should be flipped given the vector u? In [MTS$^+$13] two possibilities are given:

- Flip all positions that violate at least $max(\{u_i\}) - \delta$ parity checks, where δ is a small integer, say 5.
- Flip all positions that violate at least T_i parity checks, where T_i is a precomputed threshold for iteration i.

In previous works several variants have been invented. For example, one variant based on the first approach simply restarts decoding with a new δ if decoding fails in 10 iterations.

QcBits uses precomputed thresholds. The number decoding iterations is set to be 6, and the thresholds are

$$29, 27, 25, 24, 23, 23.$$

These thresholds are obtained by interactive experiments. I do not claim that these are the best thresholds. With the list of thresholds, no iteration failure occurs in 10^8 decoding trials. See Section 5.6 for more details about the trials.

The best results in previous implementation papers [HMG13; MG14a; MG14b; MHG16; MOG15] are achieved by a variant of the precomputed-threshold approach. In each iteration of the variant, the u_i's are computed in order. If the current u_i is greater than or equal to the precomputed threshold, v_i is flipped and the syndrome is directly updated by adding the i-th column of H to the syndrome. With this variant, [MOG15] reports that the average number of iterations is only 2.4.

`QcBits` always takes 6 decoding iterations, which is much more than 2.4. However, the algorithms presented in the following sections allow `QcBits` to run each iteration very quickly, albeit being constant-time. As the result, `Qubits` still achieves much better performance results in decryption.

5.1.3 The Niederreiter KEM/DEM encryption system for QC-MDPC codes

The KEM/DEM encryption uses the Niederreiter encryption scheme for KEM. Niederreiter encryption is used to encrypt a *random* vector e of weight t, which is then fed into a key-derivation function to obtain the symmetric encryption and authentication key. The ciphertext is then the concatenation of the Niederreiter ciphertext, the symmetric ciphertext, and the authentication tag for the symmetric ciphertext. The decryption works in a similar way as encryption; see for example [BCS13] for a more detailed description. `QcBits` uses Keccak [BDP$^+$13] with 512-bit outputs to hash e, and the symmetric encryption and authentication key are defined to be the first and second half of the hash value. For symmetric encryption and authentication, `QcBits` uses Salsa20 [Ber08b] with nonce 0 and Poly1305 [Ber05]. Note that `QcBits` does not include code for Keccak, Salsa20, and Poly1305. The user can choose their own implementations for the schemes.

The secret key is a representation of a random parity-check matrix H. Since the first row H gives enough information to form the whole matrix, it suffices to represent H using an array of indices in $\{j \mid H^{(0)}_{0,j} = 1\}$ and an array of indices in $\{j \mid H^{(1)}_{0,j} = 1\}$. In each array the indices should not repeat, but they are *not* required to be sorted. `QcBits` represents each array as a byte stream of length w, where the i-th double byte is the little-endian representation of the i-th index in the array. The secret key is then defined as the concatenation of the two byte streams.

The public key is a representation of the row reduced echelon form of H. The row reduced matrix is denoted as P. Niederreiter requires $P^{(0)}$ to be an identity matrix, or the key pair must be rejected. In other words, $P^{(1)}$ contains all information of P (if P is valid). Note that P is also quasi-cyclic; `QcBits` thus defines the public key as a byte stream of length $\lceil (n+7)/8 \rceil$, where the byte values are

$$(P^{(1)}_{7,0} P^{(1)}_{6,0} \dots P^{(1)}_{0,0})_2,\ (P^{(1)}_{15,0} P^{(1)}_{14,0} \dots P^{(1)}_{8,0})_2,\ \dots$$

The encryption process begins with generating a random vector e of weight w. The ciphertext for e is then the public syndrome $s = He$, which is represented as a byte stream of length $\lceil (n+7)/8 \rceil$, where the byte values are

$$(s_7 s_6 \dots s_0)_2,\ (s_{15} s_{14} \dots s_8)_2,\ \dots.$$

For hashing, e is represented as a byte stream of length $\lceil (N+7)/8 \rceil$ in the same way as the public syndrome. The 32-byte symmetric encryption key and the 32-byte authentication key are then generated as the first and second half of the 64-byte hash value of the byte stream. The plaintext m is encrypted and authenticated using the symmetric keys. The ciphertext for the whole KEM/DEM scheme is then the concatenation of the public syndrome, the ciphertext under symmetric encryption, and the tag. In total the ciphertext takes $\lceil (n+7)/8 \rceil + |m| + 16$ bytes.

When receiving an input stream, the decryption process parses it as the concatenation of a public syndrome, a ciphertext under symmetric encryption, and a tag. Then an error vector e' is computed by feeding the public syndrome into the decoding algorithm. Note that `QcBits` does not generate a decoding failure. Whatever is produced after the fixed number of iterations will be used as the error vector. The symmetric keys is then generated by hashing e' to perform symmetric decryption and verification. `QcBits` reports a decryption failure if and only if the verification fails.

5.2 Key-pair generation

This section shows how `QcBits` performs key-pair generation using multiplications in $\mathbb{F}_2[x]/(x^n - 1)$. Section 5.2.1 shows how the private key is generated. Section 5.2.2 shows how key-pair generation is viewed as multiplications in $\mathbb{F}_2[x]/(x^n - 1)$. Section 5.2.3 shows how multiplications in $\mathbb{F}_2[x]/(x^n - 1)$ are implemented. Section 5.2.4 shows how squarings in $\mathbb{F}_2[x]/(x^n - 1)$ are implemented.

5.2.1 Private-key generation

The private-key is defined to be an array of w random 16-bit indices. `QcBits` obtains random bytes by first reading 32 bytes from a source of randomness and then expands the 32 bytes into the required length using `salsa20`. `QcBits` allows the user to choose any source of randomness. To generate the performance numbers on Ivy Bridge, Sandy Bridge, and Cortex-A8 in Table 8.1, `/dev/urandom` is used as the source of randomness. To generate the performance numbers on Cortex-M4 in Table 8.1, the TRNG on the board is used as in [MHG16]. The `RDRAND` instruction used by [MOG15] is not considered for there are security concerns about the instruction; see the Wikipedia page of `RDRAND` [Wik16b].

5.2.2 Polynomial view: public-key generation

For any matrix M, let $M_{i,:}$ denote the vector $(M_{i,0}, M_{i,1}, \dots)$ and similarly for $M_{:,i}$. In section 5.1, the public key is defined as a sequence of bytes representing $P^{(1)}_{:,0}$, where P is the row reduced echelon form of the parity-check matrix H. A simple way to implement constant-time public-key generation is thus to generate H from the private key and then perform a Gaussian elimination. It is not hard to make Gausssian elimination constant-time; see for example, [BCS13]. However, public-key generation can be made much more time- and memory-efficient when considered as polynomial operations.

For any vector v of length n, let $v(x) = v_0 + v_1 x + \cdots + v_{n-1}$. As a result of $H^{(0)}$ being cyclic, we have

$$H^{(0)}_{i,:}(x) = x^i H^{(0)}_{0,:}(x) \in \mathbb{F}_2[x]/(x^n - 1).$$

The Gaussian elimination induces a linear combination of the rows of $H^{(0)}$ that results in $P^{(0)}_{0,:}$. In other words, there exists a set I of indices such that

$$\begin{aligned} 1 &= \sum_{i \in I} x^i H^{(0)}_{i,:}(x) = (\sum_{i \in I} x^i) H^{(0)}_{i,:}(x), \\ P^{(1)}_{0,:}(x) &= \sum_{i \in I} x^i H^{(1)}_{i,:}(x) = (\sum_{i \in I} x^i) H^{(1)}_{i,:}(x). \end{aligned}$$

In other words, the public key can be generated by finding the inverse of $H^{(0)}_{0,:}(x)$ in $\mathbb{F}_2[x]/(x^n - 1)$ and then multiplying the inverse by $H^{(1)}_{0,:}(x)$. The previous implementation papers [HMG13; MG14a; MG14b; MHG16; MOG15] compute the inverse using the extended Euclidean algorithm. Unfortunately, the algorithm in its original form is highly non-constant-time, and it is unclear how to make it both constant-time and efficient.

In order to be constant-time, `QcBits` computes the inverse by carrying out a fixed sequence of polynomial multiplications. To see this, first consider the factorization of $x^n - 1 \in \mathbb{F}_2[x]$ as $\prod_i \left(f^{(i)}(x)\right)^{p_i}$, where each $f^{(i)}$ is irreducible. $\mathbb{F}_2[x]/(x^n - 1)$ is then equivalent to

$$\prod_i \mathbb{F}_2[x]/\left(f^{(i)}(x)\right)^{p_i}.$$

Since

$$\begin{aligned} \left|\mathbb{F}_2[x]/\left(f^{(i)}(x)\right)^{p_i}\right| &= 2^{\deg(f^{(i)})\cdot p_i} \cdot (2^{\deg(f^{(i)})} - 1)/2^{\deg(f^{(i)})} \\ &= 2^{\deg(f^{(i)})\cdot p_i} - 2^{\deg(f^{(i)})\cdot(p_i - 1)}, \end{aligned}$$

one may compute the inverse of an element in $\mathbb{F}_2[x]/\left(f^{(i)}(x)\right)^{p_i}$ by raising it to power

$$\mathrm{lcm}(2^{\deg(f^{(1)})\cdot p_1} - 2^{\deg(f^{(1)})\cdot(p_1-1)}, 2^{\deg(f^{(2)})\cdot(p_2-1)} - 2^{\deg(f^{(2)})\cdot(p_2-1)}, \ldots) - 1.$$

`QcBits` uses $n = 4801$. The polynomial $x^{4801} - 1$ can be factored into

$$(x+1)f^{(1)}f^{(2)}f^{(3)}f^{(4)} \in \mathbb{F}_2[x],$$

where each $f^{(i)}$ is an irreducible polynomial of degree 1200. Therefore, `QcBits` computes the inverse of a polynomial modulo $x^{4801} - 1$ by raise it to power $\mathrm{lcm}(2 - 1, 2^{1200} - 1) - 1 = 2^{1200} - 2$.

Raising an element in $\mathbb{F}_2[x]/(x^{4801} - 1)$ to power $2^{1200} - 2$ can be carried out by a sequence of squarings and multiplications. The most naive way is to use the square-and-multiply algorithm, which leads to 1199 squarings and 1198 multiplications. `QcBits` does better by finding a good addition chain for $2^{1200} - 2$. First note that there is a systematic way to find a good addition chain for integers of the for $2^k - 1$. Take $2^{11} - 1$ for example, the chain would be

$$\begin{aligned} 1 \to 10_2 \to 11_2 \to 1100_2 \to 1111_2 \to 11110000_2 \to 11111111_2 \to 1111111100_2 \\ \to 1111111111_2 \to 11111111110_2 \to 11111111111_2. \end{aligned}$$

This takes 10 doublings and 5 additions. Using the same approach, it is easy to find an addition chain for $2^{109}-1$ that takes 108 doublings and 10 additions. QcBits then combines the addition chains for $2^{11}-1$ and $2^{109}-1$ to form an addition chain for $2^{11\cdot 109}-1 = 2^{1199}-1$, which takes $10\cdot 109+108 = 1198$ doubling and $5+10 = 15$ additions. Once the $(2^{1199}-1)$-th power is computed, the $(2^{1200}-2)$-th power can be computed using one squaring. In total, computation of the $(2^{1200}-2)$-th power takes 1199 squarings and 15 multiplications in $\mathbb{F}_2[x]/(x^{4801}-1)$.

Finally, with the inverse, $P^{(1)}_{0,:}(x)$ can be computed using one multiplication. The public key is defined to be a representation of $P^{(1)}_{:,0}$ instead of $P^{(1)}_{0,:}$. Qcbits thus derives $P^{(1)}_{0,:}$ from $P^{(1)}_{:,0}$ by noticing

$$P^{(1)}_{0,j} = \begin{cases} P^{(1)}_{0,n-j} & \text{if } j > 0 \\ P^{(1)}_{0,0} & \text{if } j = 0. \end{cases}$$

Note that the conversion from $P^{(1)}_{:,0}$ to $P^{(1)}_{0,:}$ does not need to be constant-time because it can be easily reversed from public data.

5.2.3 Generic multiplication in $\mathbb{F}_2[x]/(x^n-1)$

The task here is to compute $h = fg$, where $f, g \in \mathbb{F}_2[x]/(x^n-1)$. In QcBits, the polynomials are represented using an array of $\lceil n/b \rceil$ b-bit words in the natural way. Take f for example (the same applies to g and h), the b-bit values are:

$$(f_{b-1}f_{b-2}\cdots f_0)_2, (f_{2b-1}f_{2b-2}\cdots f_b)_2, \ldots.$$

The user can choose b to be 32 or 64, but for the best performance b should be chosen according to the machine architecture. Let $y = x^b$. One can view this representation as storing each coefficient of the radix-y representation of f using one b-bit integer. In this chapter this representation is called the “dense representation”.

Using the representation, we can compute the coefficients (each being a $2b$-bit value) of the radix-y representation of h, using carry-less multiplications on the b-bit words of f and g. Once the $2b$-bit values are obtained, the dense representation of h can be computed with a bit of post-processing. To be precise, given two b-bit number $(\alpha_{b-1}\alpha_{b-2}\cdots\alpha_0)_2$ and $(\beta_{b-1}\beta_{b-2}\cdots\beta_0)_2$, a carry-less multiplication computes the $2b$-bit value (actually only $2b-1$)

$$(\bigoplus_{i+j=2b-2}\alpha_i\beta_j \bigoplus_{i+j=2b-3}\alpha_i\beta_j \cdots \bigoplus_{i+j=0}\alpha_i\beta_j)_2.$$

In other words, the input values are considered as elements in $\mathbb{F}_2[x]$, and the output is the product in $\mathbb{F}_2[x]$.

The implementation clmul uses the PCLMULQDQ instruction to perform carry-less multiplications between two 64-bit values. For the implementation ref and no-cache, the following C code is used to compute the higher and lower b bits of the $2b$-bit value:

```
low = x * ((y >> 0) & 1);
v1 = x * ((y >> 1) & 1);
low  ^= v1 << 1;
high = v1 >> (b-1);
for (i = 2; i < b; i+=2)
{
        v0 = x * ((y >> i) & 1);
        v1 = x * ((y >> (i+1)) & 1);
        low  ^= v0 << i;
        low  ^= v1 << (i+1);
        high ^= v0 >> (b-i);
        high ^= v1 >> (b-(i+1));
}
```

5.2.4 Generic squaring in $\mathbb{F}_2[x]/(x^n - 1)$

Squarings in $\mathbb{F}_2[x]/(x^n - 1)$ can be carried out as multiplications. However, obviously squaring is a much cheaper operation as only $\lceil n/b \rceil$ carry-less multiplications (actually squarings) are required.

The implementation `clmul` again uses the `PCLMULQDQ` instruction to perform carry-less squarings of 64-bit polynomials. Following the section for interleaving bits presented in the "Bit Twiddling Hacks" by Sean Eron Anderson [And05], the implementation `ref` and `no-cache` use the following C code twice to compute the square of a 32-bit polynomial represented as 32-bit word:

```
x = (x | (x << 16)) & 0x0000FFFF0000FFFF;
x = (x | (x <<  8)) & 0x00FF00FF00FF00FF;
x = (x | (x <<  4)) & 0x0F0F0F0F0F0F0F0F;
x = (x | (x <<  2)) & 0x3333333333333333;
x = (x | (x <<  1)) & 0x5555555555555555;
```

By using the code twice we can also compute the square of a 64-bit polynomial.

5.3 KEM encryption

This section shows how `QcBits` performs the KEM encryption using multiplications in $\mathbb{F}_2[x]/(x^n - 1)$. Section 5.3.1 shows how the error vector is generated. Section 5.3.2 shows how public-syndrome computation is viewed as multiplications in $\mathbb{F}_2[x]/(x^n - 1)$. Section 5.3.3 shows how these multiplications are implemented.

5.3.1 Generating the error vector

The error vector e is generated in essentially the same way as the private key. The only difference is that for e we need t indices ranging from 0 to $N - 1$, and there is only one list of indices instead of two. Note that for hashing it is still required to generate the dense representation of e.

5.3.2 Polynomial view: public-syndrome computation

The task here is to compute the public syndrome Pe. Let $e^{(0)}$ and $e^{(1)}$ be the first and second half of e. The public syndrome is then

$$\begin{aligned} s &= P^{(0)}e^{(0)} + P^{(1)}e^{(1)} \\ &= \sum_i P^{(0)}_{:,i}e^{(0)}_i + \sum_i P^{(1)}_{:,i}e^{(1)}_i. \end{aligned}$$

Since P is quasi-cyclic, we have

$$\begin{aligned} s(x) &= \sum_i x^i P^{(0)}_{:,0}(x)e^{(0)}_i + \sum_i x^i P^{(1)}_{:,0}(x)e^{(1)}_i \\ &= P^{(0)}_{:,0}(x)e^{(0)}(x) + P^{(1)}_{:,0}(x)e^{(1)}(x) \\ &= e^{(0)}(x) + P^{(1)}_{:,0}(x)e^{(1)}(x). \end{aligned}$$

In other words, the private syndrome can be computed using one multiplications in $\mathbb{F}_2[x]/(x^n - 1)$. The multiplication is not generic in the sense that $e^{(1)}(x)$ is sparse. See below for how the multiplication is implemented in `QcBits`.

5.3.3 Sparse-times-dense multiplications in $\mathbb{F}_2[x]/(x^n - 1)$

The task here can be formalized as computing $f^{(0)} + f^{(1)}g^{(1)} \in \mathbb{F}_2[x]/(x^n - 1)$, where $g^{(1)}$ is represented in the dense representation. $f^{(0)}$ and $f^{(1)}$ are represented together using an array of indices in $I = \{i \mid f^{(0)}_i = 1\} \cup \{i + n \mid f^{(1)}_i = 1\}$, where $|I| = t$.

One can of course perform this multiplication between $f^{(1)}$ and $g^{(1)}$ in a generic way, as shown in Section 5.2.3. The implementation `clmul` indeed generates the dense representation of $f^{(1)}$ and then computes $f^{(1)}g^{(1)}$ using the `PCLMULQDQ` instruction. [MOG15] uses essentially the same technique. The implementations `ref` and `no-cache` however, make use of the sparsity in $f^{(0)}$ and $f^{(1)}$; see below for details.

Now consider a slightly simpler problem of computing $h = fg \in \mathbb{F}_2[x]/(x^n - 1)$, where f is represented as an array of indices in $I = \{i \mid f_i = 1\}$, and g is in the dense representation. Then we have

$$fg = \sum_{i \in I} x^i g.$$

Therefore, the implementation `ref` and `no-cache` first set $h = 0$. Then, for each $i \in I$, $x^i g$ is computed and then added to h. Note that $x^i g$ is represented as an array of $\lceil n/b \rceil$ b-bit words, so adding $x^i g$ to h can be implemented using $\lceil n/b \rceil$ bitwise-`XOR` instructions on b-bit words.

Now the remaining problem is how to compute $x^i g$. It is obvious that $x^i g$ can be obtained by rotating g by i bits. In order to perform a constant-time rotation, the implementation `ref` makes use of the idea of the Barrel shifter [Wik16a]. The idea is to first represent i in binary representation

$$(i_{k-1}i_{k-2}\cdots i_0)_2.$$

Since $i \leq n-1$, it suffices to use $k = \lfloor \lg(n-1) \rfloor + 1$. Then, for j from $k-1$ to $\lg b$, a rotation by 2^j bits is performed. One of the unshifted vector and shifted vector is chosen (in a constant-time way) and serves as the input for the next j. After dealing with all $i_{k-1}, i_{k-2}, \ldots, i_{\lg b}$, a rotation of $(i_{\lg b-1} i_{\lg b-2} \cdots i_0)_2$ bits is performed using a sequence of logical instructions.

To clarify the idea, here is a toy example for the case $n = 40, b = 8$. The polynomial g is

$$(x^8 + x^{10} + x^{12} + x^{14}) + (x^{16} + x^{17} + x^{20} + x^{21}) + (x^{24} + x^{25} + x^{26} + x^{27}) \\ + (x^{36} + x^{37} + x^{38} + x^{39}),$$

which is represented in an array of 5 bytes as

$$00000000_2, 01010101_2, 00110011_2, 00001111_2, 11110000_2.$$

The goal is to compute $x^i g$ where $i = 010011_2$. Since $\lfloor \lg(40-1) \rfloor + 1 = 6$, the algorithm begins with computing a rotation of 100000_2 bits, which can be carried out by moving around the bytes. The result is

$$01010101_2, 00110011_2, 00001111_2, 11110000_2.00000000_2.$$

Since the most significant bit is not set, the unshifted polynomial is chosen. Next we proceed to perform a rotation of 10000_2 bits. The result is

$$00001111_2, 11110000_2, 00000000_2, 01010101_2, 00110011_2.$$

Since the second most significant bit is set, we choose the rotated polynomial. The polynomial is then shifted by 1000_2 bits. However, since the third most significant bit is not set, the unshifted polynomial is chosen. To handle the least significant $\lg b = 3$ bits of i, a sequence of logical instructions are used to combine the most significant 011_2 and the least significant 101_2 bits of the bytes, resulting in

$$01100001_2, 11111110_2, 00000000_2, 00001010_2, 10100110_2.$$

Note that n is required to be a prime (which means n is not divisible b) in [MTS+13], so the example is showing an easier case. Roughly speaking, the implementation `ref` performs a rotation as if the vector length is $n - (n \bmod b)$ and then uses more instructions to compensate for the effect of the $n \bmod b$ extra bits. The implementation `no-cache` essentially performs a rotation of $(i_{k-1} i_{k-2} \cdots i_{\lg b} 0 \cdots 0)_2$ bits and then performs a rotation of $(i_{\lg b-1} i_{\lg b-2} \cdots i_0)_2$ bits.

With the constant-time rotation, we can now deal with the original problem of computing $f^{(0)} + f^{(1)} g^{(1)} \in \mathbb{F}_2[x]/(x^n - 1)$. `QcBits` first sets $h = 0$. Then for each $i \in I$, one of either 1 or $g^{(1)}$ is chosen according to whether $i < n$ or not, which has to be performed in a constant-time way to hide all information about i. The chosen polynomial is then rotated by $i \bmod n$ bits, and the result is added to h. Note that this means the implementations `ref` and `no-cache` perform a dummy polynomial multiplication to hide information about $f^{(0)}$ and $f^{(1)}$.

5.4 KEM decryption

This section shows how QcBits performs the KEM decryption using multiplications in $\mathbb{F}_2[x]/(x^n-1)$ and $\mathbb{Z}[x]/(x^n-1)$. The KEM decryption is essentially a decoding algorithm. Each decoding iteration computes

- the private syndrome Hv and
- the counts of unsatisfied parity checks, i.e., the vector u, using the private syndrome.

Positions in v are then flipped according the counts. Section 5.4.1 shows how private-syndrome computation is implemented as multiplications in $\mathbb{F}_2[x]/(x^n-1)$. Section 5.4.2 shows how the counting unsatisfied parity checks is viewed as multiplications in $\mathbb{Z}[x]/(x^n-1)$. Section 5.4.3 shows how these multiplications in $\mathbb{Z}[x]/(x^n-1)$ are implemented. Section 5.4.4 shows how bit flipping is implemented.

5.4.1 Polynomial view: private-syndrome computation

The public syndrome and the private syndrome are similar in the sense that they are both computed by matrix-vector products where the matrices are quasi-cyclic. For the public syndrome the matrix is P and the vector is e. For the private syndrome the matrix is H and the vector is v. Therefore, the computation of the private syndrome can be viewed as polynomial multiplication in the same way as the public syndrome. That is, the private syndrome can be viewed as

$$H^{(0)}_{:,0}(x)v^{(0)}(x) + H^{(1)}_{:,0}(x)v^{(1)}(x) \ \in \mathbb{F}_2[x]/(x^n-1).$$

The computation of the public syndrome and private syndrome are sill a bit different. For encryption the matrix P is dense, whereas the vector e is sparse. For decryption the matrix H is sparse, whereas the vector v is dense. However, the multiplications $H^{(i)}_{:,0}(x)v^{(i)}(x)$ are still sparse-times-dense multiplications. QcBits thus computes the private syndrome using the techniques described in Section 5.3.3.

Since the secret key is a sparse representation of $H^{(i)}_{0,:}$, we do not immediately have $H^{(i)}_{:,0}$. This is similar to the situation in public-key generation, where $P^{(1)}_{:,0}$ is derived from $P^{(1)}_{0,:}$. QcBits thus computes $H^{(i)}_{:,0}$ from $H^{(i)}_{0,:}$ by adjusting each index in the sparse representation.

5.4.2 Polynomial view: counting unsatisfied parity checks

Let $s = Hv$. The vector u of counts of unsatisfied parity checks can be viewed as

$$u_j = \sum_i H_{i,j} \cdot s_i \in \mathbb{Z}^N,$$

where $H_{i,j}$ and s_j are treated as integers. In other words,

$$u = \sum_i H_{i,:} \cdot s_i \in \mathbb{Z}^N.$$

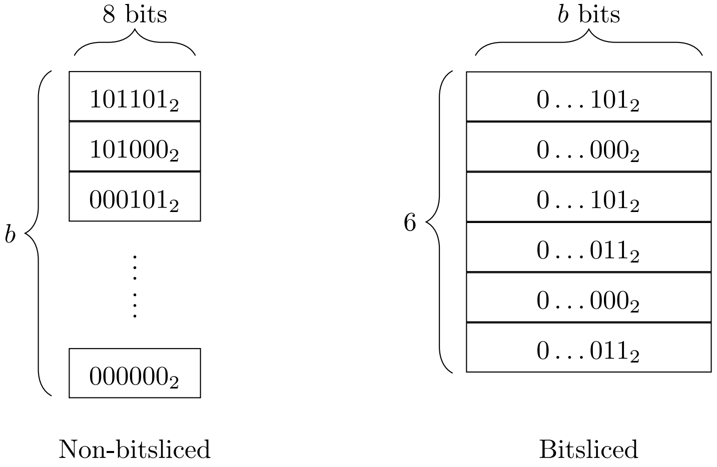

Figure 5.1: Storage of b numbers of unsatisfied parity checks in non-bitsliced form and bitsliced format.

Let $u^{(0)}$ and $u^{(1)}$ be the first and second half of u, respectively. Now we have:

$$\begin{aligned}\left(u^{(0)}(x), u^{(1)}(x)\right) &= \left(\sum_i x^i H^{(0)}_{0,:}(x) \cdot s_i, \sum_i x^i H^{(1)}_{0,:}(x) \cdot s_i\right) \\ &= \left(H^{(0)}_{0,:}(x) \cdot s(x), H^{(1)}_{0,:}(x) \cdot s(x)\right) \in \left(\mathbb{Z}[x]/(x^n-1)\right)^2.\end{aligned}$$

In other words, the vector u can be computed using 2 multiplications in $\mathbb{Z}[x]/(x^n - 1)$. Note that the multiplications are not generic: $H^{(i)}_{0,:}(x)$ is always sparse, and the coefficients of $H^{(i)}_{0,:}(x)$ and $s(x)$ can only be 0 or 1. See below for how such multiplications are implemented in `QcBits`.

5.4.3 Sparse-times-dense multiplications in $\mathbb{Z}[x]/(x^n - 1)$

The task can be formalized as computing $fg \in \mathbb{Z}[x]/(x^n - 1)$, where $f_i, g_i \in \{0, 1\}$ for all i, and f is of weight only w. f is represented as an array of indices in $I_f = \{i \mid f_i = 1\}$. g is naturally represented as an array of $\lceil n/b \rceil$ b-bit values as usual. Then we have

$$fg = \sum_{i \in I_f} x^i g.$$

Even though all the operations are now in $\mathbb{Z}[x]/(x^n - 1)$ instead of $\mathbb{F}_2[x]/(x^n - 1)$, each $x^i g$ can still be computed using a constant-time rotation as in Section 5.3.3. Therefore, `QcBits` first sets $h = 0$, and then for each $i \in I$, $x^i g$ is computed using the constant-time rotation and then added to h. After all the elements in I are processed, we have $h = fg$. Note that $x^i g$ is represented as an array of $\lceil n/b \rceil$ b-bit words.

Now the remaining problem is how to add $x^i g$ to h. A direct way to represent h is to use an array of n bytes (it suffices to use 1 byte for each coefficient when $w/2 < 256$, which is true for all parameter sets in [MTS$^+$13] with $N = 2n$), each storing one of the n coefficients. To add $x^i g$ to h, the naive way is for each coefficient of h to extract from the corresponding b-bit word the bit required using one bitwise-`AND` instruction and at most one shift instruction, and then to add the bit to the byte using one addition instruction. In other words, it takes around 3 instructions on average to update each coefficient of h.

`QcBits` does better by bitslicing the coefficients of h: Instead of using b bytes, `QcBits` uses several b-bits words to store a group of b coefficients, where the i-th b-bit word stores the i-th least signigficant bits of the b coefficients. Since the column weight of H is $w/2$, it suffices to use $\lfloor \lg w/2 \rfloor + 1$ b-bit words. To update b coefficients of h, a sequence of logical operations are performed on the $\lfloor \lg w/2 \rfloor + 1$ b-bit words and the corresponding b-bit word in $x^i g$. These logical instructions simulate b copies of a circuit for adding a 1-bit number into a $(\lfloor \lg w/2 \rfloor + 1)$-bit number. Such a circuit requires roughly $\lfloor \lg w/2 \rfloor + 1$ half adders, so updating b coefficients takes roughly $2(\lfloor \lg w/2 \rfloor + 1)$ logical instructions on b-bit words.

Figure 5.1 illustrates how the b coefficients are stored when $w = 90$. In the non-bitsliced approach b bytes are used. In the bit-sliced approach $\lfloor \lg(90/2) \rfloor + 1 = 6$ b-bit words are used, which account for $6b/8$ bytes. Note that this means bitslicing saves memory. Regarding the number of instructions, it takes $(6 \cdot 2)/b$ logical instructions on average to update each coefficient. For either $b = 32$ or $b = 64$, $(6 \cdot 2)/b$ is much smaller than 3. Therefore, bitslicing also helps to enhance performance.

The speed that McBits [BCS13] achieves relies on bitslicing as well. However, the reader should keep in mind that `QcBits`, as opposed to McBits, makes use of parallelism that lies intrinsically in one single decryption instance.

5.4.4 Flipping bits

The last step in each decoding iteration is to flip the bits according to the counts. Since `QcBits` stores the counts in a bitsliced format, bit flipping is also accomplished in a bitsliced fashion. At the beginning of each decoding iteration, the bitsliced form of b copies of $-t$ is generated and stored in $\lfloor \lg w/2 \rfloor + 1$ b-bit words. Once the counts are computed, $-t$ is added to the counts in parallel using logical instructions on b-bit words. These logical instructions simulate copies of a circuit for adding $(\lfloor \lg w/2 \rfloor + 1)$-bit numbers. Such a circuit takes $(\lfloor \lg w/2 \rfloor + 1)$ full adders. Therefore, each $u_i + (-t)$ takes roughly $5(\lfloor \lg w/2 \rfloor + 1)/b$ logical instructions.

The additions are used to generate sign bits for all $u_i - t$, which are stored in two arrays of $\lceil n/b \rceil$ b-bit words. To flip the bits, `QcBits` simply XORs the complement of b-bit words in the two arrays into $v^{(0)}$ and $v^{(1)}$. It then takes roughly $1/b$ logical instructions to update each v_i.

For $w = 90$, we have $5(\lfloor \lg w/2 \rfloor + 1)/b + 1b = 31/b$, which is smaller than 1 for either $b = 32$ or $b = 64$. In contrast, when the non-bitsliced format is used, the naive approach is to use at least one subtraction instruction for each $u_i - t$ and one XOR instruction to flip the bit. One can argue that for the non-bitsliced format there are probably better ways to compute u and perform bit flipping. For example, one can probably perform several additions/subtractions of bytes in parallel in one instruction. However, such an approach seems much more expensive than one might expect as changes of formats between a sequence of bits and bytes are required.

5.5 The future of QC-MDPC-based cryptosystems

`QcBits` provides a way to perform constant-time QC-MDPC decoding, even on platforms with caches. Moreover, decoding in `QcBits` is much faster than that in previous works. However, the fact that the bit-flipping algorithm is probabilistic can be a security issue. The security proofs in [Per12; Per13] do assume that the KEM is able to decrypt a KEM ciphertext with "overwhelming probability". As there is no good way to estimate the failure rate for a given QC-MDPC code, the best thing people can do is to run a large amount of experiments. `QcBits` manages to achieve no decoding failures in 10^8 trials. Indeed, 10^8 is not a trivial number, but whether such level of failure rate is enough to keep the system secure remains unclear, not to mention that this is for 80-bit security only. See Section 5.6 for more detailed experimental results on failure rates.

There can be some ways to mitigate the problem. Some researchers are now working on reducing the failure rate by inventing more sophisticated decoding algorithms. Some researchers are now working on designing new parameter sets that allow lower failure rates. Such research is certainly valuable. However, they still do not necessarily answer the question of whether the system is secure under the (most likely estimated) failure rate.

Another probably less serious problem is that `QcBits` and all previous implementation papers [HMG13; MG14a; MG14b; MHG16; MOG15] force the parity check matrix H to have equal weights in $H^{(0)}$ and $H^{(1)}$, which is not the same as what was described in [MTS$^+$13]. `QcBits` restricts the key space in this way to reduce the failure rate. Of course, one can argue that even if the key space is not restricted, for a very high probability $H^{(0)}$ and $H^{(1)}$ would still have the same weight. However, such an argument is valid only if the adversary can only target one system. For an adversary who aims to break one out of many systems, it is still unclear whether such restriction affects the security. Hopefully researchers will spend time on this problem also.

5.6 Experimental results for decoding

This section shows experimental results for QC-MDPC decoding under different parameter sets. The decoding algorithm used is the precomputed-threshold approach introduced in Section 5.1.2. The codes are restricted: $H^{(0)}$ and $H^{(1)}$ are required to have the same row weight. `n, w, t` have same meaning as in Section 5.1.1. `sec` indicates the security level. `T` is the list of thresholds. If not specified otherwise, the thresholds are obtained using the formulas in [MTS$^+$13, Appendix A]. `S` is a list that denotes how many iterations the tests take. The summation of the numbers in `S` is the total number of tests, which is set to 10^8. The 10^8 tests consist of 10^4 decoding attempts for each of 10^4 key pairs. The first number in the list indicates the number of tests that fail to decode in the total number of iterations (i.e., the length of `T`). The second number indicates the number of tests that succeed after 1 iteration. The third number indicates the number of tests that succeed after 2 iterations; etc. `avg` indicates the average number of iterations for the successful tests.

```
n = 4801
w = 90
t = 84
sec = 80
T = [29, 27, 25, 24, 23, 23]
S = [0, 0, 752, 69732674, 30232110, 34417, 47]
avg = 3.30
```

The thresholds are obtained by interactive experiments. `QcBits` uses this setting.

```
n = 4801
w = 90
t = 84
sec = 80
T = [28, 26, 24, 23, 23, 23, 23, 23, 23, 23]
S = [40060, 0, 9794, 87815060, 12079266, 51387, 3833, 519, 70, 10,
1]
avg = 3.12
```

```
n = 9857
w = 142
t = 134
sec = 128
T = [44, 42, 40, 37, 36, 36, 36, 36, 36, 36, 36, 36]
S = [689298, 0, 0, 86592, 53307303, 42797368, 2856446, 235479,
24501, 2651, 333, 26, 3]
avg = 4.46
```

6

Auth256: faster binary-field multiplication and faster binary-field MACs

NIST's standard AES-GCM authenticated-encryption scheme uses GHASH to authenticate ciphertext (produced by AES in counter mode) and to authenticate additional data. GHASH converts its inputs into a polynomial and evaluates that polynomial at a secret element of $\mathbb{F}_{2^{128}} = \mathbb{F}_2[x]/(x^{128} + x^7 + x^2 + x + 1)$, using one multiplication in $\mathbb{F}_{2^{128}}$ for each 128-bit input block. The cost of GHASH is an important part of the cost of GCM, and it becomes almost the entire cost when large amounts of non-confidential data are being authenticated without being encrypted, or when a denial-of-service attack is sending a flood of forgeries to consume all available processing time.

Most AES-GCM software implementations rely heavily on table lookups and presumably leak their keys to cache-timing attacks. Käsper and Schwabe [KS09] (CHES 2009) addressed this problem by introducing a constant-time implementation of AES-GCM using 128-bit vector instructions. Their GHASH implementation takes 14.4 cycles/byte on one core of an Intel Core 2 processor. On a newer Intel Sandy Bridge processor the same software takes 13.1 cycles/byte. For comparison, HMAC-SHA1, which is widely used in Internet applications, takes 6.74 Core 2 cycles/byte and 5.18 Sandy Bridge cycles/byte.

Integer-multiplication hardware. Much better speeds than GHASH were already provided by constant-time MACs that used integer multiplication rather than multiplication of polynomials mod 2. Examples include UMAC [BHK$^+$99], Poly1305 [Ber05], and VMAC [Kro07]. Current Poly1305 software from [Moo14b] runs at 1.89

Core 2 cycles/byte and 1.22 Sandy Bridge cycles/byte. VMAC, which uses "pseudo dot products" (see Section 6.3), is even faster than Poly1305.

CPUs include large integer-multiplication units to support many different applications, so it is not a surprise that these MACs are much faster in software than GHASH (including non-constant-time GHASH software; see [KS09]). However, integer multiplication uses many more bit operations than multiplication of polynomials mod 2, so for hardware designers these MACs are much less attractive. MAC choice is a continuing source of tension between software designers and hardware designers.

New speeds for binary-field MACs. This chapter introduces Auth256, an $\mathbb{F}_{2^{256}}$-based MAC at a 2^{255} security level; and a constant-time software implementation of Auth256 running at just 1.89 cycles/byte on a Core 2. We also tried our software on a Sandy Bridge; it runs at just 1.43 cycles/byte. We also have a preliminary Cortex-A8 implementation below 14 cycles/byte.

This new binary-field MAC is not quite as fast as integer-multiplication MACs. However, the gap is quite small, while the hardware advantages of binary fields are quite important. We plan to put our software into the public domain.

Caveat: All of the above performance figures ignore short-message overhead, and in particular our software has very large overhead, tens of thousands of cycles. For 32-, 64-, 128-kilobyte messages, our software takes 3.07, 2.44, 2.14 Core 2 cycles per byte. and 2.85, 2.09, 1.74 Sandy Bridge cycles per byte. This software is designed primarily for authenticating large files, not for authenticating network packets. However, a variant of Auth256 ($b = 1$ in Section 6.5) takes only 0.81 additional cycles/byte and has much smaller overhead. We also expect that, compared to previous MAC designs, this variant will allow significantly lower area for high-throughput hardware, as explained below.

New bit-operation records for binary-field multiplication. The software speed advantage of Auth256 over GHASH, despite the much higher security level of Auth256, is easily explained by the following comparison. Schoolbook multiplication would take 128^2 ANDs and approximately 128^2 XORs for each 128 bits of GHASH input, i.e., approximately 256 bit operations per authenticated bit. Computing a 256-bit authenticator in the same way would use approximately 512 bit operations per authenticated bit. Auth256 uses just 29 bit operations per authenticated bit.

Of course, Karatsuba's method saves many bit operations at this size. See, e.g., [Ber00], [RK03], [CKP+05], [WP06], [GS06], [PL07], [Ber09a], [Ber09b], and [DSV13]. Bernstein's Karatsuba/Toom combination in [Ber09b] multiplies 256-bit polynomials using only about $133 \cdot 256$ bit operations. Multiplying 256-bit field elements has only a small overhead. However, 133 bit operations is still much larger than 29 bit operations.

Our improved number of bit operations is a combination of four factors. The first factor is faster multiplication: we reduce the cost of multiplication in $\mathbb{F}_{2^{256}}$ from $133 \cdot 256$ bit operations to just $22292 \approx 87 \cdot 256$ bit operations. The second factor, which we do not take credit for, is the use of pseudo dot products to reduce the number of multiplications by a factor of 2, reducing 87 below 44. The third factor, which reduces 44 to 32, is an extra speedup from an interaction between the structure of pseudo dot products and the structure of the multiplication algorithms that we use. The fourth factor, which reduces 32 to just 29, is to use a different field representation for the input to favor the fast multiplication algorithm we use.

Specifically, we use a fast Fourier transform (FFT) to multiply polynomials in $\mathbb{F}_{2^8}[x]$. The FFT is advertised in algorithm courses as using an essentially linear number of field additions and multiplications but is generally believed to be much slower than other multiplication methods for cryptographic sizes. Bernstein, Chou, and Schwabe showed at CHES 2013 [BCS13] that a new "additive FFT" saves time for decryption in McEliece's code-based public-key cryptosystem, but the smallest FFT sizes in [BCS13] were above 10000 bits (evaluation at every element in $\mathbb{F}_{2^m}$, where $m \geq 11$). We introduce an improved additive FFT that uses fewer bit operations than any previously known multiplier for fields as small as $\mathbb{F}_{2^{64}}$, provided that the fields contain $\mathbb{F}_{2^8}$. Our additive FFT, like many AES hardware implementations, relies heavily on a tower-field representation of $\mathbb{F}_{2^8}$, but benefits from this representation in different ways from AES. The extra speedup inside pseudo dot products comes from merging inverse FFTs, which requires breaking the standard encapsulation of polynomial multiplication; see Section 6.3.

The fact that we are optimizing bit operations is also the reason that we expect our techniques to produce low area for high-throughput hardware. Optimizing the area of a fully unrolled hardware multiplier is highly correlated with optimizing the number of bit operations. We do not claim relevance to very small serial multipliers.

Polynomial-multiplication hardware: PCLMULQDQ. Soon after [KS09], in response to the performance and security problems of AES-GCM software, Intel added "AES New Instructions" to some of its CPUs. These instructions include PCLMULQDQ, which computes a 64-bit polynomial multiplication in $\mathbb{F}_2[x]$.

Krovetz and Rogaway reported in [KR11] that GHASH takes 2 Westmere cycles/byte using PCLMULQDQ. Intel's Shay Gueron reported in [Gue13] that heavily optimized GHASH implementations using PCLMULQDQ take 1.79 Sandy Bridge cycles/byte. Our results are faster at a higher security level, although they do require switching to a different authenticator.

Of course, putting sufficient resources into a hardware implementation will beat any software implementation. To quantify this, consider what is required for GHASH to run faster than 1.43 cycles/byte using PCLMULQDQ. GHASH performs one multiplication for every 16 bytes of input, so it cannot afford more than 22.88 cycles for each multiplication. If PCLMULQDQ takes t cycles and t is not very small then presumably Karatsuba is the best approach to multiplication in $\mathbb{F}_{2^{128}}$, taking $3t$ cycles plus some cycles for latency, additions, and reductions.

The latest version of Fog's well-known performance survey [Fog14] indicates that $t = 7$ for AMD Bulldozer, Piledriver, and Steamroller and that $t = 8$ for Intel Sandy Bridge and Ivy Bridge; on the other hand, $t = 2$ for Intel Haswell and $t = 1$ for AMD Jaguar. Gueron, in line with this analysis, reported 0.40 Haswell cycles/byte for GHASH.

It is quite unclear what to expect from future CPUs. Intel did not put hardware for PCLMULQDQ into its low-cost "Core i3" lines of Sandy Bridge, Ivy Bridge, and Haswell CPUs; and obviously Intel is under pressure from other manufacturers of small, low-cost CPUs. To emphasize the applicability of our techniques to a broad range of CPUs, we have avoided PCLMULQDQ in our software.

6.1 Field arithmetic in $\mathbb{F}_{2^8}$

This section reports optimized circuits for field arithmetic in $\mathbb{F}_{2^8}$. We write "circuit" here to mean a fully unrolled combinatorial circuit consisting of AND gates and XOR gates. Our main cost metric is the total number of bit operations, i.e., the total number of AND gates and XOR gates, although as a secondary metric we try to reduce the number of registers required in our software.

Subsequent sections use these circuits as building blocks. The techniques also apply to larger $\mathbb{F}_{2^s}$, but $\mathbb{F}_{2^8}$ is large enough to support the FFTs that we use in this chapter.

6.1.1 Review of tower fields

We first construct $\mathbb{F}_{2^2}$ in the usual way as $\mathbb{F}_2[x_2]/(x_2^2 + x_2 + 1)$. We write α_2 for the image of x_2 in $\mathbb{F}_{2^2}$, so $\alpha_2^2 + \alpha_2 + 1 = 0$. We represent elements of $\mathbb{F}_{2^2}$ as linear combinations of 1 and α_2, where the coefficients are in $\mathbb{F}_2$. Additions in $\mathbb{F}_{2^2}$ use 2 bit operations, namely 2 XORs.

We construct $\mathbb{F}_{2^4}$ as $\mathbb{F}_{2^2}[x_4]/(x_4^2 + x_4 + \alpha_2)$, rather than using a polynomial basis for $\mathbb{F}_{2^4}$ over $\mathbb{F}_2$. We write α_4 for the image of x_4 in $\mathbb{F}_{2^4}$. We represent elements of $\mathbb{F}_{2^4}$ as linear combinations of 1 and α_4, where the coefficients are in $\mathbb{F}_{2^2}$. Additions in $\mathbb{F}_{2^4}$ use 4 bit operations.

Finally, we construct $\mathbb{F}_{2^8}$ as $\mathbb{F}_{2^4}[x_8]/(x_8^2 + x_8 + \alpha_2\alpha_4)$; write α_8 for the image of x_8 in $\mathbb{F}_{2^8}$; and represent elements of $\mathbb{F}_{2^8}$ as $\mathbb{F}_{2^4}$-linear combinations of 1 and α_8. Additions in $\mathbb{F}_{2^8}$ use 8 bit operations.

6.1.2 Variable multiplications

A variable multiplication is the computation of ab given $a, b \in \mathbb{F}_{2^s}$ as input. We say "variable multiplication" to distinguish this operation from multiplication by a constant; we will optimize constant multiplication later.

For variable multiplication in $\mathbb{F}_{2^2}$, we perform a multiplication of $a_0 + a_1x, b_0 + b_1x \in \mathbb{F}_2[x]$ and reduction modulo $x^2 + x + 1$. Here is a straightforward sequence of 7 operations using schoolbook polynomial multiplication: $c_0 \leftarrow a_0 \otimes b_0; c_1 \leftarrow a_0 \otimes b_1; c_2 \leftarrow a_1 \otimes b_0; c_3 \leftarrow a_1 \otimes b_1; c_4 \leftarrow c_1 \oplus c_2; c_5 \leftarrow c_0 \oplus c_3; c_6 \leftarrow c_4 \oplus c_3$. The result is c_5, c_6.

For $\mathbb{F}_{2^4}$ and $\mathbb{F}_{2^8}$ we use 2-way Karatsuba. Note that since the irreducible polynomials are of the form $x^2+x+\alpha$ the reductions involve a different type of multiplication described below: multiplication of a field element with a constant.

We end up with just 110 bit operations for variable multiplication in $\mathbb{F}_{2^8}$. For comparison, Bernstein [Ber09b] reported 100 bit operations to multiply 8-bit polynomials in $\mathbb{F}_2[x]$, but reducing modulo an irreducible polynomial costs many extra operations. A team led by NIST [Tea10], improving upon various previous results such as [IHT06], reported 117 bit operations to multiply in $\mathbb{F}_2[x]$ modulo the AES polynomial $x^8 + x^4 + x^3 + x + 1$.

6.1.3 Constant multiplications

A constant multiplication in $\mathbb{F}_{2^s}$ is the computation of αb given $b \in \mathbb{F}_{2^s}$ as input for some constant $\alpha \in \mathbb{F}_{2^s}$. This is trivial for $\alpha \in \mathbb{F}_2$ so we focus on $\alpha \in \mathbb{F}_{2^s} \backslash \mathbb{F}_2$. One can substitute a specific α into our 110-gate circuit for variable multiplication to obtain a circuit for constant multiplication, and then shorten the circuit by eliminating multiplications by 0, multiplications by 1, additions of 0, etc.; but for small fields it is much better to use generic techniques to optimize the cost of multiplying by a constant matrix.

Our linear-map circuit generator combines various features of Paar's greedy additive common-subexpression elimination algorithm [Paa97] and Bernstein's two-operand "xor-largest" algorithm [Ber09c]. For $\alpha \in \mathbb{F}_{2^8} \backslash \mathbb{F}_2$ our constant-multiplication circuits use 14.83 gates on average. Compared to Paar's results, this is slightly more gates but is much better in register use; compared to Bernstein's results, it is considerably fewer gates.

The real importance of the tower-field construction for us is that constant multiplications become much faster when the constants are in subfields. Multiplying an element of $\mathbb{F}_{2^8}$ by a constant $\alpha \in \mathbb{F}_{2^4} \backslash \mathbb{F}_2$ takes only 7.43 gates on average, and multiplying an element of $\mathbb{F}_{2^8}$ by a constant $\alpha \in \mathbb{F}_{2^2} \backslash \mathbb{F}_2$ takes only 4 gates on average. The constant multiplications in our FFT-based multiplication algorithms for $\mathbb{F}_{2^{256}}$ (see Section 6.2) are often in subfields of $\mathbb{F}_{2^8}$, and end up using only 9.02 gates on average.

6.1.4 Subfields and decomposability

A further advantage of the tower-field construction, beyond the number of bit operations, is that it allows constant multiplications by subfield elements to be decomposed into independent subcomputations. For example, when an $\mathbb{F}_{2^8}$ element in this representation is multiplied by a constant in $\mathbb{F}_{2^2}$, the computation decomposes naturally into 4 independent subcomputations, each of which takes 2 input bits to 2 output bits.

Decomposability is a nice feature for software designers; it guarantees a smaller working set, which in general implies easier optimization, fewer memory operations and cache misses, etc. The ideal case is when the working set can fit into registers; in this case the computation can be done using the minimum number of memory accesses. Section 6.4 gives an example of how decomposability can be exploited to help optimization of a software implementation.

The decomposition of multiplication by a constant in a subfield has the extra feature that the subcomputations are identical. This allows extra possibilities for efficient vectorization in software, and can also be useful in hardware implementations that reuse the same circuit several times. Even when subcomputations are not identical, decomposability increases flexibility of design and is desirable in general.

6.2 Faster additive FFTs

Given a 2^{m-1}-coefficient polynomial f with coefficients in $\mathbb{F}_{2^8}$, a size-2^m additive FFT computes $f(0), f(\beta_m), f(\beta_{m-1}), f(\beta_m + \beta_{m-1}), f(\beta_{m-2})$, etc., where $\beta_m, \ldots, \beta_2, \beta_1$ are $\mathbb{F}_2$-linearly independent elements of $\mathbb{F}_{2^8}$ specified by the algorithm. We always choose a "Cantor basis", i.e., elements $\beta_m, \ldots, \beta_2, \beta_1$ satisfying $\beta_{i+1}^2 + \beta_{i+1} = \beta_i$ and $\beta_1 = 1$; specifically, we take $\beta_1 = 1$, $\beta_2 = \alpha_2$, $\beta_3 = \alpha_4 + 1$, $\beta_4 = \alpha_2\alpha_4$, $\beta_5 = \alpha_8$, and $\beta_6 = \alpha_2\alpha_8 + \alpha_2\alpha_4 + \alpha_2 + 1$. We do not need larger FFT sizes in this chapter.

Our additive FFT is an improvement of the Bernstein–Chou–Schwabe [BCS13] additive FFT, which in turn is an improvement of the Gao–Mateer [GM10] additive FFT. This section presents details of our size-4, size-8, and size-16 additive FFTs over $\mathbb{F}_{2^8}$. All of our improvements are already visible for size 16. At the end of the section gate counts for all sizes are collected and compared with state-of-the-art Karatsuba/Toom-based methods.

6.2.1 Size-4 FFTs: the lowest level of recursion

Given a polynomial $f = a + bx \in \mathbb{F}_{2^8}[x]$, the size-4 FFT computes $f(0) = a, f(\beta_2) = a + \beta_2 b, f(1) = a + b, f(\beta_2 + 1) = a + (\beta_2 + 1)b$. Recall that $\beta_2 = \alpha_2$ so $\beta_2^2 + \beta_2 + 1 = 0$. The size-4 FFT is of interest because it serves as the lowest level of recursion for larger-size FFTs.

As mentioned in Section 6.1, since $\beta_2 \in \mathbb{F}_{2^2}$, the size-4 FFT can be viewed as a collection of 4 independent pieces, each dealing with only 2 out of the 8 bits.

Let a_0, a_1 be the first 2 bits of a; similarly for b. Then a_0, a_1 and b_0, b_1 represent $a_0 + a_1\beta_2, b_0 + b_1\beta_2 \in \mathbb{F}_{2^2}$. Since $\beta_2(a_0 + a_1\beta_2) = a_1 + (a_0 + a_1)\beta_2$, a 6-gate circuit that carries out the size-4 FFTs operations on the first 2 bits is $c_{00} \leftarrow a_0; c_{01} \leftarrow a_1; c_{20} \leftarrow a_0 \oplus b_0; c_{21} \leftarrow a_1 \oplus b_1; c_{10} \leftarrow a_0 \oplus b_1; c_{31} \leftarrow a_1 \oplus b_0; c_{11} \leftarrow c_{31} \oplus b_1; c_{30} \leftarrow c_{10} \oplus b_0$. Then c_{00}, c_{01} is the 2-bit result of a; c_{10}, c_{11} is the 2-bit result of $a + \beta_2 b$; similarly for c_{20}, c_{21} and c_{30}, c_{31}. In conclusion, a size-4 FFT can be carried out using a $6 \cdot 4 = 24$-gate circuit.

The whole computation costs the same as merely 3 additions in $\mathbb{F}_{2^8}$. This is the result of having evaluation points to be in the smallest possible subfield, namely $\mathbb{F}_{2^2}$, and the use of tower field construction for $\mathbb{F}_{2^8}$.

6.2.2 The size-8 FFTs: the first recursive case

Given a polynomial $f = f_0 + f_1 x + f_2 x^2 + f_3 x^3 \in \mathbb{F}_{2^8}[x]$, the size-8 FFT computes $f(0), f(\beta_3), f(\beta_2), f(\beta_2 + \beta_3), f(1), f(\beta_3 + 1), f(\beta_2 + 1), f(\beta_2 + \beta_3 + 1)$. Recall that $\beta_3 = \alpha_4 + 1$ so $\beta_3^2 + \beta_3 + \beta_2 = 0$. The size-8 FFT is of interest because it is the smallest FFT that involves recursion.

In general, a recursive size-2^m FFT starts with a radix conversion that computes $f^{(0)}$ and $f^{(1)}$ such that $f = f^{(0)}(x^2 + x) + x f^{(1)}(x^2 + x)$. When f is a 2^{m-1}-coefficient polynomial we call this a size-2^{m-1} radix conversion. Since the size-4 radix conversion can be viewed as a change of basis in $\mathbb{F}_2^4$, each coefficient in $f^{(0)}$ and $f^{(1)}$ is a subset sum of f_0, f_1, f_2, and f_3. In fact, $f^{(0)} = f_0 + (f_2 + f_3)x$ and $f^{(1)} = (f_1 + f_2 + f_3) + f_3 x$ can be computed using exactly 2 additions.

After the radix conversion, 2 size-4 FFTs are invoked to evaluate $f^{(0)}$, $f^{(1)}$ at $0^2+0=0$, $\beta_3^2+\beta_3=\beta_2$, $\beta_2^2+\beta_2=1$, and $(\beta_2+\beta_3)^2+(\beta_2+\beta_3)=\beta_2+1$. Each of these size-4 FFTs takes 24 bit operations.

Note that we have

$$f(\alpha) = f^{(0)}(\alpha^2+\alpha) + \alpha f^{(1)}(\alpha^2+\alpha),$$
$$f(\alpha+1) = f^{(0)}(\alpha^2+\alpha) + (\alpha+1) f^{(1)}(\alpha^2+\alpha).$$

Starting from $f^{(0)}(\alpha^2+\alpha)$ and $f^{(1)}(\alpha^2+\alpha)$, Gao and Mateer multiply $f^{(1)}(\alpha^2+\alpha)$ by α and add $f^{(0)}(\alpha^2+\alpha)$ to obtain $f(\alpha)$, and then add $f^{(1)}(\alpha^2+\alpha)$ with $f(\alpha)$ to obtain $f(\alpha+1)$. We call this a **muladdadd** operation.

The additive FFT thus computes all the pairs $f(\alpha), f(\alpha+1)$ at once: given $f^{(0)}(0)$ and $f^{(1)}(0)$ apply muladdadd to obtain $f(0)$ and $f(1)$, given $f^{(0)}(\beta_2)$ and $f^{(1)}(\beta_2)$ apply muladdadd operation to obtain $f(\beta_3)$ and $f(\beta_3+1)$, and so on.

The way that the output elements form pairs is a result of having 1 as the last basis element. In general the Gao–Mateer FFT is able to handle the case where 1 is not in the basis with some added cost, but here we avoid the cost by making 1 the last basis element.

Generalizing this to the case of size-2^m FFTs implies that the i-th output element of $f^{(0)}$ and $f^{(1)}$ work together to form the ith and $(i+2^{m-1})$th output element for f. We call the collection of muladdadds that are used to combine 2 size-2^{m-1} FFT outputs to form a size-2^m FFT output a **size-2^m combine routine**.

We use our circuit generator introduced in Section 6.1 to generate the circuits for all the constant multiplications. The muladdadds take a total of 76 gates. Therefore, a size-8 FFT can be carried out using $2\cdot 8 + 2\cdot 24 + 76 = 140$ gates.

Note that for a size-8 FFT we again benefit from the special basis and the $\mathbb{F}_{2^8}$ construction. The recursive calls still use the good basis $\beta_2, 1$ so that there are only constant multiplications by $\mathbb{F}_{2^2}$ elements. The combine routine, although not having only constant multiplications by $\mathbb{F}_{2^2}$ elements, at least has only constant multiplications by $\mathbb{F}_{2^4}$ elements.

6.2.3 The size-16 FFTs: saving additions for radix conversions

The size-16 FFT is the smallest FFT in which non-trivial radix conversions happen in recursive calls. Gao and Mateer presented an algorithm performing a size-2^n radix conversion using $(n-1)2^{n-1}$ additions. We do better by combining additions across levels of recursion.

The size-8 radix conversion finds $f^{(0)}$, $f^{(1)}$ such that $f = f^{(0)}(x^2+x) + xf^{(1)}(x^2+x)$. The two size-4 radix conversion in size-8 FFT subroutines find $f^{(i0)}$, $f^{(i1)}$ such that $f^{(i)} = f^{(i0)}(x^2+x) + xf^{(i1)}(x^2+x)$ for $i \in \{0,1\}$. Combining all these leads to $f = f^{(00)}(x^4+x) + (x^2+x)f^{(01)}(x^4+x) + xf^{(10)}(x^4+x) + x(x^2+x)f^{(11)}(x^4+x)$.

In the end the size-8 and the two size-4 radix conversions together compute from f the following: $f^{(00)} = f_0 + (f_4+f_7)x$, $f^{(01)} = (f_2+f_3+f_5+f_6) + (f_6+f_7)x$, $f^{(10)} = (f_1+f_2+f_3+f_4+f_5+f_6+f_7) + (f_5+f_6+f_7)x$, and $f^{(11)} = (f_3+f_6) + f_7x$. The Gao–Mateer algorithm takes 12 additions for this computation, but one sees by hand that 8 additions suffice. One can also obtain this result by applying the circuit generator introduced in Section 6.1. Here is an 8-addition sequence generated by the circuit generator: $f_0^{(00)} \leftarrow f_0; f_1^{(11)} \leftarrow f_7; f_1^{(00)} \leftarrow f_4+f_7; f_0^{(01)} \leftarrow f_2+f_5; f_0^{(11)} \leftarrow f_3+f_6; f_1^{(01)} \leftarrow f_6+f_7; f_0^{(10)} \leftarrow f_1+f_1^{(00)}; f_1^{(10)} \leftarrow f_5+f_1^{(01)}; f_0^{(01)} \leftarrow f_0^{(01)}+f_0^{(11)}; f_0^{(10)} \leftarrow f_0^{(10)}+f_0^{(01)}$.

We applied the circuit generator for larger FFTs and found larger gains. A size-32 FFT, in which the input is a size-16 polynomial, requires 31 rather than 48 additions for radix conversions. A size-64 FFT, in which the input is a size-32 polynomial, requires 82 rather than 160 additions for radix conversions.

We also applied our circuit generator to the muladdadds, obtaining a 170-gate circuit for the size-16 combined routine and thus a size-16 FFT circuit using $8 \cdot 8 + 4 \cdot 24 + 2 \cdot 76 + 170 = 482$ gates.

6.2.4 Size-16 FFTs continued: decomposition at field-element level

The size-16 FFT also illustrates the decomposability of the combine routines of a FFT. Consider the size-16 and size-8 combine routines; the computation takes as input the FFT outputs for the $f^{(ij)}$'s to compute the FFT output for f.

Let the output for f be $a_0, a_1, \ldots, a_{15}$, the output for $f^{(i)}$ be $a_0^{(i)}, a_1^{(i)}, \ldots, a_7^{(i)}$, and similarly for $f^{(ij)}$. For $k \in \{0,1,2,3\}$, a_k, a_{k+8} are functions of $a_k^{(0)}$ and $a_k^{(1)}$, which in turn are functions of $a_k^{(00)}$, $a_k^{(01)}$, $a_k^{(10)}$, and $a_k^{(11)}$; a_{k+4}, a_{k+12} are functions of $a_{k+4}^{(0)}$ and $a_{k+4}^{(1)}$, which in turn are functions of the same 4 elements. We conclude that $a_k, a_{k+4}, a_{k+8}, a_{k+12}$ depend only on $a_k^{(00)}$, $a_k^{(01)}$, $a_k^{(10)}$, and $a_k^{(11)}$. In this way, the computation is decomposed into 4 independent parts; each takes as input 4 field elements and outputs 4 field elements. Note that here the decomposition is at the field-element level, while Section 6.1 considered decomposability at the bit level.

More generally, for size-2^m FFTs we suggest decomposing k levels of combine routines into 2^{m-k} independent pieces, each taking 2^k $\mathbb{F}_{2^8}$ elements as input and producing 2^k $\mathbb{F}_{2^8}$ elements as output.

6.2.5 Improvements: a summary

We have two main improvements to the additive FFT: reducing the cost of multiplications and reducing the number of additions in radix conversion. We also use these ideas to accelerate size-32 and size-64 FFTs, and obviously they would also save time for larger FFTs.

The reduction in the cost of multiplications is a result of (1) choosing a "good" basis for which constant multiplications use constants in the smallest possible subfield; (2) using a tower-field representation to accelerate those constant multiplications; and (3) searching for short sequences of additions. The reduction of additions for radix conversion is a result of (1) merging radix conversion at different levels of recursion and again (2) searching for short sequences of additions.

b	forward	pointwise	inverse	total	competition
16	$2 \cdot 24$	$4 \cdot 110$	60	$448 \approx 14 \cdot 2 \cdot 16$	$350 \approx 10.9 \cdot 2 \cdot 16$
32	$2 \cdot 140$	$8 \cdot 110$	228	$1388 \approx 21.7 \cdot 2 \cdot 32$	$1158 \approx 18.1 \cdot 2 \cdot 32$
64	$2 \cdot 482$	$16 \cdot 110$	746	$3470 \approx 27.1 \cdot 2 \cdot 64$	$3682 \approx 28.8 \cdot 2 \cdot 64$
128	$2 \cdot 1498$	$32 \cdot 110$	2066	$8582 \approx 33.5 \cdot 2 \cdot 128$	$11486 \approx 44.9 \cdot 2 \cdot 128$
256	$2 \cdot 4068$	$64 \cdot 110$	5996	$21172 \approx 41.4 \cdot 2 \cdot 256$	$34079 \approx 66.6 \cdot 2 \cdot 256$

Table 6.1: Cost of multiplying $b/8$-coefficient polynomials over $\mathbb{F}_{2^8}$. "Forward" is the cost of two size-$b/4$ FFTs with size-$b/8$ inputs. "Pointwise" is the cost of pointwise multiplication. "Inverse" is the cost of an inverse size-$b/4$ FFT. "Total" is the sum of forward, pointwise, and inverse. "Competition" is the cost from [Ber09b] of an optimized Karatsuba/Toom multiplication of b-coefficient polynomials over $\mathbb{F}_2$; note that slight improvements appear in [DSV13].

6.2.6 Polynomial multiplications: a comparison with Karatsuba and Toom

Just like other FFT algorithms, any additive FFT can be used to multiply polynomials. Given two 2^{m-1}-coefficient polynomials in $\mathbb{F}_{2^s}$, we apply a size-2^m additive FFT to each polynomial, a pointwise multiplication consisting of 2^m variable multiplications in $\mathbb{F}_{2^s}$, and a size-2^m inverse additive FFT, i.e., the inverse of an FFT with both input and output size 2^m. An FFT (or inverse FFT) with input and output size 2^m is slightly more expensive than an FFT with input size 2^{m-1} and output size 2^m: input size 2^{m-1} is essentially input size 2^m with various 0 computations suppressed.

Table 6.1 summarizes the number of bit operations required for multiplying b-bit (i.e., $b/8$-coefficient) polynomials in $\mathbb{F}_{2^8}[x]$. Field multiplication is slightly more expensive than polynomial multiplication. For $\mathbb{F}_{2^{256}}$ we use the polynomial $x^{32}+x^{17}+x^2+\alpha_8$; reduction costs 992 bit operations. However, as explained in Section 6.3, in the context of Auth256 we can almost eliminate the inverse FFT and the reduction, and eliminate many operations in the forward FFTs, making the additive FFT even more favorable than Karatsuba.

6.3 The Auth256 message-authentication code: major features

Auth256, like GCM's GHASH, follows the well-known Wegman–Carter [WC81] recipe for building an MAC with (provably) information-theoretic security. The recipe is to apply a (provably) "δ-xor-universal hash" to the message and encrypt the result with a one-time pad. Every forgery attempt then (provably) has success probability at most δ, no matter how much computer power the attacker used.

Of course, real attackers do not have unlimited computer power, so GCM actually replaces the one-time pad with counter-mode AES output to reduce key size. This is safe against any attacker who cannot distinguish AES output from uniform random; see, e.g., [IOM12, comments after Corollary 3]. Similarly, it is safe to replace the one-time pad in Auth256 with cipher output.

This section presents two important design decisions for Hash256, the hash function inside Auth256. Section 6.3.1 describes the advantages of the Hash256 output size. Section 6.3.2 describes the choice of pseudo dot products inside Hash256, and the important interaction between FFTs and pseudo dot products. Section 6.3.3 describes the use of a special field representation for inputs to reduce the cost of FFTs.

Section 6.5 presents, for completeness, various details of Hash256 and Auth256 that are not relevant to this chapter's performance evaluation.

6.3.1 Output size: bigger-birthday-bound security

Hash256 produces 256-bit outputs, as its name suggests, and Auth256 produces 256-bit authenticators. Our multiplication techniques are only slightly slower per bit for $\mathbb{F}_{2^{256}}$ than for $\mathbb{F}_{2^{128}}$, so Auth256 is only slightly slower than an analogous Auth128 would be. An important advantage of an increased output size is that one can safely eliminate nonces.

Encrypting a hash with a one-time pad, or with a stream cipher such as AES in counter mode, requires a nonce, and becomes insecure if the user accidentally repeats a nonce; see, e.g., [HP08]. Directly applying a PRF (as in HMAC) or PRP (as in WMAC) to the hash, without using a nonce, is much more resilient against misuse but becomes insecure if hashes collide, so b-bit hashes are expected to be broken within $2^{b/2}$ messages (even with an optimal $\delta = 2^{-b}$) and already provide a noticeable attack probability within somewhat fewer messages.

This problem has motivated some research into "beyond-birthday-bound" mechanisms for authentication and encryption that can safely be used for more than $2^{b/2}$ messages. See, e.g., [LST12]. Hash256 takes a different approach, which we call "bigger-birthday-bound" security: simply increasing b to 256 (and correspondingly reducing δ) eliminates all risk of collisions. For the same reason, Hash256 provides extra strength inside other universal-hash applications, such as wide-block disk encryption; see, e.g., [Hal07].

In applications with space for only 128-bit authenticators, it is safe to simply truncate the Hash256 and Auth256 output from 256 bits to 128 bits. This increases δ from 2^{-255} to 2^{-127}.

6.3.2 Pseudo dot products and FFT addition

Hash256 uses the same basic construction as UMAC [BHK$^+$99], Badger [BSP$^+$05], NMH [HK97, Section 5], and VMAC [Kro07]: the hash of a message with blocks m_1, m_2, m_3, m_4, $\ldots$ is $(m_1 + r_1)(m_2 + r_2) + (m_3 + r_3)(m_4 + r_4) + \cdots$. Halevi and Krawczyk [HK97] credit this hash to Carter and Wegman; Bernstein [Ber07a] credits it to Winograd and calls it the "pseudo dot product". The pseudo-dot-product construction of Hash256 gives $\delta < 2^{-255}$; see Section 6.6 for the proof.

A simple dot product $m_1r_1 + m_2r_2 + m_3r_3 + m_4r_4 + \cdots$ uses one multiplication per block. The same is true for GHASH and many other polynomial-evaluation hashes. The basic advantage of $(m_1 + r_1)(m_2 + r_2) + (m_3 + r_3)(m_4 + r_4) + \cdots$ is that there are only 0.5 multiplications per block.

For Auth256 each block contains 256 bits, viewed as an element of the finite field $\mathbb{F}_{2^{256}}$. Our cost of Auth256 per 512 authenticated bits is $29 \cdot 512 = 58 \cdot 256$ bit operations, while the our cost for a multiplication in $\mathbb{F}_{2^{256}}$ is $87 \cdot 256$ bit operations. We now explain one of the two sources of this gap.

FFT-based multiplication of two polynomials $f_1 f_2$ has several steps: apply an FFT to evaluate f_1 at many points; apply an FFT to evaluate f_2 at many points; compute the corresponding values of the product $f_1 f_2$ by pointwise multiplication; and apply an inverse FFT to reconstruct the coefficients of $f_1 f_2$. FFT-based multiplication of field elements has the same steps plus a final reduction step.

These steps for $\mathbb{F}_{2^{256}}$, with our optimizations from Section 6.2, cost 4068 bit operations for each forward FFT, $64 \cdot 110$ bit operations for pointwise multiplication, 5996 bit operations for the inverse FFT (the forward FFT is less expensive since more polynomial coefficients are known to be 0), and 992 bit operations for the final reduction. Applying these steps to each 512 bits of input would cost approximately 15.89 bit operations per bit for the two forward FFTs, 13.75 bit operations per bit for pointwise multiplication, 11.71 bit operations per bit for the inverse FFT, and 1.94 bit operations per bit for the final reduction, plus 1.5 bit operations per bit for the three additions in the pseudo dot product.

We do better by exploiting the structure of the pseudo dot product as a sum of the form $f_1 f_2 + f_3 f_4 + f_5 f_6 + \cdots$. Optimizing this computation is not the same problem as optimizing the computation of $f_1 f_2$. Specifically, we apply an FFT to each f_i and compute the corresponding values of $f_1 f_2$, $f_3 f_4$, etc., but we then add these values *before* applying an inverse FFT. See Figure 6.1. There is now only one inverse FFT (and one final reduction) per message, rather than one inverse FFT for every two blocks. Our costs are now 15.89 bit operations per bit for the two forward FFTs, 13.75 bit operations per bit for pointwise multiplication, 1 bit operation per bit for the input additions in the pseudo dot product, and 1 bit operation per bit for the pointwise additions, for a total of 31.64 bit operations per bit, plus a constant (not very large) overhead per message.

This idea is well known in the FFT literature (see, e.g., [Ber08a, Section 2]) but we have never seen it applied to message authentication. It reduces the cost of FFT-based message authentication by a factor of nearly 1.5. Note that this also reduces the cutoff between FFT and Karatsuba.

UMAC and VMAC actually limit the lengths of their pseudo dot products, to limit key size. This means that longer messages produce two or more hashes; these hashes are then further hashed in a different way (which takes more time per byte but is applied to far fewer bytes). For simplicity we instead use a key as long as the maximum message length. We have also considered the small-key hashes from [Ber07a] but those hashes obtain less benefit from merging inverse FFTs.

6.3.3 Embedding invertible linear operations into FFT inputs

Section 6.3.2 shows how to achieve 31.64 bit operations per message bit by skipping the inverse FFTs for almost all multiplications in the pseudo dot product. Now we show how Auth256 achieves 29 bit operations per message bit by skipping operations in the forward FFTs.

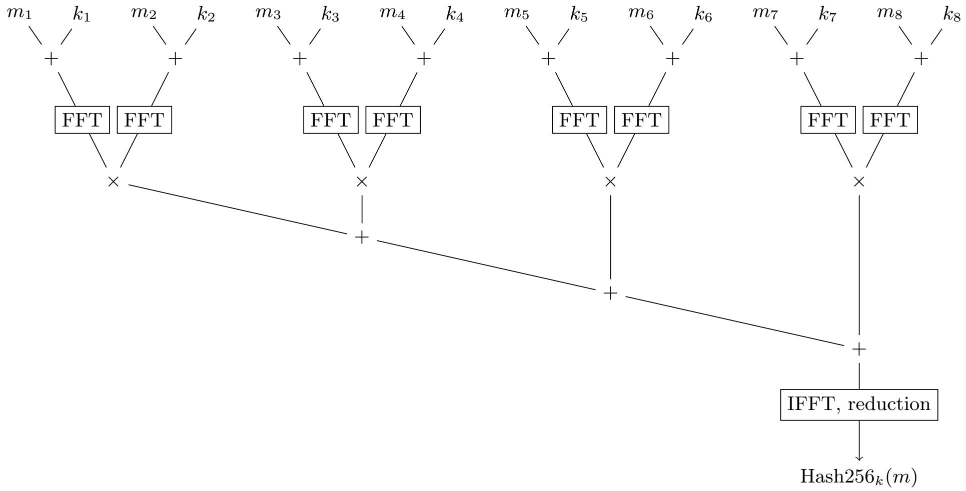

Figure 6.1: Hash256 flowchart

Section 6.2.3 shows that the radix conversions can be merged into one invertible $\mathbb{F}_{2^8}$-linear (actually $\mathbb{F}_2$-linear) map, which takes place before all other operations in the FFT. The input is a $\mathbb{F}_{2^{256}}$ element which is represented as coefficients in $\mathbb{F}_{2^8}$ with respect to a polynomial basis. Applying an invertible linear map on the coefficients implies a change of basis. In other words, the radix conversions convert the input into another 256-bit representation. If we define the input to be elements in this new representation, all the radix conversions can simply be skipped. Note that the authenticator still uses the original representation. See Section 6.5.2 for a definition of the new representation.

This technique saves a significant fraction of the operations in the forward FFT. As shown in Section 6.2, one forward FFT takes 4068 bit operations, where $82 \cdot 8 = 656$ of them are spent on radix conversions. Eliminating all radix conversions then gives the 29 bit operations per message bit.

The additive FFTs described so far are "2-way split" FFTs since they require writing the input polynomial $f(x)$ in the form $f^{(0)}(x^2+x) + xf^{(1)}(x^2+x)$. It is easy to generalize this to a "2^k-way split" in which $f(x)$ is written as $\sum_{i=0}^{2^k-1} x^i f^{(i)}(\psi^k(x))$, where $\psi(x) = x^2 + x$. In particular, Gao and Mateer showed how to perform $2^{2^{k-1}}$-way-split FFTs for polynomials in $\mathbb{F}_{2^{2^k}}[x]$. The technique of changing input representation works for any 2^k-way split. In fact we found that with 8-way-split FFTs, the number of bit operations per message bit can be slightly better than 29. However, for simplicity, Auth256 is defined in a way that favors 2-way-split FFTs.

6.4 Software implementation

Our software implementation uses bitslicing. This means that we convert each bit in previous sections into w bits, where w is the register width on the machine; we convert each AND into a bitwise w-bit AND instruction; and we convert each XOR into a bitwise w-bit XOR instruction.

Bitslicing is efficient only if there is adequate parallelism in the algorithm. Fortunately, the pseudo-dot-product computation is naturally parallelizable: we let the jth bit position compute the sum of all products $(m_{2i+1} + r_{2i+1})(m_{2i+2} + r_{2i+2})$ where $i \equiv j \pmod{w}$. After all the products are processed, the results in all bit positions are summed up to get the final value.

The detailed definition of Auth256 (see Section 6.5) has a parameter b. Our software takes $b = w$, allowing it to simply pick up message blocks as vectors. If b is instead chosen as 1 then converting to bitsliced form requires a transposition of message blocks; in our software this transposition costs an extra 0.81 cycles/byte.

6.4.1 Minimizing memory operations in radix conversions

We exploit the decomposability of additions to minimize memory operations for a radix conversion. When dealing with size-2^k radix conversions with $k \leq 4$, we decompose at bit level the computation into 2^k parts, each of which deals with $16/2^k$ bit positions. This minimizes the number of loads and stores. The same technique applies for a radix conversion combined with smaller-size radix conversions in the FFT subroutines.

Our implementation uses the size-16 FFT as a subroutine. Inside a size-16 FFT the size-8 radix conversion is combined with the 2 size-4 radix conversions in FFT subroutines. Our bit-operation counts handle larger radix conversions in the same way, but in our software we sacrifice some of the bit operations saved here to improve instruction-level parallelism and register utilization. For size-16 radix conversion the decomposition method is adopted. For size-32 radix conversion the decomposition method is used only for the size-16 recursive calls.

6.4.2 Minimizing memory operations in muladdadd operations

For a single muladdadd operation $a \leftarrow a + \alpha b; b \leftarrow b + a$, each of a and b consumes 8 vectors; evidently at least 16 loads and 16 stores are required. While we showed how the sequence of bit operations can be generated, it does not necessarily mean that there are enough registers to carry out the bit operations using the minimum number of loads and stores.

Here is one strategy to maintain both the number of bit operations and the lower bound on number of loads and stores. First load the 8 bits of b into 8 registers $t_0, t_1, \ldots, t_7$. Use the sequence of XORs generated by the code generator, starting from the t_i's, to compute the 8 bits of αb, placing them in the other 8 registers $s_0, s_1, \ldots, s_7$. Then perform $s_i \leftarrow s_i \oplus a[i]$, where $a[i]$ is the corresponding bit of a in memory, to obtain $a + \alpha b$. After that overwrite a with the s_i's. Finally, perform $t_i \leftarrow t_i \oplus s_i$ to obtain $a + (\alpha + 1)b$, and overwrite b with the t_i's.

In our software muladdadd operations are handled one by one in size-64 and size-32 combine routines. See below for details about how muladdadds in smaller size combine routines are handled.

6.4.3 Implementing the size-16 additive FFT

In our size-16 FFT implementation the size-8 radix conversion is combined with the two size-4 ones in the FFT subroutines using the decomposition method described earlier in this section. Since the size-4 FFTs deal with constants in $\mathbb{F}_{2^2}$, we further combine the radix conversions with size-4 FFTs.

At the beginning of one of the 4 rounds of the whole computation, the $2 \cdot 8 = 16$ bits of the input for size-8 radix conversion are loaded. Then the logic operations are carried out in registers, and eventually the result is stored in $2 \cdot 16 = 32$ bits of the output elements. The same routine is repeated 4 times to cover all the bit positions.

The size-16 and size-32 combine routines are also merged as shown in Section 6.2. The field-level decomposition is used together with a bit-level decompostion: in size-16 FFT all the constants are in $\mathbb{F}_{2^4}$, so it is possible to decompose any computation that works on field elements into a 2-round procedure and handle 4 bit positions in each round. In conclusion, the field-level decomposition turns the computation into 4 pieces, and the bit-level decompostion further decomposes each of these into 2 smaller pieces. In the end, we have an 8-round procedure.

At the beginning of one of the 8 rounds of the whole computation, the $4 \cdot 4 = 16$ bits of the outputs of the size-4 FFTs are loaded. Then the logic operations are carried out in registers, and eventually the result is stored in $4 \cdot 4 = 16$ bits of the output elements. The same routine is repeated 8 times to cover all the bit positions.

6.5 Auth256: minor details

To close we fill in, for completeness, the remaining details of Hash256 and Auth256.

6.5.1 Review of Wegman–Carter MACs

Wegman–Carter MACs work as follows. The authenticator of the nth message m is $H(r, m) \oplus s_n$. The key consists of independent uniform random $r, s_1, s_2, s_3, \ldots$; the pad is $s_1, s_2, s_3, \ldots$.

The hash function H is designed to be "δ-xor-universal", i.e., to have "differential probability at most δ". This means that, for every message m, every message $m' \neq m$, and every difference Δ, a uniform random r has $H(r, m) \oplus H(r, m') = \Delta$ with probability at most δ.

6.5.2 Field representation

We represent an element of $\mathbb{F}_{2^s}$ as a sequence of s bits. If we construct $\mathbb{F}_{2^s}$ as $\mathbb{F}_{2^t}[x]/\phi$ then we recursively represent the element $c_0 + c_1 x + \cdots + c_{t/s-1} x^{t/s-1}$ as the concatenation of the representations of $c_0, c_1, \ldots, c_{t/s-1}$. At the bottom of the recursion, we represent an element of $\mathbb{F}_2$ as 1 bit in the usual way. See Sections 6.1 and 6.2.6 for the definition of ϕ for $\mathbb{F}_{2^2}$, $\mathbb{F}_{2^4}$, $\mathbb{F}_{2^8}$, and $\mathbb{F}_{2^{256}}$.

As mentioned in Section 6.3.3, we do not use the polynomial basis $1, x, \ldots, x^{31}$ for $\mathbb{F}_{2^{256}}$ inputs. Here we define the representation for them. Let $y_{(b_{k-1}b_{k-2}\cdots b_0)_2} = \prod_{i=0}^{k-1} (\psi^i(x))^{b_i}$, where $\psi(x)$ follows the definition in Section 6.3.3. Then each $\mathbb{F}_{2^{256}}$

input $c_0y_0 + c_1y_1 + \cdots + c_{31}y_{31}$ is defined as the concatenation of the representations of $c_0, c_1, \ldots, c_{31}$. One can verify that $y_0, y_1, \ldots, y_{31}$ is the desired basis by writing down the equation between $f(x)$ and $f^{(00000)}(x)$, $f^{(00001)}(x)$, $\ldots$, $f^{(111111)}(x)$ as in Section 6.2.3.

If $s \geq 8$ then we also represent an element of $\mathbb{F}_{2^s}$ as a sequence of $s/8$ bytes, i.e., $s/8$ elements of $\{0, 1, \ldots, 255\}$. The 8-bit sequence $b_0, b_1, \ldots, b_7$ is represented as the byte $b = \sum_i 2^i b_i$.

6.5.3 Hash256 padding and conversion

Hash256 views messages as elements of $K^0 \cup K^2 \cup K^4 \cup \cdots$, i.e., even-length strings of elements of K, where K is the finite field $\mathbb{F}_{2^{256}}$. It is safe to use a single key with messages of different lengths.

In real applications, messages are strings of bytes, so strings of bytes need to be encoded invertibly as strings of elements of K. The simplest encoding is standard "10*" padding, where a message is padded with a 1 byte and then as many 0 bytes as necessary to obtain a multiple of 64 bytes. Each 32-byte block is then viewed as an element of K.

We define a more general encoding parametrized by a positive integer b; the encoding of the previous paragraph has $b = 1$. The message is padded with a 1 byte and then as many 0 bytes as necessary to obtain a multiple of $64b$ bytes, say $64bN$ bytes. These bytes are split into $2N$ segments $M_0, M'_0, M_1, M'_1, \ldots, M_{N-1}, M'_{N-1}$, where each segment contains $32b$ consecutive bytes. Each segment is then transposed into b elements of K: segment M_i is viewed as a column-major bit matrix with b rows and 256 columns, and row j of this matrix is labeled c_{bi+j}, while c'_{bi+j} is defined similarly using M'_i. This produces $2bN$ elements of K, namely $m_0, m'_0, m_1, m'_1, m_2, m'_2, \ldots, m_{bN-1}, m'_{bN-1}$.

The point of this encoding is to allow a simple bitsliced vectorized implementation; see Section 6.4. Our 1.59 cycle/byte implementation uses $b = 256$. We have also implemented $b = 1$, which costs 0.81 cycles/byte extra for transposition and is compatible with efficient handling of much shorter messages. An interesting intermediate possibility is to take, e.g., $b = 8$, eliminating the most expensive (non-bytewise) transposition steps while still remaining suitable for authentication of typical network packets.

6.5.4 Hash256 and Auth256 keys and authenticators

The Hash256 key is a uniform random byte string of the same length as a maximum-length padded message, representing elements $r_0, r'_0, r_1, r'_1, \ldots$ of K. If the key is actually defined as, e.g., counter-mode AES output then the maximum length does not need to be specified in advance: extra key elements can be computed on demand and cached for subsequent use.

The Hash256 output is $(m_0 + r_0)(m'_0 + r'_0) + (m_1 + r_1)(m'_1 + r'_1) + \cdots$. This is an element of K.

The Auth256 key is a Hash256 key together with independent uniform random elements $s_1, s_2, \ldots$ of K. The Auth256 authenticator of the nth message m_n is

Auth256$(r, m_n) \oplus s_n$.

6.6 Security proof

This section proves that Hash256 has differential probability smaller than 2^{-255}. This is not exactly the same as the proofs for the pseudo-dot-product portions of UMAC and VMAC: UMAC and VMAC specify fixed lengths for their pseudo dot products, whereas we allow variable lengths.

Theorem 1. *Let K be a finite field. Let ℓ, ℓ', k be nonnegative integers with $\ell \le k$ and $\ell' \le k$. Let $m_1, m_2, \ldots, m_{2\ell-1}, m_{2\ell}$ be elements of K. Let $m'_1, m'_2, \ldots, m'_{2\ell'-1}$, $m'_{2\ell'}$ be elements of K. Assume that $(m_1, m_2, \ldots, m_{2\ell}) \ne (m'_1, m'_2, \ldots, m'_{2\ell'})$. Let Δ be an element of K. Let $r_1, r_2, \ldots, r_{2k}$ be independent uniform random elements of k. Let p be the probability that $h = h' + \Delta$, where*

$$\begin{aligned} h &= (m_1 + r_1)(m_2 + r_2) + (m_3 + r_3)(m_4 + r_4) + \cdots \\ &\quad + (m_{2\ell-1} + r_{2\ell-1})(m_{2\ell} + r_{2\ell}), \\ h' &= (m'_1 + r_1)(m'_2 + r_2) + (m'_3 + r_3)(m'_4 + r_4) + \cdots \\ &\quad + (m'_{2\ell'-1} + r_{2\ell'-1})(m'_{2\ell'} + r_{2\ell'}). \end{aligned}$$

Then $p < 2/\#K$. If $\ell = \ell'$ then $p \le 1/\#K$, and if $\ell \ne \ell'$ then $p < 1/\#K + 1/\#K^{|\ell-\ell'|}$.

Proof. Case 1: $\ell = \ell'$. Then $h = h' + \Delta$ if and only if

$$\begin{aligned} &r_1(m_2 - m'_2) + r_2(m_1 - m'_1) + r_3(m_4 - m'_4) + r_4(m_3 - m'_3) + \cdots \\ &\quad = \Delta + m'_1 m'_2 - m_1 m_2 + m'_3 m'_4 - m_3 m_4 + \cdots . \end{aligned}$$

This is a linear equation in $r_1, r_2, \ldots, r_{2k}$. This linear equation is nontrivial: by hypothesis $(m_1, m_2, \ldots, m_{2\ell}) \ne (m'_1, m'_2, \ldots, m'_{2\ell'})$, so there must be some i for which $m_i - m'_i \ne 0$. Consequently there are most $\#K^{2k-1}$ solutions to the equation out of the $\#K^{2k}$ possibilities for r; i.e., $p \le 1/\#K$ as claimed.

Case 2: $\ell < \ell'$ and $(m_1, \ldots, m_\ell) \ne (m'_1, \ldots, m'_\ell)$. Define

$$f = (m'_{2\ell+1} + r_{2\ell+1})(m_{2\ell+2} + r_{2\ell+2}) + \cdots + (m'_{2\ell'-1} + r_{2\ell'-1})(m'_{2\ell'} + r_{2\ell'}).$$

Then $h = h' + \Delta$ if and only if

$$\begin{aligned} &r_1(m_2 - m'_2) + r_2(m_1 - m'_1) + r_3(m_4 - m'_4) + r_4(m_3 - m'_3) + \cdots \\ &\qquad + r_{2\ell-1}(m_{2\ell} - m'_{2\ell}) + r_{2\ell}(m_{2\ell-1} - m'_{2\ell-1}) \\ &\quad = f + \Delta + m'_1 m'_2 - m_1 m_2 + m'_3 m'_4 - m_3 m_4 + \cdots + m'_{2\ell-1} m'_{2\ell} - m_{2\ell-1} m_{2\ell}. \end{aligned}$$

This is a linear equation in $r_1, \ldots, r_{2\ell}$, since f is independent of $r_1, \ldots, r_{2\ell}$. For each choice of $(r_{2\ell+1}, r_{2\ell+2}, \ldots, r_{2k})$, there are at most $\#K^{2\ell-1}$ choices of $(r_1, \ldots, r_{2\ell})$ satisfying this linear equation. Consequently $p \le 1/\#K$ as above.

Case 3: $\ell < \ell'$ and $(m_1, \ldots, m_\ell) = (m'_1, \ldots, m'_\ell)$. Then $h = h' + \Delta$ if and only if $0 = f + \Delta$, where f is defined as above. This is a linear equation in $r_{2\ell+2}, r_{2\ell+4}, \ldots, r_{2\ell'}$ for each choice of $r_{2\ell+1}, r_{2\ell+3}, \ldots, r_{2\ell'-1}$. The linear equation is nontrivial except when $r_{2\ell+1} = -m'_{2\ell+1}$, $r_{2\ell+3} = -m'_{2\ell+3}$, and so on through $r_{2\ell'-1} = -m'_{2\ell'-1}$. The linear equation thus has at most $\#K^{\ell'-\ell-1}$ solutions $(r_{2\ell+2}, r_{2\ell+4}, \ldots, r_{2\ell'})$ for $\#K^{\ell'-\ell} - 1$ choices of $(r_{2\ell+1}, r_{2\ell+3}, \ldots, r_{2\ell'-1})$, plus at most $\#K^{\ell'-\ell}$ solutions $(r_{2\ell+2}, r_{2\ell+4}, \ldots, r_{2\ell'})$ for 1 exceptional choice of $(r_{2\ell+1}, r_{2\ell+3}, \ldots, r_{2\ell'-1})$, for a total of $\#K^{2\ell'-2\ell-1} - \#K^{\ell'-\ell-1} + \#K^{\ell'-\ell} < \#K^{2\ell'-2\ell}(1/\#K + 1/\#K^{\ell'-\ell})$ solutions. Consequently $p < 1/\#K + 1/\#K^{\ell'-\ell}$ as claimed.

Case 4: $\ell' < \ell$. Exchanging ℓ, m with ℓ', m' produces Case 2 or Case 3. □

Part III

Elliptic-Curve Cryptography

7 The simplest protocol for oblivious transfer

Oblivious Transfer (OT) is a cryptographic primitive defined as follows: in its simplest flavour, 1-out-of-2 OT, a sender has two input messages M_0 and M_1 and a receiver has a choice bit c. At the end of the protocol the receiver is supposed to learn the message M_c and nothing else, while the sender is supposed to learn nothing. Perhaps surprisingly, this extremely simple primitive is sufficient to implement any cryptographic task [Kil88]. OT can also be used to implement most advanced cryptographic tasks, such as secure two- and multi-party computation (e.g., the millionaire's problem) in an efficient way [NNO+12; BLN+15].

Given the importance of OT, and the fact that most OT applications require a very large number of OTs, it is crucial to construct OT protocols which are at the same time efficient and secure against realistic adversaries.

A novel OT protocol. In this chapter we present a novel and extremely *simple*, *efficient* and *secure* OT protocol. The protocol is a simple tweak of the celebrated Diffie-Hellman (DH) key exchange protocol. Given a group $\mathbb{G}$ and a generator g, the DH protocol allows two players Alice and Bob to agree on a key as follows: Alice samples a random a, computes $A = g^a$ and sends A to Bob. Symmetrically Bob samples a random b, computes $B = g^b$ and sends B to Alice. Now both parties can compute $g^{ab} = A^b = B^a$ from which they can derive a key k. The key observation is now that Alice can also derive a different key from the value $(B/A)^a = g^{ab-a^2}$, and that Bob cannot compute this group element (assuming that the computational DH problem is hard).

We can now turn this into an OT protocol by letting Alice play the role of the sender and Bob the role of the receiver (with choice bit c) as shown in Figure 7.1. The

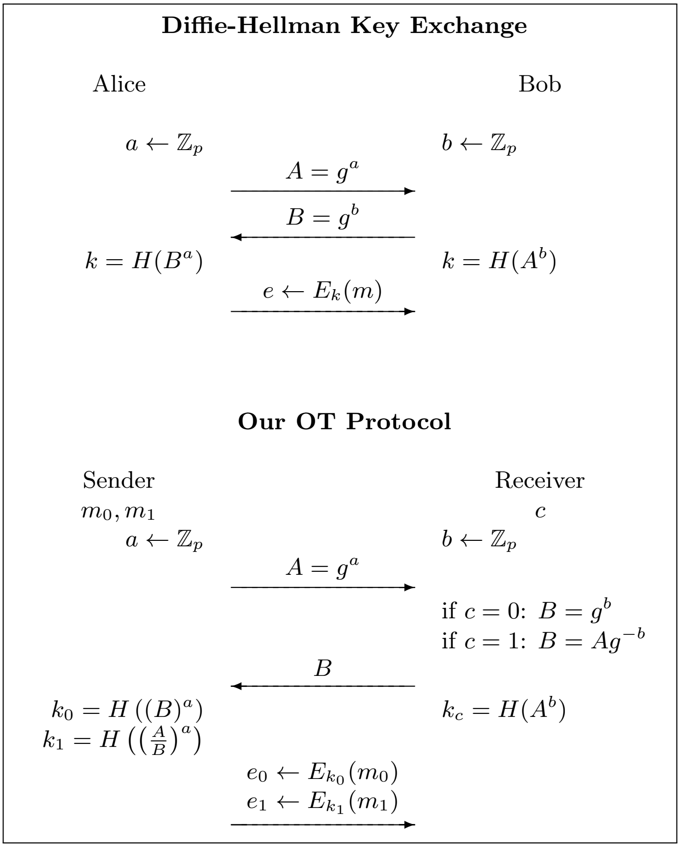
Diffie-Hellman Key Exchange
Alice
Bob
$a \leftarrow \mathbb{Z}_p$
$b \leftarrow \mathbb{Z}_p$
$A = g^a$
$B = g^b$
$k = H(B^a)$
$k = H(A^b)$
$e \leftarrow E_k(m)$
Our OT Protocol
Sender
Receiver
m_0, m_1
c
$a \leftarrow \mathbb{Z}_p$
$b \leftarrow \mathbb{Z}_p$
$A = g^a$
if $c = 0$: $B = g^b$
if $c = 1$: $B = Ag^{-b}$
B
$k_0 = H\left((B)^a\right)$
$k_1 = H\left(\left(\frac{A}{B}\right)^a\right)$
$k_c = H(A^b)$
$e_0 \leftarrow E_{k_0}(m_0)$
$e_1 \leftarrow E_{k_1}(m_1)$

Figure 7.1: Our protocol in a nutshell

first message (from Alice to Bob) is left unchanged (and can be reused over multiple instances of the protocol) but now Bob computes B as a function of his choice bit c: if $c = 0$ Bob computes $B = g^b$ and if $c = 1$ Bob computes $B = Ag^b$. At this point Alice derives two keys k_0, k_1 from $(B)^a$ and $(B/A)^a$ respectively. It is easy to check that Bob can derive the key k_c corresponding to his choice bit from A^b, but cannot compute the other one. This can be seen as a *random OT* i.e., an OT where the sender has no input but instead receives two random messages from the protocol, which can be used later to encrypt his inputs.

We show that combining the above random OT protocol with the right symmetric encryption scheme (e.g., a *robust encryption scheme* [ABN10; FLP$^+$13]) achieves security in a strong, simulation based sense and in particular we prove UC-security against active and adaptive corruptions in the random oracle model.

A secure and efficient implementation. We report on an efficient and secure implementation of the 1-out-of-2 random OT protocol: Our choice for the group is a twisted Edwards curve that has been used by Bernstein, Duif, Lange, Schwabe and Yang for building the Ed25519 signature scheme [BDL+11]. The security of the curve comes from the fact that it is birationally equivalent to Bernstein's Montgomery curve Curve25519 [Ber06] where ECDLP is believed to be hard: Bernstein and Lange's SafeCurves website [BL15a] reports cost of $2^{125.8}$ for solving ECDLP on Curve25519 using the *rho method*. The speed comes from the complete formulas for twisted Edwards curves proposed by Hisil, Wong, Carter, and Dawson in [HWC+08].

We first modify the code in [BDL+11] and build a fast implementation for a single OT. Later we build a vectorized implementation that runs OTs in batches. A comparison with the state of the art shows that our vectorized implementation is at least an order of magnitude faster than previous work (we compare in particular with the implementation reported by Asharov, Lindell, Schneider and Zohner in [ALS+13]) on recent Intel microarchitectures. Furthermore, we take great care to make sure that our implementation is secure against both passive attacks (our software is *immune to timing attacks*, since the implementation is *constant-time*) and active attacks (by designing an appropriate encoding of group elements, which can be efficiently verified and computed on). Our code can be downloaded from `http://orlandi.dk/simpleOT`.

Related works. OT owes its name to Rabin [Rab81] (a similar concept was introduced earlier by Wiesner [Wie83] under the name of "conjugate coding"). There are different flavours of OT, and in this chapter we focus on the most common and useful flavour, namely $\binom{n}{1}$-OT, which was first introduced in [EGL85]. Many efficient protocols for OT have been proposed over the years. Some of the protocols which are most similar to ours are those of Bellare-Micali [BM89] and Naor-Pinkas[NP01]: those protocols are (slightly) less efficient than ours and, most importantly, are not known to achieve full simulation based security. More recent OT protocols such as [HL10; DNO08; PVW08] focus on achieving a strong level of security in concurrent settings[1] without relying on the random oracle model. Unfortunately this makes these protocols more cumbersome for practical applications: even the most efficient of these protocols i.e., the protocol of Peikert, Vaikuntanathan, and Waters [PVW08] requires 11 exponentiations for a single $\binom{2}{1}$-OT and a common random string (which must be generated by some trusted source of randomness at the beginning of the protocol). In comparison our protocol uses fewer exponentiations (e.g., 5 for $\binom{2}{1}$-OT), generalizes to $\binom{n}{1}$-OT and does not require any (hard to implement in practice) setup assumptions.

OT extension. While OT provably requires "public-key" type of assumptions [IR89] (such as factoring, discrete log, etc.), OT can be "extended" [Bea96] in the sense that it is enough to generate few "seed" OTs based on public-key cryptography which can then be extended to any number of OTs using symmetric-key primitives only (PRG, hash functions, etc.). This can be seen as the OT equivalent of *hybrid encryption* (where one encrypts a large amount of data using symmetric-key cryptography, and then encapsulates the symmetric-key using a public-key cryptosystem). OT extension can be performed very efficiently both against passive [IKN+03; ALS+13] and active [Nie07; NNO+12; Lar14; ALS+15; KOS15] adversaries. Still, to bootstrap OT

[1] I.e., UC security [Can01], which is impossible to achieve without some kind of trusted setup assumptions [CF01].

extension we need a secure and efficient OT protocol for the seed OTs (as much as we need secure and efficient public-key encryption schemes to bootstrap hybrid encryption): The OT extension of [ALS$^+$15] reports that it takes time $(7 \cdot 10^5 + 1.3m)\mu s$ to perform m OTs, where the fixed term comes from running 190 base OTs. Using our protocol as the base OT in [ALS$^+$15] would reduce the initial cost to approximately $190 \cdot 114 \approx 2 \cdot 10^4 \mu s$ [Sch15], which leads to a significant overall improvement (e.g., a factor 10 for up to $4 \cdot 10^4$ OTs and a factor 2 for up to $5 \cdot 10^5$ OTs).

7.1 The protocol

7.1.1 Notation

If S is a set $s \leftarrow S$ is a random element sampled from S. We work over an additive group $(\mathbb{G}, B, p, +)$ of prime order p (with $\log(p) > \kappa$) generated by B (the base point), and we use the additive notation for the group since we later implement our protocol using elliptic curves. Given the representation of some group element P we assume it is possible to efficiently verify if $P \in \mathbb{G}$. We use $[n]$ as a shortcut for $\{0, 1, \ldots, n-1\}$.

7.1.2 Building blocks

We use a hash-function $H : (\mathbb{G} \times \mathbb{G}) \times \mathbb{G} \to \{0,1\}^\kappa$ as a key-derivation function to extract a κ bit key from a group element, and the first two inputs are used to seed the function.[2] We model H as a random oracle when arguing about the security of our protocol.

7.1.3 The ideal functionality

We want to implement m $\binom{n}{1}$-OT's for ℓ-bit messages with κ-bit security between a sender $\mathcal{S}$ and a receiver $\mathcal{R}$. We define a functionality $\mathcal{F}_{OT}^{-}(n, m, \ell)$ as follows:

Honest Use: the functionality receives a vector of indices $(c^1, \ldots, c^m) \in [n]^m$ from the receiver $\mathcal{R}$ and m vectors of message $\{(M_0^i, \ldots, M_{n-1}^i)\}_{i \in [m]}$ from the sender $\mathcal{S}$ where for all $i, j : M_i^j \in \{0,1\}^\ell$. The functionality outputs a vector of ℓ-bit strings $(z^1, \ldots, z^n)$ to the receiver $\mathcal{R}$, such that for all $i \in [m]$, $z^i = M_{c^i}^i$.

Dishonest Use: We weaken the functionality (hence the minus in the name) in the following way: a corrupted receiver $\mathcal{R}^*$ can input the choice values in an adaptive fashion i.e., the ideal adversary can input the choice indices c^i one by one and learn the message z^i before choosing the next index.

Note that when $m = 1$ the weakening has no effect. We choose to describe the protocol for m OTs in parallel since we can do this more efficiently than simply repeating m times the protocol for a single OT.

[2] Standard hash functions do not take group elements as inputs, and in later sections we will give explicit encodings of group elements into bitstrings.

7.1.4 Random OT

We split the presentation in two parts: first, we describe and analyze a protocol for *random OT* where the sender outputs n random keys and the receiver only learns one of them; then, we describe how to combine this protocol with an appropriate encryption scheme to complete the OT. We are now ready to describe our novel *random OT* protocol:

Setup: (only once, independent of m):

1. $\mathcal{S}$ samples $y \leftarrow \mathbb{Z}_p$ and computes $S = yB$ and $T = yS$;
2. $\mathcal{S}$ sends S to $\mathcal{R}$, who aborts if $S \notin \mathbb{G}$;

Choose: (in parallel for all $i \in [m]$)

1. $\mathcal{R}$ (with input $c^i \in [n]$) samples $x^i \leftarrow \mathbb{Z}_p$ and computes
$$R^i = c^i S + x^i B$$
2. $\mathcal{R}$ sends R^i to $\mathcal{S}$, who aborts if $R^i \notin \mathbb{G}$;

Key Derivation: (in parallel for all $i \in [m]$)

1. For all $j \in [n]$, $\mathcal{S}$ computes
$$k_j^i = H_{(S,R^i)}(yR^i - jT)$$
2. $\mathcal{R}$ computes
$$k_R^i = H_{(S,R^i)}(x^i S)$$

7.1.5 Basic properties

The key k_j^i is computed by hashing $x^i yB + (c^i - j)T$ and therefore at the end of the protocol $k_R^i = k_{c^i}^i$ if both parties are honest. It is also easy to see that:

Lemma 7.1.1. *No (computationally unbounded) $\mathcal{S}^*$ on input R^i can guess c^i with probability greater than $1/n$.*

Lemma 7.1.2. *No (computationally bounded) $\mathcal{R}^*$ can output any two keys $k_{j_0}^i$ and $k_{j_1}^i$ with $j_0 \neq j_1 \in [n]$ if the computational Diffie-Hellman problem is hard in $\mathbb{G}$.*

7.1.6 How to use the protocol and UC Security

We now show how to combine our random OT protocol with an appropriate encryption scheme to achieve UC security.

7.1.7 Motivation

Lemma 7.1.1 and 7.1.2 only state that some form of "privacy" holds for both the sender and the receiver. However, since OT is mostly used as a building block into more complex protocols, it is important to understand to which extent our protocol offers security when composed arbitrarily with itself or other protocols: Simulation based security is the minimal requirement which enables to argue that a given protocol is secure when composed with other protocols. Without simulation based security, it is not even possible to argue that a protocol is secure if it is executed twice in a sequential way! (See e.g., [DNO08] for a concrete counterexample for OT). The UC theorem [Can01] allows us to say that if a protocol satisfies the UC definition of security, then that protocol will be secure even when arbitrarily composed with other protocols. Among other things, to show that a protocol is UC secure one needs to show that a simulator can *extract* the input of a corrupted party: intuitively, this is a guarantee that the party *knows* its input, and its not reusing/modifying messages received in other protocols (aka malleability attack).

7.1.8 From random OT to standard OT

We start by adding a transfer phase to the protocol, where the sender sends the encryption of his messages to the receiver:

Transfer: (in parallel for all $i \in [m]$)

1. For all $j \in [n]$, $\mathcal{S}$ computes $e^i_j \leftarrow E(k^i_j, M^i_j)$
2. $\mathcal{S}$ sends $(e^i_0, \ldots, e^i_{n-1})$ to $\mathcal{R}$;

Retrieve: (in parallel for all $i \in [m]$)

1. $\mathcal{R}$ computes and outputs $z^i = D(k^i, e^i_{c^i})$.

7.1.9 The encryption scheme

The protocol uses a symmetric encryption scheme (E, D). We call $\mathcal{K}, \mathcal{M}, \mathcal{C}$ the key space, message space and ciphertext space respectively and κ the security parameter. We allow the decryption algorithm to output a special symbol $\perp$ to indicate an invalid ciphertext. We need the encryption scheme to satisfy the following properties:

Definition We say a symmetric encryption scheme (E, D) is *non-committing* if there exist PPT algorithms $\mathcal{S}_1, \mathcal{S}_2$ such that $\forall M \in \mathcal{M}$ (e', k') and (e, k) are computationally indistinguishable where $e' \leftarrow \mathcal{S}_1(1^\kappa)$, $k' \leftarrow \mathcal{S}_2(e', M)$, $k \leftarrow \mathcal{K}$ and $e \leftarrow E(k, M)$ ($\mathcal{S}_1, \mathcal{S}_2$ are allowed to share a state).

The definition says that it is possible for a simulator to come up with a ciphertext e which can later be "explained" as an encryption of any message, in such a way that the joint distribution of the ciphertext and the key in this simulated experiment is indistinguishable from the normal use of the encryption scheme, where a key is first sampled and then an encryption of M is generated.

Definition Let S be a set of random keys from $\mathcal{K}$ and $V_{S,e} \subseteq S$ the subset of valid keys for a given ciphertext e i.e., the keys in S such that $D(k, e) \neq \perp$.

We say (E, D) satisfies *robustness* if for all ciphertexts $e \leftarrow \mathcal{A}(1^\kappa, S)$ adversarially generated by a PPT $\mathcal{A}$, $|V_{S,e}| \leq 1$ except with negligible probability.

The definition says that it should be hard for an adversary to generate a ciphertext which can be decrypted to more than one valid ciphertext using any polynomial number of randomly generated keys (even for adversaries who see those keys before generating the ciphertext).

7.1.10 A concrete example

We give a concrete example of a very simple scheme which satisfies Definition 7.1.9 and 7.1.9: let $\mathcal{M} = \{0,1\}^\ell$ and $\mathcal{K} = \mathcal{C} = \{0,1\}^{\ell+\kappa}$. The encryption algorithm $E(k, m)$ parses k as (α, β) and $e = (m \oplus \alpha, \beta)$. The decryption algorithm $D(k, e)$ parses $k = (\alpha, \beta)$ and $e = (e_1, e_2)$ and outputs $\perp$ if $e_2 \neq \beta$ or outputs $m = e_1 \oplus \alpha$ otherwise. It can be shown that:

Lemma 7.1.3. *The scheme (E, D) defined above satisfies Definition 7.1.9 and 7.1.9.*

7.1.11 Simulation based security (UC)

We can finally argue UC security of our protocol.[3] The main ideas behind the proof are: it is possible to extract the choice value by checking whether a corrupted receiver queries the random oracle on points of the form $yR^i - cT$ for some c, since no adversary can query on points of this form for more than one c (without breaking the CDH assumption) and the *non-committing* property of (E, D) allows us to complete a successful simulation even if the corrupted receiver queries the oracle *after* he receives the ciphertexts; it is also possible to extract the sender messages by decrypting the ciphertexts with every key which the receiver got from the random oracle and Definition 7.1.9) allows us to conclude that except with negligible probability D returns $\perp$ for all keys different from the correct one.

Theorem 7.1.4. *The above protocol securely implements the functionality $\mathcal{F}_{OT}^{-}(n, m, \ell)$ under the following conditions:*

Corruption Model: *any active, adaptive corruption;*

Hybrid Functionalities: *we model H as a random oracle and we assume an authenticated channel (but not confidential) between the parties;*

Computational Assumptions: *we assume that the symmetric encryption scheme (E, D) satisfies Definition 7.1.9 and 7.1.9 and the computational Diffie-Hellman problem is hard in $\mathbb{G}$.*

[3] This subsection assumes that the reader is familiar with standard security definitions and proofs for two-party computation protocols such as those presented in [HL10].

7.1.12 Non-malleability in practice

Clearly, a proof that $a \to b$ only says that b is true when a is true, and since cryptographic security models (a) are not always a good approximation of the real world, we discuss some of these discrepancies here and therefore to which extent our protocol achieves security in practice (b), with particular focus on malleability attacks.

When instantiating our protocol we must replace the random oracle with a hash function: UC proofs crucially rely on the fact that the oracle is *local* to the protocol i.e., it can be only queried by the protocol participants, and different instances of the protocol run with different random oracles: clearly, there is no such a thing in the real world. To approximate the model, one can "localize" the random oracle by prepending the parties *id*'s and the session *id* to the hash function. We argue here that our choice of using the transcript of the protocol (S, R^i) as salt for the hash function helps in making sure that the oracle is *local* to the protocol, and helps against malleability attacks in cases where the parties' and session *id*'s are unavailable. Consider the following man-in-the middle attack, where an adversary $\mathcal{A}$ plays two copies of the $\binom{n}{1}$-OT, one as the sender with $\mathcal{R}$ and one as the receiver with $\mathcal{S}$. Here is how the attack works: 1) $\mathcal{A}$ receives S from $\mathcal{S}$ and forwards it to $\mathcal{R}$; 2) Then the adversary receives R from $\mathcal{R}$ and sends $R' = S + R$ to $\mathcal{S}$; 3) Finally $\mathcal{A}$ receives the $\{e_i\}_{i \in [n]}$ from $\mathcal{S}$ and sets $e'_i = e_{(i-1 \bmod n)}$ to $\mathcal{R}$. It is easy to see that if the same hash function is used to instantiate the random oracle in the two protocols (and if $c \neq 0$), then the honest receiver outputs $z = M_{c+1}$, which is clearly a breach of security (i.e., this attack could not be run if the protocols are replaced with OT functionalities).

The previous can be seen as a malleability attack on the choice bit. An adversary can also try a malleability attack on the sender messages by forwarding $(S', R') = (S, R)$ but then manipulating the e_i's into ciphertexts e'_i which decrypt to related messages. In the $\binom{2}{1}$-OT, these attacks can be mitigated by using *authenticated encryption* for (E, D) (which also satisfies *robustness* as in Definition 7.1.9). Now an adversary who changes both ciphertext is equivalent to an ideal adversary using input $(\perp, \perp)$, while an adversary who only changes one ciphertext, say e_c, is equivalent to an adversary which uses input bit $1 - c$ on the left and inputs $(m_{1-c}, \perp)$ on the right. Unfortunately for $\binom{n}{1}$-OT (with $n > 2$) this does not work. For instance, an adversary who corrupts only 1 out of m ciphertext cannot be simulated having access to ideal functionalities.

Finally we note that no practical instantiation of the encryption scheme leads to a *non-committing* encryption scheme (as required in Definition 7.1.9), but we conjecture that this an artificial requirement and does not lead to any concrete vulnerabilities.

7.2 The random OT protocol in practice

This section describes how the random OT protocol can be realized in practice. In particular, this section focuses on describing how group elements are represented as bitstrings, i.e., the *encodings*. In the abstract description of the random OT protocol, the sender and the receiver transmit and compute on "group elements", but clearly any implementation of the protocol transmits and computes on bitstrings. We describe how the encodings are designed to achieve efficiency (both for communication and computation) and security (particularly against a malicious party who might try to send malformed encodings).

7.2.1 The group

The group $\mathbb{G}$ we choose for the protocol is a subset of $\bar{\mathbb{G}}$; $\bar{\mathbb{G}}$ is defined by the set of points on the twisted Edwards curve

$$\{(x, y) \in \mathbb{F}_{2^{255}-19} \times \mathbb{F}_{2^{255}-19} : -x^2 + y^2 = 1 + dx^2y^2\}$$

and the twisted Edwards addition law

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + x_2y_1}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 + x_1x_2}{1 - dx_1x_2y_1y_2}\right)$$

introduced by Bernstein, Birkner, Joye, Lange, and Peters in [BBJ$^+$08]. The constant d and the generator B can be found in [BDL$^+$11]. The two groups $\bar{\mathbb{G}}$ and $\mathbb{G}$ are isomorphic respectively to $\mathbb{Z}_p \times \mathbb{Z}_8$ and $\mathbb{Z}_p$ with

$$p = 2^{252} + 27742317777372353535851937790883648493.$$

7.2.2 Encoding of group element

An *encoding* $\mathcal{E}$ for a group $\mathbb{G}_0$ is a way of representing group elements as fixed-length bitstrings. We write $\mathcal{E}(P)$ for a bitstring which represents $P \in \mathbb{G}_0$. Note that there can be multiple bitstrings that represent P; if there is only one bitstring for each group element, $\mathcal{E}$ is said to be *deterministic* ($\mathcal{E}$ is said to be *non-deterministic* otherwise[4]). Also note that some bitstrings (of the fixed length) might not represent any group element; we write $\mathcal{E}(\mathbb{G}_1)$ for the set of bitstrings which represent some element in $\mathbb{G}_1 \subseteq \mathbb{G}_0$. $\mathcal{E}$ is said to be *verifiable* if there exists an efficient algorithm that, given a bitstring as input, outputs whether it is in $\mathcal{E}(\mathbb{G}_0)$ or not.

7.2.3 The encoding $\mathcal{E}_X$ for group operations

The non-deterministic encoding $\mathcal{E}_X$ for $\bar{\mathbb{G}}$, which is based on the *extended coordinates* in [HWC$^+$08], represents each point using the tuple $(X : Y : Z : T)$ with $XY = ZT$, representing $x = X/Z$ and $y = Y/Z$. We use $\mathcal{E}_X$ whenever we need to perform group operations since given $\mathcal{E}_X(P), \mathcal{E}_X(Q)$ where $P, Q \in \bar{\mathbb{G}}$, it is efficient to compute $\mathcal{E}_X(P+P)$, $\mathcal{E}_X(P+Q)$, and $\mathcal{E}_X(P-Q)$. In particular, given an integer scalar $r \in \mathbb{Z}_p$ it is efficient to compute $\mathcal{E}_X(rB)$, and given r and $\mathcal{E}_X(P)$ it is efficient to compute $\mathcal{E}_X(rP)$.

7.2.4 The encoding $\mathcal{E}_0$ and related encodings

The deterministic encoding $\mathcal{E}_0$ for $\bar{\mathbb{G}}$ represents each group element as a 256-bit bitstring: the natural 255-bit encoding of y followed by a sign bit which depends only

[4] We stress that non-deterministic in this context does not mean that the encoding involves any randomness.

on x. The way to recover the full value x is described in [BDL$^+$11, Section 5], and group membership can be verified efficiently by checking whether $x^2(y^2-1) = dy^2+1$ holds; therefore $\mathcal{E}_0$ is verifiable. See [BDL$^+$11] for more details of $\mathcal{E}_0$.

For the following discussions, we define deterministic encodings $\mathcal{E}_1$ and $\mathcal{E}_2$ for $\mathbb{G}$ as

$$\mathcal{E}_1(P) = \mathcal{E}_0(8P), \mathcal{E}_2(P) = \mathcal{E}_0(64P), P \in \mathbb{G}.$$

We also define non-deterministic encodings $\mathcal{E}^{(0)}$ and $\mathcal{E}^{(1)}$ for $\mathbb{G}$ as

$$\mathcal{E}^{(0)}(P) = \mathcal{E}_0(P+t), \mathcal{E}^{(1)}(P) = \mathcal{E}_0(8P+t'), P \in \mathbb{G},$$

where t, t' can be any 8-torsion point. Note that each element in $\mathbb{G}$ has exactly 8 representations under $\mathcal{E}^{(0)}$ and $\mathcal{E}^{(1)}$.

7.2.5 Point compression/decompression

It is efficient to convert from $\mathcal{E}_X(P)$ to $\mathcal{E}_0(P)$ and back; since $\mathcal{E}_0$ represents points as much shorter bitstrings, these operations are called *point compression* and *point decompression*, respectively. Roughly speaking, point compression outputs $y = Y/Z$ along with the sign bit of $x = X/Z$, and point decompression first recovers x and then outputs $X = x, Y = y, Z = 1, T = xy$. We always check for group membership during point decompression.

We use $\mathcal{E}_0$ for data transmission: the parties send bitstrings in $\mathcal{E}_0(\bar{\mathbb{G}})$ and expect to receive bitstrings in $\mathcal{E}_0(\bar{\mathbb{G}})$. This means a computed point encoded by $\mathcal{E}_X$ has to be compressed before it is sent, and a received bitstring has to be decompressed for subsequent group operations. Sending compressed points helps to reduce the communication complexity: the parties only need to transfer $32 + 32m$ bytes in total.

7.2.6 Secure data transmission

At the beginning of the protocol $\mathcal{S}$ computes and sends $\mathcal{E}_0(S)$. In the ideal case, $\mathcal{R}$ should receive a bitstring in $\mathcal{E}_0(\mathbb{G})$ which he interprets as $\mathcal{E}_0(S)$. However, an attacker (a corrupted $\mathcal{S}^*$ or a man-in-the-middle) can send $\mathcal{R}$ 1) a bitstring that is not in $\mathcal{E}_0(\bar{\mathbb{G}})$ or 2) a bitstring in $\mathcal{E}_0(\bar{\mathbb{G}} \setminus \mathbb{G})$. In the first case, $\mathcal{R}$ detects that the received bitstring is not valid during point decompression and ignores it. In the second case, $\mathcal{R}$ can check group membership by computing the pth multiple of the point, but a more efficient way is to use a new encoding $\mathcal{E}'$ such that each bitstrings in $\mathcal{E}_0(\bar{\mathbb{G}})$ represents a point in $\mathbb{G}$ under $\mathcal{E}'$. Therefore $\mathcal{R}$ considers the received bitstring as $\mathcal{E}^{(0)}(S) = \mathcal{E}_0(S+t)$, where t can be any 8-torsion point.

The encoding $\mathcal{E}^{(0)}$ (along with point decompression) makes sure that $\mathcal{R}$ receives bitstrings representing elements in $\mathbb{G}$. However, an attacker can derive c^i by exploiting the extra information given by a nonzero t: a naive $\mathcal{R}$ would compute and send $\mathcal{E}_0(c^i(S+t) + x^iB) = \mathcal{E}_0(c^it + R^i)$; now by testing whether the result is $\mathcal{E}_0(\mathbb{G})$ the attacker learns whether $c^i = 0$.

To get rid of the 8-torsion point, $\mathcal{R}$ can multiply received point by $8 \cdot (8^{-1} \bmod p)$, but a more efficient way is to just multiply by 8 and then operate on $\mathcal{E}_X(8S)$ and $\mathcal{E}_X(8x^iB)$ to obtain and send $\mathcal{E}_1(R^i) = \mathcal{E}_0(8R^i)$, i.e, the encoding switches to $\mathcal{E}_1$ for

$\mathcal{S}$			$\mathcal{R}$		
Output	Input	Operations	Output	Input	Operations
S	y	$y \cdot B$	$8S$	$\mathcal{E}^{(0)}(S)$	$8 \cdot \mathcal{D}(\mathcal{E}^{(0)}(S))$
$\mathcal{E}^{(0)}(S)$	S	$\mathcal{C}(S)$	$\mathcal{E}_1(S)$	$8S$	$\mathcal{C}(8S)$
$8S$	S	$8 \cdot S$			
$\mathcal{E}_1(S)$	$8S$	$\mathcal{C}(8S)$			
$64T$	$8y, 8S$	$8 \cdot (y \cdot 8S)$			
$64R^i$	$\mathcal{E}^{(1)}(R^i)$	$8 \cdot \mathcal{D}(\mathcal{E}^{(1)}(R^i)$	$8x^iB$	$8x^i$	$8x^i \cdot B$
$\mathcal{E}_2(R^i)$	$64R^i$	$\mathcal{C}(64R^i)$	$8x^iB+8S$	$8S, 8x^iB$	$8x^iB + 8S$
$64yR^i$	$y, 64R^i$	$y \cdot 64R^i$	$\mathcal{E}^{(1)}(R^i)$	$8R^i$	$\mathcal{C}(8R^i)$
$\mathcal{E}_2(yR^i)$	$64yR^i$	$\mathcal{C}(64yR^i)$	$\mathcal{E}_2(R^i)$	$8R^i$	$\mathcal{C}(8 \cdot 8R^i)$
$64(yR^i - T)$	$64T, 64yR^i$	$64yR^i - 64T$	$64x^iS$	$8x^i, 8S$	$8x^i \cdot 8S$
$\mathcal{E}_2(yR^i - T)$	$64(yR^i - T)$	$\mathcal{C}(64(yR^i - T))$	$\mathcal{E}_2(x^iS)$	$64x^iS$	$\mathcal{C}(64x^iS)$

Table 7.1: How the parties compute encodings of group elements: each row shows that the "Output" is computed given "Input" using the operations "Operations". The input might come from the output of a previous row, a received string (e.g., $\mathcal{E}^{(1)}(R^i)$), or a random scalar that the party generates (e.g., $8x^i$). The upper half of the table are the operations that does not depend on i, which means the operations are performed only once for the whole protocol. $\mathcal{E}_X$ is suppressed: group elements written without encoding are actually encoded by $\mathcal{E}_X$. $\mathcal{C}$ and $\mathcal{D}$ stand for point compression and point decompression respectively. Computation of the rth multiple of P is denoted as "$r \cdot P$". In particular, $8 \cdot P$ can be carried out with only 3 point doublings.

R^i. After this $\mathcal{S}$ works similarly as $\mathcal{R}$: to ensure that the received bitstring represents an element in $\mathbb{G}$, $\mathcal{S}$ interprets the bitstring as $\mathcal{E}^{(1)}(R^i) = \mathcal{E}_0(8R^i + t)$; to get rid of the 8-torsion point $\mathcal{S}$ also multiplies the received point by 8, and then $\mathcal{S}$ operates on $\mathcal{E}_X(64R^i)$ and $\mathcal{E}_X(64T)$ to obtain $\mathcal{E}_X(64(yR^i - jT))$.

7.2.7 Key derivation

The protocol computes $H_{S,R^i}(P)$ where P can be x^iS, yR^i, or $yR^i - jT$ for $j \in [n]$. This is implemented by hashing $\mathcal{E}_1(S) \| \mathcal{E}_2(R^i) \| \mathcal{E}_2(P)$ with Keccak [BDP+11] with 256-bit output. The choice of encodings is natural: $\mathcal{S}$ computes $\mathcal{E}_X(S)$, and $\mathcal{R}$ computes $\mathcal{E}_X(8S)$; since multiplication by 8 is much cheaper than multiplication by (8^{-1} mod p), we use $\mathcal{E}_1(S) = \mathcal{E}_0(8S)$ for hashing. For similar reasons we use $\mathcal{E}_2$ for R^i and P.

7.2.8 Actual operations

For completeness, we present in Table 7.1 a full overview of operations performed during the protocol for the case of 1 out of 2 OT (i.e., $n = 2$).

7.3 Field arithmetic

This section describes our implementation strategy for arithmetic operations in the field $\mathbb{F}_{2^{255}-19}$, which serve as low-level building blocks for operations on the curve.

instruction	latency	throughput	description
`vandpd`	1	1	bitwise and
`vorpd`	1	1	bitwise or
`vxorpd`	1	1 (4)	bitwise xor
`vaddpd`	3	1	4-way parallel double-precision floating-point additions
`vsubpd`	3	1	4-way parallel double-precision floating-point subtractions
`vmulpd`	5	1	4-way parallel double-precision floating-point multiplications

Table 7.2: 256-bit vector instructions used in our implementation. Note that `vxorpd` has throughput of 4 when it has only one source operand.

Field operations are decomposed into double-precision floating-point operations using our strategy. A straightforward way for implementation is then using double-precision floating-point instructions. However, a better way to utilize the 64 × 64 → 128-bit serial multiplier is to decompose field operations into integer instructions as [BDL$^+$11] does. The real reason we decide to use floating-point operations is that it allows us to use 256-bit vector instructions on the target microarchitectures, which are functionally equivalent to 4 double-precision floating-point instructions. The technique, which is called *vectorization*, makes our vectorized implementation achieve much higher throughtput than our non-vectorized implementation based on [BDL$^+$11].

7.3.1 Representation of field elements

Each field element $x \in \mathbb{F}_{2^{255}-19}$ is represented as 12 *limbs* $(x_0, x_1, \ldots, x_{11})$ such that $x = \sum x_i$ and $x_i/2^{\lceil 21.25i \rceil} \in \mathbb{Z}$. Each x_i is stored as a double-precision floating-point number. Field operations are then carried out by limb operations such as floating-point additions and multiplications.

When a field element gets initialized (e.g., when obtained from a table lookup), each x_i uses no more than 21 bits of the 53-bit mantissa. However, after a series of limb operations, the number of bits x_i takes can grow. It is thus necessary to reduce the number of bits (in the mantissa) with carries before any precision is lost; see below for more discussions.

7.3.2 Field arithmetic

Additions and subtractions of field elements are implemented in a straightforward way: simply adding/subtracting the corresponding limbs. This does increase the number of bits in the mantissa, but in our application it suffices to reduce bits only at the end of the multiplication function.

A field multiplication is divided into two steps. The first step is a schoolbook multiplication on the $2 \cdot 12$ input limbs, with reduction modulo $2^{255} - 19$ to bring the result back to 12 limbs. The schoolbook multiplication takes 132 floating-point additions, 144 floating-point multiplications, and a few more multiplications by constants to handle the reduction.

Let $(c_0, c_1, \ldots, c_{11})$ be the result after schoolbook multiplication. The second step is to perform carries to reduce number of bits in c_i. Carry from c_i to c_{i+1} (indices

		`h6sandy`	`h9ivy`
[Moo15]	Average cycles to compute a public key	61828	57612
[BDL$^+$11]	Average cycles to compute a shared secret	194036	182708
this work	Average cycles to generate a public key	61458	60853
	Average cycles to compute a shared secret	182169	180343

Table 7.3: DH speeds of our work and existing Curve25519 implementations.

work modulo 12), which we denote as $c_i \to c_{i+1}$, is performed with 4 floating-point operations: $c \leftarrow c_i + \alpha_i$; $c \leftarrow c - \alpha_i$; $c_i \leftarrow c_i - c$; $c_{i+1} \leftarrow c_{i+1} + c$. The idea is to use $\alpha_i = 3 \cdot 2^{k_i}$ where k_i is big enough so that the less significant part of c_i are discarded in $c_i + \alpha_i$, forcing c to contain only the more significant part of c_i. For $i = 11$, one extra multiplication is required to scale c by $19 \cdot 2^{-255}$ before it is added to c_0.

A straightforward way to reduce number of bits in all limbs is to use the carry chain $c_0 \to c_1 \to c_2 \to \cdots \to c_{11} \to c_0 \to c_1$. The problem with the straightforward carry chain is that there is not enough instruction level parallelism to hide the 3-cycle latencies (see discussion below). To hide the latencies we thus interleave the following 3 carry chains:

$$\begin{aligned} &c_0 \to c_1 \to c_2 \to c_3 \to c_4 \to c_5, \\ &c_4 \to c_5 \to c_6 \to c_7 \to c_8 \to c_9, \\ &c_8 \to c_9 \to c_{10} \to c_{11} \to c_0 \to c_1. \end{aligned}$$

In total the multiplication function takes 192 floating-point additions/subtractions and 156 floating-point multiplications.

When the input operands are the same, many limb products will repeat in the schoolbook multiplication; a field squaring is therefore cheaper than a field multiplication. In total the squaring function takes 126 floating-point additions/subtractions and 101 floating-point multiplications.

Field inversion is implemented as a fix sequence of field squarings and multiplications.

7.3.3 Vectorization

We decompose field operations into 64-bit floating-point and logical operations. The Intel Sandy Bridge and Ivy Bridge microarchitectures, as well as many recent microarchitectures, offer instructions that operate on 256-bit registers. Some of these instructions treat the registers as vectors of 4 double-precision floating-point numbers and perform 4 floating-point operations in parallel; there are also 256-bit logical instructions that can be viewed as 4 64-bit logical instructions. We thus use these instructions to run 4 scalar multiplications in parallel. Table 7.2 shows the instructions we use, along with their latencies and throughputs on the Sandy Bridge and Ivy Bridge given in Fog's well-known survey [Fog14].

	m	4	8	16	32	64	128	256	512	1024
this work	Running time of $\mathcal{S}$	548	381	321	279	265	257	246	237	228
	Running time of $\mathcal{R}$	472	366	279	229	205	200	193	184	177
[ALS$^+$13]	Running time of $\mathcal{S}$	17976	10235	6132	4358	3348	2877	2650	2528	2473
	Running time of $\mathcal{R}$	16968	9261	5188	3415	3382	2909	2656	2541	2462

Table 7.4: Timings for per OT in kilocycles. Multiplying the number of kilocycles by 0.5 one can obtain the running time (in μs) on our test architecture.

7.4 Implementation results

This section compares the speed of our implementation of $\binom{2}{1}$-OT (i.e., $n = 2$) with other similar implementations. We stress that our software is a constant-time one: timing attacks are avoided using the same high-level strategy as [BDL$^+$11].

To show that our speeds for curve operations are competitive, we modify the software to support the function of Diffie-Hellman key exchange and compare the results with existing Curve25519 implementations (our implementation performs scalar multiplications on the twisted Edwards curve, so it is not the same as Curve25519). The experiments are carried out on two machines on the eBACS site for publicly verifiable benchmarks [BL15b]: `h6sandy` (Sandy Bridge) and `h9ivy` (Ivy Bridge). Since our protocol can serve as the base OTs for an OT extension protocol, we also compare our speed with a base OT implementation presented in [ALS$^+$13], which is included in the Scapi multi-party computation library; the experiments are made on an Intel Core i7-3537U processor (Ivy Bridge) where each party runs on one core. Note that all experiments are performed with Turbo Boost disabled.

7.4.1 Comparing with Curve25519 implementations

Table 7.3 compares our work with existing Curve25519 implementations. “Cycles to generate a public key” indicates the time to generate the public key given a secret key; the Curve25519 implementation is the implementation by Andrew Moon [Moo15]. “Cycles to compute a shared secret” indicates the time to generate the shared secret, given a secret key and a public key; the Curve25519 implementation is from [BDL$^+$11]. Note that since our software runs 4 scalar multiplications in parallel, the numbers in the table are the time for generating 4 public keys or 4 shared secrets divided by 4. In other words, our implementation is optimized for *througput* instead of *latency.*

7.4.2 Comparing with Scapi

Table 7.4 shows the timings of our implementation for the random OT protocol, along with the timings of a base-OT implementation presented in [ALS$^+$13]. The paper presents several base-OT implementations; the one we compare with is Miracl-based with “long-term security” using random oracle (cf. [ALS$^+$13, Section 6.1]). The implementation uses the NIST K-283 curve and SHA-1 for hashing, and it is not a constant-time implementation. It turns out that our work is an order of magnitude faster for $m \in \{4, 8, \ldots, 1024\}$.

7.4.3 Memory consumption

Our code for public-key generation uses a 284-KB table. For shared-secret computation the table size is 12 KB. For OTs, $\mathcal{S}$ uses a 12-KB table, while $\mathcal{R}$ is *allowed* to use a table of size up to 1344 KB which depends on the parameters given. The current code provides 4 copies of the precomputed points, one for each of the 4 scalar multiplcations, so it is possible to reduce the table sizes by a factor of 4 by broadcasting the precomputed points. Another reason that we have large tables is because of the representation for field elements: each limbs takes 8 bytes, so each field element already takes $12 \cdot 8 = 96$ bytes. The window sizes we use are the same as [BDL+11]. See [BDL+11] for issues related to table sizes.

8 Sandy2x: new Curve25519 speed records

In 2006, Bernstein proposed Curve25519, which uses a fast Montgomery curve for Diffie-Hellman (DH) key exchange. In 2011, Bernstein, Duif, Schwabe, Lange and Yang proposed the Ed25519 digital signature scheme, which uses a fast twisted Edwards curve that is birationally equivalent to the same Montgomery curve. Both schemes feature a conservative 128-bit security level, very small key sizes, and consistent fast speeds on various CPUs (cf. [BDL$^+$11], [BL13]), as well as microprocessors such as ARM ([BS12], [DHH$^+$15]), Cell ([CS09]), etc.

Curve25519 and Ed25519 have gained public acceptance and are used in many applications. The IANIX site [Bro] has lists for Curve25519 and Ed25519 deployment, which include the Tor anonymity network, the QUIC transport layer network protocol developed by Google, OpenSSH, and many more.

This chapter presents `Sandy2x`, a new software which sets new speed records for Curve25519 and Ed25519 on the Intel Sandy Bridge and Ivy Bridge microarchitectures. Previous softwares set speed records for these CPUs using the serial multiplier. `Sandy2x`, instead, uses of a vectorized multiplier. Our results show that previous elliptic-curve cryptography (ECC) papers using the serial multiplier might have made a suboptimal choice.

A part of our software (the code for Curve25519 shared-secret computation) has been submitted to the SUPERCOP benchmarking toolkit, but the speeds have not been included in the eBACS [BL13] site yet. We plan to submit the whole software soon for public use.

Serial multipliers versus vectorized multipliers. Prime field elements are usually represented as big integers in softwares. The integers are usually divided into several small chunks called *limbs*, so that field operations can be carried out as sequences of operations on limbs. Algorithms involving field arithmetic are usually bottlenecked by multiplications, which are composed of limb multiplications. On Intel CPUs, each core has a powerful $64 \times 64 \rightarrow 128$-bit serial multiplier, which is convenient for limb multiplications. There have been many ECC papers that use the serial multiplier for field arithmetic. For example, [BDL$^+$11] uses the serial multipliers on Nehalem/Westmere; [LS12c] uses the serial multipliers on Sandy Bridge; [CHS14] uses the serial multipliers on Ivy Bridge.

On some other chips, it is better to use a vectorized multiplier. The Cell Broadband Engine has 7 Synergistic Processor Units (SPUs) which are specialized for vectorized instructions; the primary processor has no chance to compete with them. ARM has a 2-way vectorized $32 \times 32 \rightarrow 64$-bit multiplier, which is clearly stronger than the $32 \times 32 \rightarrow 64$ serial multiplier. A few ECC papers exploit the vectorized multipliers, including [BS12] for ARM and [CS09] for Cell. In 2014, there is finally one effort for using a vectorized multiplier on Intel chips, namely [BCL$^+$14]. The paper uses vectorized multipliers to carry out hyperelliptic-curve cryptography (HECC) formulas that provide a natural 4-way parallelism. ECC formulas do not exhibit such nice internal parallelism, so vectorization is expected to induce much more overhead than HECC.

Our speed records rely on using a 2-way vectorized multipliers on Sandy Bridge and Ivy Bridge. The vectorized multiplier carries out only a pair of $32 \times 32 \rightarrow 64$-bit multiplication in one instruction, which does not seem to have any chance to compete with the $64 \times 64 \rightarrow 128$-bit serial multiplier, which is used to set speed records in previous Curve25519/Ed25519 implementations. In this chapter we investigate how serial multipliers and vectorized multipliers work (Section 8.1), and give arguments on why the vectorized multiplier can compete.

Our work is similar to [BCL$^+$14] in the sense that we both use vectorized multipliers on recent Intel microarchitectures. The difference is that our algorithm does not have very nice internal parallelism, especially for verification. Our work is also similar to [BS12] in the sense that the vectorized multipliers have the same input and output size. We stress that the low-level optimization required on ARM is different to Sandy/Ivy Bridge, and it is certainly harder to beat the serial multiplier on Sandy/Ivy Bridge.

Performance results. The performance results for our software are summarized in Table 8.1, along with the results for [BDL$^+$11] and [LM13]. [BDL$^+$11] is chosen because it holds the speed records on the eBACS site for publicly verifiable benchmarks [BL13]; [LM13] is chosen because it is the fastest constant-time public implementation for Ed25519 (and Curve25519 public-key generation) to our knowledge. The speeds of our software (as [BDL$^+$11] and [LM13]) are fully protected against simple timing attacks, cache-timing attacks, branch-prediction attacks, etc.: all load addresses, all store addresses, and all branch conditions are public.

For comparison, Longa reported $\approx 298\,000$ Sandy Bridge cycles for the "ECDHE" operation, which is essentially 1 public-key generation plus 1 secret-key computation, using Microsoft's 256-bit NUMS curve [BBC$^+$14]. OpenSSL 1.0.2, after heavy optimization work from Intel, compute a NIST P-256 scalar multiplication in

	SB cycles	IB cycles	table size	reference	implementation
Curve25519 public-key generation	54 346	52 169	30720 + 0	**(new)** `sandy2x`	
	61 828	57 612	24576 + 0	[LM13]	
	194 165	182 876	0 + 0	[BDL+11] CHES 2011	`amd64-51`
Curve25519 shared secret computation	159 128	156 995	0 + 0	**(new)** `sandy2x`	
	194 036	182 708	0 + 0	[BDL+11] CHES 2011	`amd64-51`
Ed25519 public-key generation	57 164	54 901	30720 + 0	**(new)** `sandy2x`	
	63 712	59 332	24576 + 0	[LM13]	
	64 015	61 099	30720 + 0	[BDL+11] CHES 2011	`amd64-51-30k`
Ed25519 sign	63 526	59 949	30720 + 0	**(new)** `sandy2x`	
	67 692	62 624	24576 + 0	[LM13]	
	72 444	67 284	30720 + 0	[BDL+11] CHES 2011	`amd64-51-30k`
Ed25519 verification	205 741	198 406	10240 + 1920	**(new)** `sandy2x`	
	227 628	204 376	5120 + 960	[LM13]	
	222 564	209 060	5120 + 960	[BDL+11] CHES 2011	`amd64-51-30k`

Table 8.1: Performance results for Curve25519 and Ed25519 of `sandy2x`, the CHES 2011 paper [BDL+11], and the implementation by Andrew Moon "floodyberry" [LM13]. All implementations are benchmarked on the Sandy Bridge machine "**`h6sandy`**" and the Ivy Bridge machine"**`h9ivy`**" (abbreviated as SB and IB in the table), of which the details can be found on the eBACS website [BL13]. Each cycle count listed is the measurement result of running the software on one CPU core, with Turbo Boost disabled. The table sizes (in bytes) are given in two parts: read-only memory size + writable memory size.

311 434 Sandy Bridge cycles or 277 994 Ivy Bridge cycles.

For Curve25519 public-key generation, [LM13] and our implementation gain much better results than [BDL+11] by performing the fixed-base scalar multiplications on the twisted Edwards curve used in Ed25519 instead of the Montgomery curve; see Section 8.2.2. Our implementation strategy for Ed25519 public-key generation and signing is the same as Curve25519 public-key generation. Also see Section 8.2.1 for Curve25519 shared-secret computation, and Section 8.3 for Ed25519 verification.

We also include the tables sizes of [BDL+11], [LM13] and `Sandy2x` in Table 8.1. Note that our current code uses the same window sizes as [BDL+11] and [LM13] but larger tables for Ed25519 verification. This is because we use a data format that is not compact but more convenient for vectorization. Also note that [BDL+11] has two implementations for Ed25519: `amd64-51-30k` and `amd64-64-24k`. The tables sizes for `amd64-64-24k` are 20% smaller than those of `amd64-51-30k`, but the speed records on eBACS are set by `amd64-51-30k`.

Other fast Diffie-Hellman and signature schemes. On the eBACS site [BL13] there are a few DH schemes that achieve fewer Sandy/Ivy Bridge cycles for shared-secret computation than our software:
`gls254prot` from [OLA+13b] uses a GLS curve over a binary field; `gls254` is a non-constant-time version of `gls254prot`; `kummer` from [BCL+14] is a HECC scheme; `kumfp127g` from [BCH+13] implements the same scheme as [BCL+14] but uses an obsolete approach to perform scalar multiplication on hyperelliptic curves as explained in [BCL+14].

GLS curves are patented, making them much less attractive for deployment, and papers such as [PQ12] and [Sem15] make binary-field ECC less confidence-inspiring.

There are algorithms that are better than the Rho method for high-genus curves; see, for example, [Thé03]. Compared to these schemes, Curve25519, using an elliptic curve over a prime field, seems to be a more conservative (and patent-free) choice for deployment.

The eBACS website also lists some signature schemes which achieve better signing and/or verification speeds than our work. Compared to these schemes, Ed25519 has the smallest public-key size (32 bytes), fast signing speed (superseded only by *multivariate* schemes with much larger key sizes), reasonably fast verification speed (can be much better if batched verification is considered, as shown in [BDL+11]), and a high security level (128-bit).

8.1 Arithmetic in $\mathbb{F}_{2^{255}-19}$

A radix-2^r representation represents an element f in a b-bit prime field as $(f_0, f_1, \ldots, f_{\lceil b/r \rceil - 1})$, such that

$$f = \sum_{i=0}^{\lceil b/r \rceil - 1} f_i 2^{\lceil ir \rceil}.$$

This is called a radix-2^r representation. Field arithmetic can then be carried out using operations on limbs; as a trivial example, a field addition can be carried out by adding corresponding limbs of the operands.

Since the choice of radix is often platform-dependent, several radices have been used in existing software implementations of Curve25519 and Ed25519. This section describes and compares the radix-2^{51} representation (used by [BDL+11]) with the radix-$2^{25.5}$ representation (used by [BS12] and this chapter), and explains how a small-radix implementation can beat a large-radix one on Sandy Bridge and Ivy Bridge, even though the vectorized multiplier seems to be slower. The radix-2^{64} representation by [BDL+11] appears to be slower than the radix-2^{51} representation for Curve25519 shared-secret computation, so only the latter is discussed in this section.

8.1.1 The radix-2^{51} representation

[BDL+11] represents an integer f modulo $2^{255} - 19$ as

$$f_0 + 2^{51} f_1 + 2^{102} f_2 + 2^{153} f_3 + 2^{204} f_4$$

As the result, the product of $f_0 + 2^{51} f_1 + 2^{102} f_2 + 2^{153} f_3 + 2^{204} f_4$ and $g_0 + 2^{51} g_1 + 2^{102} g_2 + 2^{153} g_3 + 2^{204} g_4$ is $h_0 + 2^{51} h_1 + 2^{102} h_2 + 2^{153} h_3 + 2^{204} h_4$ modulo $2^{255} - 19$ where

$$\begin{aligned}
h_0 &= f_0 g_0 + 19 f_1 g_4 + 19 f_2 g_3 + 19 f_3 g_2 + 19 f_4 g_1,\\
h_1 &= f_0 g_1 + f_1 g_0 + 19 f_2 g_4 + 19 f_3 g_3 + 19 f_4 g_2,\\
h_2 &= f_0 g_2 + f_1 g_1 + f_2 g_0 + 19 f_3 g_4 + 19 f_4 g_3,\\
h_3 &= f_0 g_3 + f_1 g_2 + f_2 g_1 + f_3 g_0 + 19 f_4 g_4,\\
h_4 &= f_0 g_4 + f_1 g_3 + f_2 g_2 + f_3 g_1 + f_4 g_0.
\end{aligned}$$

One can replace g by f to derive similar equations for squaring.

The radix-2^{51} representation is designed to fit the 64 × 64 → 128-bit serial multiplier, which can be accessed using the `mul` instruction. The usage of the `mul` is as follows: given a 64-bit integer (either in memory or a register) as operand, the instruction computes the 128-bit product of the integer and `rax`, and stores the higher 64 bits of in `rdx` and lower 64 bits in `rax`.

The field multiplication function begins with computing $f_0g_0, f_0g_1, \ldots, f_0g_4$. For each g_j, f_0 is first loaded into `rax`, and then a `mul` instruction is used to compute the product; some `mov` instructions are required to move the `rdx` and `rax` to the registers where h_j is stored. Each monomial involving f_i where $i > 0$ also takes a `mul` instruction, and an addition (`add`) and an addition with carry (`adc`) are required to accumulate the result into h_k. Multiplications by 19 can be handled by the `imul` instruction. In total, it takes 25 `mul`, 4 `imul`, 20 `add`, and 20 `adc` instructions to compute $h_0, h_1, \ldots, h_4$ [1]. Note that some *carries* are required to bring the h_k back to around 51 bits. We denote such a radix-51 field multiplication including carries as $\mathbf{m}$; $\mathbf{m}^-$ represents $\mathbf{m}$ without carries.

8.1.2 The radix-$2^{25.5}$ representation

[BS12] represents an integer f modulo $2^{255} - 19$ as

$$f_0 + 2^{26}f_1 + 2^{51}f_2 + 2^{77}f_3 + 2^{102}f_4 + 2^{128}f_5 + 2^{153}f_6 + 2^{179}f_7 + 2^{204}f_8 + 2^{230}f_9.$$

As the result, the product of $f_0 + 2^{26}f_1 + 2^{51}f_2 + \cdots$ and $g_0 + 2^{26}g_1 + 2^{51}g_2 + \cdots$ is $h_0 + 2^{26}h_1 + 2^{51}h_2 + \cdots$ modulo $2^{255} - 19$ as shown in Figure 8.1

One can replace g by the f to derive similar equations for squaring.

The representation is designed to fit the vector multiplier on Cortex-A8, which performs a pair of 32 × 32 → 64-bit multiplications in one instruction. On Sandy Bridge and Ivy Bridge a similar vectorized multiplier can be accessed using the `vpmuludq` [2] instruction. The AT&T syntax of the `vpmuludq` instruction is as follows:

```
vpmuludq src2, src1, dest
```

where `src1` and `dest` are 128-bit registers, and `src2` can be either a 128-bit register or (the address of) an aligned 32-byte memory block. The instruction multiplies the lower 32 bits of the lower 64-bit words of `src1` and `src2`, multiplies the lower 32 bits of the higher 64-bit words of`src1` and `src2`, and stores the 64 bits products in 64-bit words of `dest`.

To compute $h = fg$ and $h' = f'g'$ at the same time, we follow the strategy of [BS12] but replace the vectorized addition and multiplication instructions by corresponding ones on Sandy/Ivy Bridge. Given $(f_0, f'_0), \ldots (f_9, f'_9)$ and $(g_0, g'_0), \ldots (g_9, g'_9)$, first prepare 9 vectors $(19g_1, 19g'_1), \ldots, (19g_9, 19g'_9)$ with 10 `vpmuludq` instructions and

[1] [BDL+11] uses one more `imul`; perhaps this is for reducing memory access.

[2] The starting 'v' indicate that the instruction is the VEX extension of the `pmuludq` instruction. The benefit of using `vpmuludq` is that it is a 3-operand instruction. In this chapter we show vector instructions in their VEX extension form, even though vector instructions are sometimes used without the VEX extension.

$$
\begin{aligned}
h_0 &= f_0g_0 + 38f_1g_9 + 19f_2g_8 + 38f_3g_7 + 19f_4\,g_6 + 38f_5g_5 + 19f_6g_4 + 38f_7g_3 \\
&\qquad + 19f_8g_2 + 38f_9g_1, \\
h_1 &= f_0g_1 + f_1g_0 + 19f_2g_9 + 19f_3g_8 + 19f_4g_7 + 19f_5g_6 + 19f_6g_5 + 19f_7g_4 \\
&\qquad + 19f_8g_3 + 19f_9g_2, \\
h_2 &= f_0g_2 + 2f_1g_1 + f_2g_0 + 38f_3g_9 + 19f_4g_8 + 38f_5g_7 + 19f_6g_6 + 38f_7g_5 \\
&\qquad + 19f_8g_4 + 38f_9g_3, \\
h_3 &= f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0 + 19f_4g_9 + 19f_5g_8 + 19f_6g_7 + 19f_7g_6 \\
&\qquad + 19f_8g_5 + 19f_9g_4, \\
h_4 &= f_0g_4 + 2f_1g_3 + f_2g_2 + 2f_3g_1 + f_4g_0 + 38f_5g_9 + 19f_6g_8 + 38f_7g_7 \\
&\qquad + 19f_8g_6 + 38f_9g_5, \\
h_5 &= f_0g_5 + f_1g_4 + f_2g_3 + f_3g_2 + f_4g_1 + f_5g_0 + 19f_6g_9 + 19f_7g_8 \\
&\qquad + 19f_8g_7 + 19f_9g_6, \\
h_6 &= f_0g_6 + 2f_1g_5 + f_2g_4 + 2f_3g_3 + f_4g_2 + 2f_5g_1 + f_6g_0 + 38f_7g_9 \\
&\qquad + 19f_8g_8 + 38f_9g_7, \\
h_7 &= f_0g_7 + f_1g_6 + f_2g_5 + f_3g_4 + f_4g_3 + f_5g_2 + f_6g_1 + f_7g_0 \\
&\qquad + 19f_8g_9 + 19f_9g_8, \\
h_8 &= f_0g_8 + 2f_1g_7 + f_2g_6 + 2f_3g_5 + f_4g_4 + 2f_5g_3 + f_6g_2 + 2f_7g_1 \\
&\qquad + f_8g_0 + 38f_9g_9, \\
h_9 &= f_0g_9 + f_1g_8 + f_2g_7 + f_3g_6 + f_4g_5 + f_5g_4 + f_6g_3 + f_7g_2 \\
&\qquad + f_8g_1 + f_9g_0.
\end{aligned}
$$

Figure 8.1: Field multiplication in $\mathbb{F}_{2^{255}-19}$ using radix $2^{25.5}$.

$(2f_1, 2f_1')$, $(2f_3, 2f_3')$, $\ldots$, $(2f_9, 2f_9')$ with 5 vectorized addition instructions `vpaddq`. Note that the reason to use `vpaddq` instead of `vpmuludq` is to balance the loads of different execution units on the CPU core; see analysis in Section 8.1.3. Each $(f_0g_j, f_0'g_j')$ then takes 1 `vpmuludq`, while each $(f_ig_j, f_i'g_j')$ where $i > 0$ takes 1 `vpmuludq` and 1 `vpaddq`. In total, it takes 109 `vpmuludq` and 95 `vpaddq` to compute $(h_0, h_0'), (h_1, h_1'), \ldots, (h_9, h_9')$. We denote such a vector of two field multiplications as $\mathbf{M^2}$, including the carries that bring h_k (and also h_k') back to $26 - (k \bmod 2)$ bits; $\mathbf{M^{2-}}$ represents $\mathbf{M^2}$ without carries. Similarly, we use $\mathbf{S^2}$ and $\mathbf{S^{2-}}$ for squarings.

We perform a carry from h_k to h_{k+1} (the indices work modulo 10), which is denoted by $h_k \to h_{k+1}$, in 3 steps:

- Perform a logical right shift for the 64-bit words in h_k using a `vpsrlq` instruction. The shift amount is $26 - (k \bmod 2)$.
- Add the result of the first step into h_{k+1} using a `vpaddq` instruction.
- Mask out the most significant $38+(k \bmod 2)$ bits of h_k using a `vpand` instruction.

For $h_9 \to h_0$ the result of the shift has to be multiplied by 19 before being added to h_0. Note that the usage of `vpsrlq` suggests that we are using *unsigned* limbs; there is no vectorized arithmetic shift instruction on Sandy Bridge and Ivy Bridge.

instruction	port	throughput	latency
vpmuludq	0	1	5
vpaddq	either 1 or 5	2	1
vpsubq	either 1 or 5	2	1
mul	0 and 1	1	3
imul	1	1	3
add	either 0, 1, or 5	3	1
adc	either two of 0,1,5	1	2

Table 8.2: Instructions field arithmetic used in [BDL+11] and this chapter. The data is mainly based on the well-known survey by Fog [Fog14]. The survey does not specify the port utilization for `mul`, so we figure this out using the performance counter (accessed using `perf-stat`). Throughputs are per-cycle. Latencies are given in cycles.

To reduce number of bits in all of $h_0, h_1, \ldots, h_9$, the simplest way is to perform the carry chain

$$h_0 \to h_1 \to h_2 \to h_3 \to h_4 \to h_5 \to h_6 \to h_7 \to h_8 \to h_9 \to h_0 \to h_1.$$

The problem of the simple carry chain is that it suffers severely from the instruction latencies. To mitigate the problem, we instead interleave the 2 carry chains

$$h_0 \to h_1 \to h_2 \to h_3 \to h_4 \to h_5 \to h_6,$$

$$h_5 \to h_6 \to h_7 \to h_8 \to h_9 \to h_0 \to h_1.$$

It is not always the case that there are two multiplications that can be paired with each other in an elliptic-curve operation; sometimes there is a need to vectorize a field multiplication internally. We use a similar approach to [BS12] to compute $h_0, h_1, \ldots, h_9$ in this case; the difference is that we compute vectors (h_0, h_1), ..., (h_8, h_9) as result. The strategy for performing the expensive carries on $h_0, h_1, \ldots, h_9$ is the same as [BS12]. Such an internally-vectorized field multiplication is denoted as $\mathbf{M}$.

8.1.3 Why is smaller radix better?

$\mathbf{m}$ takes 29 multiplication instructions (`mul` and `imul`), while $\mathbf{M^2}$ takes $109/2 = 54.5$ multiplication instructions (`vpmuludq`) per field multiplication. How can our software, (which is based on $\mathbf{M^2}$) be faster than [BDL+11] (which is based on $\mathbf{m}$) using almost twice as many multiplication instructions?

On Intel microarchitechtures, an instruction is decoded and decomposed into some *micro-operations* (μops). Each μop is then stored in a pool, waiting to be executed by one of the *ports* (when the operands are ready). On each Sandy Bridge and Ivy Bridge core there are 6 ports. In particular, Port 0, 1, 5 are responsible for arithmetic. The remaining ports are responsible for memory access, which is beyond the scope of this chapter.

The arithmetic ports are not identical. For example, vpmuludq is decomposed into 1 μop, which is handled by Port 0 each cycle with latency 5. vpaddq is decomposed into 1 μop, which is handled by Port 1 or 5 each cycle with latency 1. Therefore, an $\mathbf{M^{2-}}$ would take at least 109 cycles. Our experiment shows that $\mathbf{M^{2-}}$ takes around 112 Sandy Bridge cycles, which translates to 56 cycles per multiplication.

The situation for $\mathbf{m}$ is more complicated: mul is decomposed into 2 μops, which are handled by Port 0 and 1 each cycle with latency 3. imul is decomposed into 1 μop, which is handled by Port 1 each cycle with latency 3. add is decomposed into 1 μop, which is handled by one of Port 0,1,5 each cycle with latency 1. adc is decomposed into 2 μops, which are handled by two of Port 0,1,5 each cycle with latency 2. In total it takes 25 mul, 4 imul, 20 add, and 20 adc, accounting for at least $(25\cdot 2+4+20+20\cdot 2)/3 = 38$ cycles. Our experiment shows that $\mathbf{m}^-$ takes 52 Sandy Bridge cycles. The mov instructions explain a few cycles out of the $52 - 38 = 14$ cycles. Also, the performance counter shows that the core fails to distribute μops equally over the ports.

Of course, by just looking at these cycle counts it seems that $\mathbf{M^2}$ is still a bit slower, but at least we have shown that the serial multiplier is not as powerful as it seems to be. Here are some more arguments in favor of $\mathbf{M^2}$:

- $\mathbf{m}$ spends more cycles on carries than $\mathbf{M^2}$ does: $\mathbf{m}$ takes 68 Sandy Bridge cycles, while $\mathbf{M^2}$ takes 69.5 Sandy Bridge cycles per multiplication.
- The algorithm built upon $\mathbf{M^2}$ might have additions/subtractions. Some speedup can be gained by interleaving the code; see Section 8.1.5.
- The computation might have some non-field-arithmetic part which can be improved using vector unit; see Section 8.2.2.

8.1.4 Importance of using a small constant

For the ease of reduction, the prime fields used in ECC and HECC are often a big power of 2 subtracted by a small constant c. It might seem that as long as c is not too big, the speed of field arithmetic would remain the same. However, in the following example, we show that using the slightly larger $c = 31$ ($2^{255} - 31$ is the large prime before $2^{255} - 19$) might already cause some overhead.

Consider two field elements f, g which are the results of two field multiplications. Because the limbs are reduced, the upper bound of f_0 would be close to 2^{26}, and the upper bound of f_1 would be close to 2^{25}, and so on; the same bounds apply for g. Now suppose we need to compute $(f-g)^2$, which is batched with another squaring to form an $\mathbf{S}^2$. To avoid possible underflow, we compute the limbs of $h = f - g$ as $h_i = (f_i + 2\cdot q_i) - g_i$ instead of $h_i = f_i - g_i$, where q_i is the corresponding limb of $2^{255} - 19$. As the result, the upper bound of h_6 is around $3 \cdot 2^{26}$. To perform the squaring $c \cdot h_6^2$ is required. When $c = 19$ we can simply multiply h_6 by 19 using 1 vpmuludq, and then multiply the product by h_6 using another vpmuludq. Unfortunately the same instructions do not work for $c = 31$, since $31 \cdot h_6$ can take more than 32 bits.

To overcome such problem, an easy solution is to use a smaller radix so that each (reduced) limb takes fewer bits. This method would increase number of limbs and thus increase number of vpmuludq required. A better solution is to delay the multiplication by c: instead of computing $31f_{i_1}g_{j_1} + 31f_{i_2}gj_2 + \cdots$ by first computing $31g_{j_1}, 31g_{j_2}, \ldots$, compute $f_{i_1}g_{j_1} + f_{i_2}g_{j_2} + \cdots$ and then multiply the sum by 31. The sum can take more than 32 bits (and vpmuludq takes only 32-bit inputs), so the multiplication by 31 cannot be handled by vpmuludq. Let $s = f_{i_1}g_{j_1} + f_{i_2}g_{j_2} + \cdots$, one way to handle the multiplication by 19 is to compute $32s$ with one shift instruction vpsllq and then compute $32s - s = 31s$ with one subtraction instruction vpsubq. This solution does not make Port 0 busier as vpsllq also takes only one cycle in Port 0 as vpmuludq, but it does make Port 1 and 5 busier (because of vpsubq), which can potentially increase the cost for $\mathbf{S^{2-}}$ by a few cycles.

It is easy to imagine for some c's the multiplication can not be handled in such a cheap way as 31. In addition, delaying multiplication cannot handle as many c's as using a smaller radix; as a trivial example, it does not work if $cf_{i_1}g_{j_1} + cf_{i_2}g_{j_2} + \cdots$ takes more than 64 bits. We note that the computation pattern in the example is actually a part of elliptic-curve operation (see lines 6–9 in Algorithm 1), meaning a bigger constant c actually can slow down elliptic-curve operations.

We comment that usage of a larger c has bigger impact on constrained devices. If c is too big for efficient vectorization, at least one can go for the $64 \times 64 \to 128$-bit serial multiplier, which can handle a wide range of c without increasing number of limbs. However, on ARM processors where the serial multiplier can only perform $32 \times 32 \to 64$-bit multiplications, even the serial multiplier would be sensitive to the size of c. For even smaller devices the situation is expected to be worse.

8.1.5 Instruction scheduling for vectorized field arithmetic

The fact that μops are stored in a pool before being handled by a port allows the CPU to achieve so called *out-of-order execution*: a μop can be executed before another μop which is from an earlier instruction. This feature is sometimes viewed as the CPU core being able to "look ahead" and execute a later instruction whose operands are ready. However, the ability of out-of-order execution is limited: the core is not able to look too far away. It is thus better to arrange the code so that each code block contains instructions for each port.

While Port 0 is quite busy in $\mathbf{M^2}$, Port 1 and 5 are often idle. In an elliptic-curve operation (see the following sections) an $\mathbf{M^2}$ is often preceded by a few field additions/subtractions. Since vpaddq and the vectorized subtraction instruction vpsubq can only be handled by either Port 1 and Port 5, we try to interleave the multiplication code with the addition/subtraction code to reduce the chance of having an idle port. Experiment results show that the optimization brings a small yet visible speedup. It seems more difficult for an algorithm built upon $\mathbf{m}$ to use the same optimization.

8.2 The Curve25519 elliptic-curve-Diffie-Hellman scheme

[Ber06] defines Curve25519 as a function that maps two 32-byte input strings to a 32-byte output string. The function can be viewed as an x-coordinate-only scalar multiplication on the curve

$$E_M : y^2 = x^3 + 486662x^2 + x$$

over $\mathbb{F}_{2^{255}-19}$. The curve points are denoted as $E_M(\mathbb{F}_{2^{255}-19})$. The first input string is interpreted as an integer scalar s, while the second input string is interpreted as a 32-byte encoding of x_P, the x-coordinate of a point $P \in E_M(\mathbb{F}_{2^{255}-19})$; the output is the 32-byte encoding of x_{sP}.

Given a 32-byte secret key and the 32-byte encoding of a standard base point defined in [Ber06], the function outputs the corresponding public key. Similarly, given a 32-byte secret key and a 32-byte public key, the function outputs the corresponding shared secret. Although the same routine can be used for generating both public keys and shared secrets, the public-key generation can be done much faster by performing the scalar multiplication on an equivalent curve. The rest of this section describes how we implement the Curve25519 function for shared-secret computation and public-key generation.

8.2.1 Shared-secret computation

Algorithm 1 The Montgomery ladder step for Curve25519

```
function LadderStep($x_2, z_2, x_3, z_3, x_P$)
    $t_0 \leftarrow x_3 - z_3$
    $t_1 \leftarrow x_2 - z_2$
    $x_2 \leftarrow x_2 + z_2$
    $z_2 \leftarrow x_3 + z_3$
    $z_3 \leftarrow t_0 \cdot x_2$; $z_2 \leftarrow z_2 \cdot t_1$        ▷ batched multiplications
    $x_3 \leftarrow z_3 + z_2$
    $z_2 \leftarrow z_3 - z_2$
    $x_3 \leftarrow x_3^2$; $z_2 \leftarrow z_2^2$        ▷ batched squarings
    $z_3 \leftarrow x_P \cdot z_2$;
    $t_0 \leftarrow t_1^2$; $t_1 \leftarrow x_2^2$        ▷ batched squarings
    $x_2 \leftarrow t_1 - t_0$
    $z_3 \leftarrow x_2 \cdot 121666$
    $z_2 \leftarrow t_0 + z_3$
    $z_2 \leftarrow x_2 \cdot z_2$; $x_2 \leftarrow t_1 \cdot t_0$        ▷ batched multiplications
    return ($x_2, z_2, x_3, z_3$)
end function
```

The best known algorithm for x-coordinate-only variable-base-point scalar multiplication on Montgomery curves is the *Montgomery ladder*. [BDL$^+$11], [BS12] and our software all use the Montgomery ladder for Curve25519 shared secret computation. Similar to the double-and-add algorithm, the algorithm also iterates through each bit of the scalar, from the most significant to the least significant one. For each bit of the scalar the ladder performs a differential addition and a doubling. The differential addition and the doubling together are called a *ladder step*. Since the ladder step can be carried out by a fixed sequence of field operations, the Montgomery ladder is almost intrinsically constant-time. We summarize the ladder step for Curve25519 in Algorithm 1. Note that Montgomery uses projective coordinates.

In order to make the best use of the vector unit (see Section 8.1), multiplications and squarings are handled in pairs whenever convenient. The way we pair multiplications is shown in the comments of Algorithm 1. It is not specified in [BS12] whether they pair multiplications and squarings in the same way, but this seems to be the most natural way. Note that the multiplication by 121666 (line 13) and the multiplication by x_1 (line 10) are not paired with other multiplications. We deal with the two multiplications as follows:

- Compute multiplications by 121666 without carries using 5 `vpmuludq`.
- Compute multiplications by x_1 without carries. This can be completed in 50 `vpmuludq` since we precompute the products of small constants (namely, 2, 19, and 38) and limbs in x_1 before the ladder begins.
- Perform batched carries for the two multiplications.

This uses far fewer cycles than handling the carries for the two multiplications separately.

Note that we often have to "transpose" data in the ladder step. More specifically, after an $\mathbf{M^2}$ which computes $(h_0, h'_0), \ldots, (h_9, h'_9)$, we might need to compute $h + h'$ and $h - h'$; see lines 6–8 of Algorithm 1. In this case, we compute (h_i, h_{i+1}), (h'_i, h'_{i+1}) from $(h_i, h'_i), (h_{i+1}, h'_{i+1})$ for $i \in \{0, 2, 4, 6, 8\}$, and then perform additions and subtractions on the vectors. The transpositions can be carried out using the "unpack" instructions `vpunpcklqdq` and `vpunpckhqdq`. Similarly, to obtain the operands for $\mathbf{M^2}$ some transpositions are also required. Unpack instructions are the same as `vpaddq` and `vpsubq` in terms of port utilization, so we also try to interleave them with $\mathbf{M^2}$ or $\mathbf{S^2}$ as described in Section 8.1.5.

8.2.2 Public-key generation

Instead of performing a fixed-base scalar multiplication directly on the Montgomery curve, we follow [LM13] to perform a fixed-base scalar multiplication on the twisted Edwards curve

$$E_T : -x^2 + y^2 = 1 - 121665/121666x^2y^2$$

over $\mathbb{F}_{2^{255}-19}$ and convert the result back to the Mongomery curve with one inversion. The curve points are denoted as $E_T(\mathbb{F}_{2^{255}-19})$. There is an efficiently computable birational equivalence between E_T and E_M, which means the curves share the same group structure and ECDLP difficulty. Unlike Mongomery curves, there are complete formulas for point addition and doubling on twisted Edwards curves; we follow [BDL$^+$11] to use the formulas for the extended coordinates proposed in [HWC$^+$08]. The complete formulas allow utilization of a table of many precomputed points to accelerate the scalar multiplication, which is the reason fixed-base multiplications (on both E_M and E_T) can be carried out much faster than variable-base scalar multiplications.

In [BDL$^+$11] a fixed-base scalar multiplication sB where $B \in E_T(\mathbb{F}_{2^{255}-19})$ and $s \in \mathbb{Z}$ (B corresponds to the standard base point in $E_M(\mathbb{F}_{2^{255}-19})$) is performed as follows: write s (modulo the order of B) as $\sum_{i=0}^{15} 16^i s_i$ where $s_i \in \{-8, -7, \ldots, 7\}$ and obtain sB by computing the summation of $s_0B, s_1 16B, \ldots, s_{15}16^{15}B$. To obtain $s_i 16^i B$, the strategy is to precompute several multiples of 16^iB and store them in a table, and then perform a constant-time table lookup using s_i as index on demand. [BDL+11] also shows how to reduce the size of the table by dividing the sum into two parts:

$$P_0 = s_0B + s_2 16^2 B + \cdots + s_{14}16^{14}B$$

and

$$P_1 = s_1B + s_3 16^2 B + \cdots + s_{15}16^{14}B.$$

$sB = P_0 + 16P_1$ is then obtained with 4 point doublings and 1 point addition. In this way, the table contains only multiples of $B, 16^2B, \ldots, 16^{14}B$.

We do better by vectorizing between computations of P_0 and P_1: all the data related to P_0 and P_1 are loaded into the lower half and upper half of the 128-bit registers, respectively. This type of computation pattern is very friendly for vectorization since there no need to "transpose" the data as in the case of Section 8.2.1.

While parallel point additions can be carried out easily, an important issue is how to perform parallel constant-time table lookups in an efficient way. In [BDL+11], suppose there is a need to lookup s_iP, the strategy is to precompute a table containing $P, 2P, \ldots, 8P$, and then the lookup is carried out in two steps:

- Load $|s_i|P$ in constant time, which is the main bottleneck of the table lookup.
- Negate $|s_i|P$ if s_i is negative.

For the first step it is convenient to use the conditional move instruction (`cmov`): To obtain each limb (of each coordinate) of $|s_i|P$, first initialize a 64-bit register to the corresponding limb of ∞, then for each of $P, 2P, \ldots, 8P$, conditionally move the corresponding limb into the register. Computation of the conditions and conditional negation are relatively cheap compared to the `cmov` instructions. [BDL+11] uses a 3-coordinate system for precomputed points, so the table-lookup function takes $3 \cdot 8 \cdot 5 = 120$ `cmov` instructions. The function takes 159 Sandy Bridge cycles or 158 Ivy Bridge cycles.

We could use the same routine twice for parallel table lookups, but we do better by using vector instructions. Here is a part of the inner loop of our qhasm ([Ber07b]) code.

```
v0 = mem64[input_1 + 0] x2
v1 = mem64[input_1 + 40] x2
v2 = mem64[input_1 + 80] x2
v0 &= mask1
v1 &= mask1
v2 &= mask1
t0 |= v0
t1 |= v1
t2 |= v2
```

Algorithm 2 The doubling function for twisted Edwards curves

```
function GE_DBL_P2($X, Y, Z$)
    $A \leftarrow X^2; B \leftarrow Y^2$                ▷ batched squarings
    $G \leftarrow A - B$
    $H \leftarrow A + B$
    $C \leftarrow 2Z^2; D = (X + Y)^2$                 ▷ batched squarings
    $E \leftarrow H - D$
    $I \leftarrow G + C$
    $X' \leftarrow E \cdot I; Y' \leftarrow G \cdot H$      ▷ batched multiplications
    $Z' \leftarrow G \cdot I$
    return $(X', Y', Z')$
end function
```

The first line `v0 = mem64[input_1 + 0] x2` loads the first and second limb (each taking 32 bits) of the first coordinate of P and broadcasts the value to the lower half and upper half of the 128-bit register `v0` using the `movddup` instruction. The line `v0 &= mask1` performs a bitwise AND of `v0` and a mask; the value in the mask depends on whether $s_i = 1$. Finally, `v0` is ORed into `t0`, which is initialized in a similar way as in the `cmov`-based approach. Similarly, the rest of the lines are for the second and third coordinates. Similar code blocks are repeated 7 more times for $2P, 3P, \ldots, 8P$, and all the code blocks are surrounded by a loop which iterates through all the limbs. In total it takes $3 \cdot 8 \cdot 5 \cdot 2 = 240$ logic instructions. The parallel table-lookup function (inlined in our implementation) takes less than 160 Sandy/Ivy Bridge cycles, which is almost twice as fast as the `cmov`-based table lookup function.

8.3 Vectorizing the Ed25519 signature scheme

This section describes how the Ed25519 verification is implemented with focus on the challenge of vectorization. Since the public-key generation and signing process, as the Curve25519 public-key generation, is bottlenecked by a fixed-base scalar multiplication on E_T, the reader can check Section 8.2.2 for the implementation strategy.

8.3.1 Ed25519 verification

[BDL$^+$11] verifies a message by computing the double-scalar multiplication of the form $s_1P_1 + s_2P_2$. The double-scalar multiplication is implemented using a generalization of the sliding-window method such that s_1P_1 and s_2P_2 share the doublings. With the same window sizes, we do better by vectorizing the point doubling and point addition functions.

On average each verification takes about 252 point doublings, accounting for more than 110000 cycles. There are two doubling functions in our implementation; `ge_dbl_p2`, which is adapted from the "$\mathcal{E} \leftarrow 2\mathcal{E}$" doubling described in [HWC$^+$08], is the most frequently used one; see [HWC$^+$08] for the reason to use different doubling and addition functions. On average `ge_dbl_p2` is called 182 times per verification, accounting for more than 74000 cycles. The function is summarized in Algorithm 2. Given $(X : Y : Z)$ representing $(X/Z, Y/Z) \in E_T$, the function returns $(X' : Y' : Z') = (X : Y : Z) + (X : Y : Z)$. As in Section 8.2.1, squarings and multiplications are paired whenever convenient. However it is not always possible to do so, as the multiplication in line 9 can not be paired with other operations. The single multiplication slows down the function, and the same problem also appears in addition functions.

Another problem is harder to see. $E = X^2 + Y^2 - (X + Y)^2$ has limbs with upper bound around $4 \cdot 2^{26}$, and $I = X^2 - Y^2 + 2Z^2$ has limbs with upper bound around $5 \cdot 2^{26}$. For the multiplication $E \cdot I$, limbs of either E or I have to be multiplied by 19 (see Section 8.1.2), which can be more than 32 bits. This problem is solved by performing extra carries on limbs in E before the multiplication. The same problem appears in the other doubling function.

In general the computation pattern for verification is not so friendly for vectorization. However, even in this case our software still gains non-negligible speedup over [BDL$^+$11] and [LM13]. We conclude that the power of vector unit on recent Intel microarchitectures might have been seriously underestimated, and implementors for ECC software should consider trying vectorized multipliers instead of serial multipliers.

9

How to manipulate curve standards: a white paper for the black hat

More and more Internet traffic is encrypted. This poses a threat to our society as it limits the ability of government agencies to monitor Internet communication for the prevention of terrorism and globalized crime. For example, an increasing number of servers use *Transport Layer Security* (TLS) as default (not only for transmissions that contain passwords or payment information) and also most modern chat applications encrypt all communication. This increases the cost of protecting society as it becomes necessary to collect the required information at the end points, i.e., either the servers or the clients. This requires agencies to either convince the service providers to make the demanded information available or to deploy a back door on the client system respectively. Both actions are much more expensive for the agencies than collecting unprotected information from the transmission wire.

Fortunately, under reasonable assumptions, it is feasible for agencies to fool users into deploying cryptographic systems that the users believe are secure but that the agencies are able to break.

Elliptic-curve cryptography. Elliptic-curve cryptography (ECC) has a reputation for high security and has become increasingly popular. For definiteness we consider the elliptic-curve Diffie–Hellman (ECDH) key-exchange protocol, specifically "ephemeral ECDH", which has a reputation of being the best way to achieve forward secrecy. The literature models ephemeral ECDH as the following protocol $\mathrm{ECDH}_{E,P}$, Diffie–Hellman key exchange using a point P on an elliptic curve E:

1. Alice generates a private integer a and sends the ath multiple of P on E.

2. Bob generates a private integer b and sends bP.

3. Alice computes abP as the ath multiple of bP.

4. Bob computes abP as the bth multiple of aP.

5. Alice and Bob encrypt data using a secret key derived from abP.

There are various published attacks showing that this protocol is breakable for many elliptic curves E, no matter how strong the encryption is. See Section 9.1 for details. However, there are also many (E, P) for which the public literature does not indicate any security problems. Similar comments apply to, e.g., elliptic-curve signatures.

This model begs the question of where the curve (E, P) comes from. The standard answer is that a central authority generates a curve for the public (while advertising the resulting benefits for security and performance).[1] This does not mean that the public will accept arbitrary curves; our main objective in this chapter is to analyze the security consequences of various possibilities for what the public will accept. The general picture is that Alice, Bob, and the central authority Jerry are actually participating in the following three-party protocol ECDH$_A$, where A is a function determining the public acceptability of a standard curve:

−1. Jerry generates a curve E, a point P, auxiliary data S with $A(E, P, S) = 1$. (The "seeds" for the NIST curves are examples of S; see Section 9.3.)

0. Alice and Bob verify that $A(E, P, S) = 1$.

1. Alice generates a private integer a and sends aP.

2. Bob generates a private integer b and sends bP.

3. Alice computes abP as the ath multiple of bP.

4. Bob computes abP as the bth multiple of aP.

5. Alice and Bob encrypt data using a secret key derived from abP.

Our paper is targeted at Jerry. We make the natural assumption that Jerry is cooperating with a heroic eavesdropper Eve to break the encryption used by potential terrorists Alice and Bob. The central question is how Jerry can use his curve-selection flexibility to minimize the attack cost.

Obviously the cost c_A of breaking ECDH$_A$ depends on A, the same way that the cost $c_{E,P}$ of breaking ECDH$_{E,P}$ depends on (E, P). One might think that, to evaluate c_A, one simply has to check what the public literature says about $c_{E,P}$, and then minimize $c_{E,P}$ over all (E, P, S) with $A(E, P, S) = 1$. The reality is more complicated, for three reasons:

[1] See, e.g., ANSI X9.62 [ANS99] ("public key cryptography for the financial services industry"), IEEE P-1363 [IEE00], SECG [Cer00a], NIST FIPS 186 [NIS00], ANSI X9.63 [ANS01], Brainpool [Bra05], NSA Suite B [NSA05], and ANSSI FRP256V1 [ANS11]. Note that this chapter is not a historical review of which standards have been sabotaged and which have not; it is a sabotage cost assessment and a guide for manipulating future standards.

1. There may be vulnerabilities not known to the public: curves E for which $c_{E,P}$ is smaller than indicated by the best public attacks. Our starting assumption is that Jerry and Eve are secretly aware of a vulnerability that applies to a fraction ϵ of all curves that the public believes to be secure. The obvious strategy for Jerry is to standardize a vulnerable curve. Of course, Jerry should object to any public suggestions that a vulnerable curve could have been standardized.

2. Some choices of A limit the number of curves E for which there *exists* suitable auxiliary data S. If $1/\epsilon$ is much larger than this limit then Jerry cannot expect any vulnerable (E, P, S) to have $A(E, P, S) = 1$. We show that, fortunately for Jerry, this limit is much larger than the public thinks it is. See Sections 9.4 and 9.5.

3. Other choices of A do not limit the number of vulnerable E for which S *exists* but nevertheless complicate Jerry's task of *finding* a vulnerable (E, P, S) with $A(E, P, S) = 1$. See Section 9.3 for analysis of the cost of this computation.

If Jerry succeeds in finding a vulnerable (E, P, S) with $A(E, P, S) = 1$, then Eve simply exploits the vulnerability, obtaining access to the information that Alice and Bob have encrypted for transmission.

Of course, this could require considerable computation for Eve, depending on the details of the secret vulnerability. Obviously, given the risk of this chapter being leaked to the public, it would be important for us to avoid discussing details of secret vulnerabilities, even if we were aware of such vulnerabilities.[2] Our goal in this chapter is not to evaluate the cost of Eve's computation, but rather to evaluate the impact of A and ϵ upon the cost of Jerry's computation.

For this evaluation it is adequate to use simplified models of secret vulnerabilities. We specify various artificial curve criteria that have no connection to vulnerabilities but that are satisfied by (E, P, S) with probability ϵ for various sizes of ϵ. We then evaluate how difficult it is for Jerry to find (E, P, S) that satisfy these criteria and that have $A(E, P, S) = 1$.

The possibilities that we analyze for A are models built from data regarding what the public will accept. See Figure 9.1 for the data flow. Consider, for example, the following data: the public has accepted without complaint the constants $\sin(1), \sin(2), \ldots, \sin(64)$ in MD5, the constants $\sqrt{2}, \sqrt{3}, \sqrt{5}, \sqrt{10}$ in SHA-1, the constants $\sqrt[3]{2}, \sqrt[3]{3}, \sqrt[3]{5}, \sqrt[3]{7}$ in SHA-2, the constant $(1 + \sqrt{5})/2$ in RC5, the constant $e = \exp(1)$ in Brainpool, the constant $1/\pi$ in ARIA, etc. All of these constants are listed in [Wik15b] as examples of "nothing up my sleeve numbers". Extrapolating from this data, we confidently predict that the public would accept, e.g., the constant $\cos(1)$ used in our example curve BADA55-VPR-224 in Section 9.4. Enumerating a complete list of acceptable constants would require more systematic psychological experiments, so we have chosen a conservative acceptability function A in Section 9.4 that allows just 17 constants and their reciprocals.

The reader might object to our specification of ECDH_A as implicitly assuming that the party sabotaging curve choices to protect society is the same as the party

[2] Note to any journalists who might end up reading this chapter: There are no secret vulnerabilities. Really. ECC is perfectly safe. You can quote Jerry.

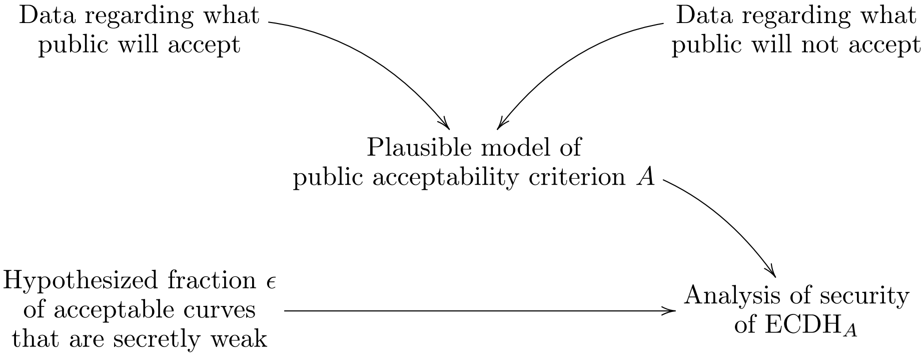
Data regarding what public will accept
Data regarding what public will not accept
Plausible model of public acceptability criterion A
Hypothesized fraction ϵ of acceptable curves that are secretly weak
Analysis of security of ECDH_A

Figure 9.1: Data flow in this chapter. The available data regarding public acceptability is stratified into five different models of the public acceptability criterion A, considered in Sections 9.2, 9.3, 9.4, 9.5, and 9.6 respectively, with five different shapes of the auxiliary curve data S. The security of each A is analyzed for variable ϵ.

issuing curve standards to be used by Alice and Bob. In reality, these two parties are different, and having the first party exercise sufficient control over the second party is often a delicate exercise in finesse. See, for example, [Kel03; CFN$^+$14].

Organization. Section 9.1 reviews the curve attacks known to the public and analyzes the probability that a curve resists these attacks; this probability has an obvious impact on the cost of generating curves. Section 9.2, as a warm-up, shows how to manipulate curve choices when A merely checks for these public vulnerabilities.

Section 9.3 shows how to manipulate "verifiably random" curve choices obtained by hashing seeds. Section 9.4 shows how to manipulate "verifiably pseudorandom" curve choices obtained by hashing "nothing-up-my-sleeves numbers". Section 9.5 shows how to manipulate "minimal" curve choices. Section 9.6 shows how to manipulate "the fastest curve".

Research contributions of this chapter. We appear to be the first to formally introduce the three-party protocol ECDH_A. The general idea of Section 9.3 is not new, but our cost analysis is new. We are the first to implement the attack,[3] showing how little computer power is necessary to target highly unusual curve properties. Our theoretical and experimental analysis of the percentage of secure curves (see Section 9.1) is also new.

The general idea of Sections 9.4 and 9.5 is new. We are the first to show that curves using so-called "nothing-up-my-sleeves numbers" can very well be manipulated to contain a secret vulnerability. We present concrete ways to gain many bits of freedom

[3] To be precise: No previous implementations are reported in the *public* literature.

and analyze how likely a new vulnerability needs to be in order to hide in this framework. It is surprising that millions of curves can be generated by plausible variations of the Brainpool [Bra05] curve-generation procedure, and that hundreds of thousands of curves can be generated by plausible variations of the Microsoft [BCL+15] curve-generation procedure.

As discussed in Sections 9.3.2 and 9.4, we encourage Jerry to experimentally study the exact boundary of what the public will accept. In followup work to Section 9.4, Aumasson has posted a "Generator of 'nothing-up-my-sleeve' (NUMS) constants" that "generates close to 2 million constants, and is easily tweaked to generate many more". See [Aum15].

9.1 Pesky public researchers and their security analyses

One obstacle Jerry has to face in deploying his backdoored elliptic curves is that public researchers have raised awareness of certain weaknesses an elliptic curve may have. Once sufficient awareness of a weakness has been raised, many standardization committees will feel compelled to mention that weakness in their standards. This in turn may alert the targeted users, i.e., the general public: some users will check what standards say regarding the properties that an elliptic curve should have or should not have.

The good thing about standards is that there are so many to choose from. Standards evaluating or claiming the security of various elliptic curves include ANSI X9.62 (1999) [ANS99], IEEE standard P1363 (2000) [IEE00], Certicom SEC 1 v1 (2000) [Cer00b], Certicom SEC 2 v1 (2000) [Cer00a], NIST FIPS 186-2 (2000) [NIS00], ANSI X9.63 (2001) [ANS01], Brainpool (2005) [Bra05], NSA Suite B (2005) [NSA05], Certicom SEC 1 v2 (2009) [Cer09], Certicom SEC 2 v2 (2010) [Cer10], OSCCA SM2 (2010) [OSC10a; OSC10b], ANSSI FRP256V1 (2011) [ANS11], and NIST FIPS 186-4 (2013) [NIS13]. These standards vary in many details, and also demonstrate important variations in public acceptability criteria, an issue explored in depth later in this chapter.

Unfortunately for Jerry, some public criteria have become so widely known that all of the above standards agree upon them. Jerry's curves need to satisfy these criteria. This means not only that Jerry will be unable to use these public attacks as back doors, but also that Jerry will have to bear these criteria in mind when searching for a vulnerable curve. Perhaps the vulnerability known secretly to Jerry does not occur in curves that satisfy the public criteria; on the other hand, perhaps this vulnerability occurs *more* frequently in curves that satisfy the public criteria than in general curves. The chance ϵ of a curve being vulnerable is defined relative to the curves that the public will accept.

This section has three goals:

- Review these standard criteria for "secure" curves, along with attacks those pesky researchers have found. Jerry must be careful, when designing and justifying his curve, to avoid revealing attacks outside this list; such attacks are not known to the public.

- Analyze the probability δ that a curve satisfies the standard security criteria. This has a direct influence on Jerry's curve-generation cost. Two particular criteria, "small cofactor" and "small twist cofactor", are satisfied by only a small fraction of curves.

- Analyze the probability that a curve is actually feasible to break by various public attacks. It turns out that there are many probabilities on different scales, showing that one should also consider a range of probabilities ϵ for Jerry's secret vulnerability. Recall that ϵ is, by definition, the probability that curves passing the public criteria are secretly vulnerable to Jerry's attack.

Each curve that Jerry tries works with probability only $\delta\epsilon$. The number of curves that Jerry can afford to try and is allowed to try depends on various optimizations and constraints analyzed later in this chapter; combining this number with $\delta\epsilon$ immediately reveals Jerry's overall success chance at creating a vulnerable curve that passes the public criteria, avoiding alarms from the pesky researchers.

9.1.1 Warning: math begins here

For simplicity we cover only prime fields here. If Jerry's secret vulnerability works only in binary fields then we would expect Jerry to have a harder time convincing his targets to use vulnerable curves, although of course he should try.

Let E be an elliptic curve defined over a large prime field $\mathbb{F}_p$. One can always write E in the form $y^2 = x^3 + ax + b$. Most curve standards choose $a = -3$ for efficiency reasons. Practically all curves have low-degree isogenies to curves with $a = -3$, so this choice does not affect security.

Write $|E(\mathbb{F}_p)|$ for the number of points on E defined over $\mathbb{F}_p$, and write $|E(\mathbb{F}_p)|$ as $p + 1 - t$. Hasse's theorem (see, e.g., [Sil09]) states that $|E(\mathbb{F}_p)|$ is in the "Hasse interval" $[p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}]$; i.e., t is between $-2\sqrt{p}$ and $2\sqrt{p}$.

Define ℓ as the largest prime factor of $|E(\mathbb{F}_p)|$, and define the "cofactor" h as $|E(\mathbb{F}_p)|/\ell$. Let P be a point on E of order ℓ.

9.1.2 Review of public ECDLP security criteria. Elliptic curve cryptography is based on the believed hardness of the *elliptic-curve discrete-logarithm problem* (ECDLP), i.e., the belief that it is computationally infeasible to find a scalar k satisfying $Q = kP$ given a random multiple Q of P on E. The state-of-the-art public algorithm for solving the ECDLP is Pollard's rho method (with negation), which on average requires approximately $0.886\sqrt{\ell}$ point additions. Most publications require the value ℓ to be large; for example, the SafeCurves web page [BL15a] requires that $0.886\sqrt{\ell} > 2^{100}$.

Some standards put upper limits on the cofactor h, but the limits vary. FIPS 186-2 [NIS00, page 24] claims that "for efficiency reasons, it is desirable to take the cofactor to be as small as possible"; the 2000 version of SEC 1 [Cer00b, page 17] required $h \leq 4$; but the 2009 version of SEC 1 [Cer09, pages 22 and 78] claims that there are efficiency benefits to "some special curves with cofactor larger than four" and thus requires merely $h \leq 2^{\alpha/8}$ for security level 2^{α}. We analyze a few possibilities for h and later give examples with $h = 1$; many standard curves have $h = 1$.

Another security parameter is the *complex-multiplication field discriminant* (CM field discriminant) which is defined as $D = (t^2 - 4p)/s^2$ if $(t^2 - 4p)/s^2 \equiv 1 \pmod 4$ or otherwise $D = 4(t^2 - 4p)/s^2$, where t is defined as $p + 1 - |E(\mathbb{F}_p)|$ and s^2 is the largest square dividing $t^2 - 4p$. One standard, Brainpool, requires $|D|$ to be large (by requiring a related quantity, the "class number", to be large). However, other standards do not constrain D, there are various ECC papers choosing curves where D is small, and the only published attacks related to the size of D are some improvements to Pollard's rho method on a few curves. If Jerry needs a curve with small D then it is likely that Jerry can convince the public to accept the curve. We do not pursue this possibility further.

All standards prohibit efficient *additive and multiplicative transfers*. An additive transfer reduces the ECDLP to an easy DLP in the additive group of $\mathbb{F}_p$; this transfer is applicable when ℓ equals p. A degree-k multiplicative transfer reduces the ECDLP to the DLP in the multiplicative group of $\mathbb{F}_{p^k}$ where the problem can be solved efficiently using index calculus if the *embedding degree* k is not too large; this transfer is applicable when ℓ divides $p^k - 1$. All standards prohibit $\ell = p$, ℓ dividing $p - 1$, ℓ dividing $p + 1$, and ℓ dividing various larger $p^k - 1$; the exact limit on k varies from one standard to another.

9.1.3 ECC security vs. ECDLP security

The most extensive public list of requirements is on the SafeCurves web page [BL15a]. SafeCurves covers hardness of ECDLP, generally imposing more stringent constraints than the standards listed in Section 9.1.2; for example, SafeCurves requires the discriminant D of the CM field to satisfy $|D| > 2^{100}$ and requires the order of p modulo ℓ, i.e., the embedding degree, to be at least $(\ell - 1)/100$. Potentially more troublesome for Jerry is that SafeCurves also covers the general security of ECC, i.e., the security of ECC implementations.

For example, if an implementor of NIST P-224 ECDH uses the side-channel-protected scalar-multiplication algorithm recommended by Brier and Joye [BJ02], reuses an ECDH key for more than a few sessions,[4] and fails to perform a moderately expensive input validation that has no impact on normal usage,[5] then a *twist attack* finds the user's secret key using approximately 2^{58} elliptic-curve additions. See [BL15a] for details. SafeCurves prohibits curves with low *twist security*, such as NIST P-224.

Luckily for Jerry, the other standards listed above focus on ECDLP hardness and impose very few additional ECC security constraints. This gives Jerry the freedom (1) to choose a non-SafeCurves-compliant curve that encourages insecure ECC implementations even if ECDLP is difficult, and (2) to deny that there are any security problems. Useful denial text can be found in a May 2014 presentation [Moo14a] from NIST: "The NIST curves do NOT belong to any known class of elliptic curves with weak security properties. No sufficiently large classes of weak curves are known."

[4] [CFN+14, Section 4.2] reports that Microsoft's SChannel automatically reuses "ephemeral" keys "for two hours".

[5] A very recent paper [JSS15] reports complete breaks of the ECC implementations in Bouncy Castle and Java Crypto Extension, precisely because those implementations fail to validate input points.

Unfortunately, there is some risk that twist-security and other SafeCurves criteria will be added to future standards.[6] This chapter considers the possibility that Jerry is forced to generate twist-secure curves; it is important for Jerry to be able to sabotage curve standards even under the harshest conditions. Obviously it is also preferable for Jerry to choose a curve for which *all* implementations are insecure, rather than merely a curve that encourages insecure implementations.

Twist-security requires the twist E' of the original curve E to be secure. If $|E(\mathbb{F}_p)| = p + 1 - t$ then $|E'(\mathbb{F}_p)| = p + 1 + t$. Define ℓ' as the largest prime factor of $p+1+t$. SafeCurves requires $0.886\sqrt{\ell'} > 2^{100}$ to prevent Pollard's rho method; $\ell' \neq p$ to prevent additive transfers; and p having order at least $(\ell' - 1)/100$ modulo ℓ' to prevent multiplicative transfers. SafeCurves also requires various "combined attacks" to be difficult; this is automatic when cofactors are very small, i.e. when $(p + 1 - t)/\ell$ and $(p + 1 + t)/\ell'$ are very small integers.

9.1.4 The probability δ of passing public criteria

This subsection analyzes the probability of random curves passing the public criteria described above.

We begin by analyzing how many random curves have small cofactors. As illustrations we consider cofactors $h = 1$, $h = 2$, and $h = 4$. Note that, for primes p large enough to pass a laugh test (at least 224 bits), curves with these cofactors automatically satisfy the requirement $0.886\sqrt{\ell} > 2^{100}$; in other words, requiring a curve to have a small cofactor supersedes requiring a curve to meet minimal public requirements for security against Pollard's rho method.

Let $\pi(x)$ be the number of primes $p \leq x$, and let $\pi(S)$ be the number of primes p in a set S. The prime-number theorem states that the ratio between $\pi(x)$ and $x/\log x$ converges to 1 as $x \to \infty$, where log is the natural logarithm. Explicit bounds such as [RS62] are not sufficient to count the number of primes in a short interval $I = [x - y, x]$, but there is nevertheless ample experimental evidence that $\pi(I)$ is very close to $y/\log x$ when y is larger than $\sqrt{x}$.

The number of integers in I of the form $\ell, 2\ell$, or 4ℓ, where ℓ is prime, is the same as the total number of primes in the intervals $I_1 = [x - y, x]$, $I_2 = [(x - y)/2, x/2]$ and $I_4 = [(x - y)/4, x/4]$, namely

$$\pi(I_1) + \pi(I_2) + \pi(I_4) \approx \frac{y}{\log x} + \frac{y/2}{\log{(x/2)}} + \frac{y/4}{\log{(x/4)}} = \sum_{h\in\{1,2,4\}} \frac{y/h}{\log(x/h)}.$$

Take $x = p + 1 + 2\sqrt{p}$ and $y = 4\sqrt{p}$ to see that the number of such integers in the Hasse interval is approximately $\sum_{h\in\{1,2,4\}}(4\sqrt{p}/h)/(\log{((p + 1 + 2\sqrt{p})/h)})$. The total number of integers in the Hasse interval is almost exactly $4\sqrt{p}$, so the chance of

[6] For example, after we wrote this, CFRG appeared to reach consensus on twist-secure curves. The resulting documents are still in draft form but the risk is clear. On the other hand, a recent document [LW15] claims that "using twist secure curves can lead to insecure implementations and degrade security"; the details of these claims have already received various public objections, but one can still imagine the authors of [LW15] issuing a new non-twist-secure standard.

an integer in the interval having the form ℓ, 2ℓ, or 4ℓ is approximately

$$\sum_{h\in\{1,2,4\}} \frac{1}{h \log\left((p+1+2\sqrt{p})/h\right)}. \tag{9.1}$$

This does not imply, however, that the same approximation is valid for the number of points on a random elliptic curve. It is known, for example, that the number of points on an elliptic curve is odd with probability almost exactly 1/3, not 1/2; this suggests that the number is prime less often than a uniformly distributed random integer in the Hasse interval would be.

A further difficulty is that we need to know not merely the probability that the cofactor h is small, but the joint probability that both h and $h' = (p+1+t)/\ell'$ are small. Even if one disregards the subtleties in the distribution of $p+1-t$, one should not expect (e.g.) the probability that $p+1-t$ is prime to be independent of the probability that $p+1+t$ is prime: for example, if one quantity is odd then the other is also odd.

Galbraith and McKee in [GM00, Conjecture B] formulated a precise conjecture for the probability of any particular h (called "k" there). Perhaps the techniques of [GM00] can be extended to formulate a conjecture for the probability of any particular pair (h, h'). However, no such conjectures appear to have been formulated yet, let alone tested.

To collect facts we performed the following experiment: take $p = 2^{224} - 2^{96} + 1$ (the NIST P-224 prime, which is also used in the following sections), and count the number of points on 1000000 curves. Specifically, we took the curves $y^2 = x^3 - 3x + 1$ through $y^2 = x^3 - 3x + 1000001$, skipping the non-elliptic curve $y^2 = x^3 - 3x + 2$. It is conceivable that the small coefficients make these curves behave nonrandomly, but the same type of nonrandomness appears naturally in Section 9.5, so this is a relevant experiment. Furthermore, the simple description makes the experiment easy to reproduce.

Within this sample we found probability 0.003705 of $h = 1$, probability 0.002859 of $h = 2$, and probability 0.002372 of $h = 4$, with total $0.008936 \approx 2^{-7}$. We also found, unsurprisingly, practically identical probabilities for the twist cofactor: probability 0.003748 of $h' = 1$, probability 0.002902 of $h' = 2$, and probability 0.002376 of $h' = 4$, with total 0.009026.

For comparison, Formula (9.1) evaluates to approximately 0.011300 (about 25% too optimistic), built from 0.006441 for $h = 1$ (about 74% too optimistic), 0.003235 for $h = 2$ (about 13% too optimistic), and 0.001625 for $h = 4$ (about 32% too pessimistic).

In our sample we found probability 0.000049 that simultaneously $h \in \{1, 2, 4\}$ and $h' \in \{1, 2, 4\}$. This provides reasonable confidence, although not a guarantee, that the events $h \in \{1, 2, 4\}$ and $h' \in \{1, 2, 4\}$ are statistically dependent: independence would mean that the joint event would occur with probability approximately 0.000081, so a sample of size 1000000 would contain ≤ 49 such curves with probability under 0.0001.

We found probability $0.000032 \approx 2^{-15}$ of $h = h' = 1$. Our best estimate, with the caveat of considerable error bars, is therefore that Jerry must try about 2^{15} curves before finding one with $h = h' = 1$. If Jerry is free to neglect twist security, searching only for $h = 1$, then the probability jumps by two orders of magnitude to about 2^{-8}. If Jerry is allowed to take any $h \in \{1, 2, 4\}$ then the probability is about 2^{-7}.

These probabilities are not noticeably affected by the SafeCurves requirements regarding the CM discriminant, additive transfers, and multiplicative transfers. Specifically, random curves have a large CM field discriminant, practically always meeting the SafeCurves CM criterion; none of our experiments found a CM field discriminant below 2^{100}. We also found, unsurprisingly, no curves with $\ell = p$. As for multiplicative transfers: Luca, Mireles, and Shparlinski gave a theoretical estimate [LMS04] for the probability that for a sufficiently large prime number p and a positive integer K with $\log K = O(\log \log p)$ a randomly chosen elliptic curve $E(\mathbb{F}_p)$ has embedding degree $k \leq K$; this result shows that curves with small embedding degree are very rare. The SafeCurves bound $K = (\ell - 1)/100$ is not within the range of applicability of their theorem, but experimentally we found that about 99% of all curves had a high embedding degree $\geq K$.

9.1.5 The probabilities for various feasible attacks

We now consider various feasible public attacks as models of Jerry's secret vulnerability. Specifically, for each attack, we evaluate the probability that the attack works against curves that were *not* chosen to be secure against this type of attack. Any such probability is a reasonable guess for an ϵ of interest to Jerry.

At the low end is, e.g., an additive transfer, applying only to curves having exactly p points. The probability here is roughly $p^{-1/2}$: e.g., below 2^{-100} for the NIST P-224 prime.

At the high end, most curves fail the "rho" and "twist" security criteria; see Section 9.1.4. But this does not mean that the curves are feasible to break, or that the breaking cost is low enough for Jerry to usefully apply to billions of targets. These security criteria are extremely cautious, staying far away from anything potentially breakable by these attacks. For example, $\ell \approx 2^{150}$ fails the SafeCurves security criteria but still requires about 2^{75} elliptic-curve operations to break by the rho attack, costing roughly 100 million watt-years of energy with current hardware, a feasible but highly nontrivial cost. A much smaller $\ell \approx 2^{120}$ would require about 2^{60} elliptic-curve operations, and breaking 2^{30} targets by standard multiple-target techniques would again require about 2^{75} elliptic-curve operations. Even smaller values of ℓ are of interest for twist attacks.

The prime-number theorem can be used to estimate the probabilities of various sizes of ℓ as in Section 9.1.4, but it loses applicability as ℓ drops below $\sqrt{p}$. To estimate the probability for a wider range of ℓ we use the following result by Dickman (see, e.g., [Gra08]). Define $\Psi(x, y)$ as the number of integers $\leq x$ whose largest prime factor is at most y; these numbers are called y-smooth integers. Dickman's result is then as follows:

$$\Psi(x, y) \sim x\rho(u) \qquad \text{as } x \to \infty, \qquad \text{where } x = y^u.$$

Here ρ, the "Dickman ρ function", satisfies $\rho(u) = 1$ for $0 \leq u \leq 1$ and $-u\rho'(u) = \rho(u-1)$ for $u \geq 1$, where ρ' means the right derivative. It is not difficult to compute $\rho(u)$ to high accuracy.

We experimentally verified how well ℓ adheres to this estimate, again for the NIST P-224 prime. For each k we used the Dickman rho function to compute an estimate

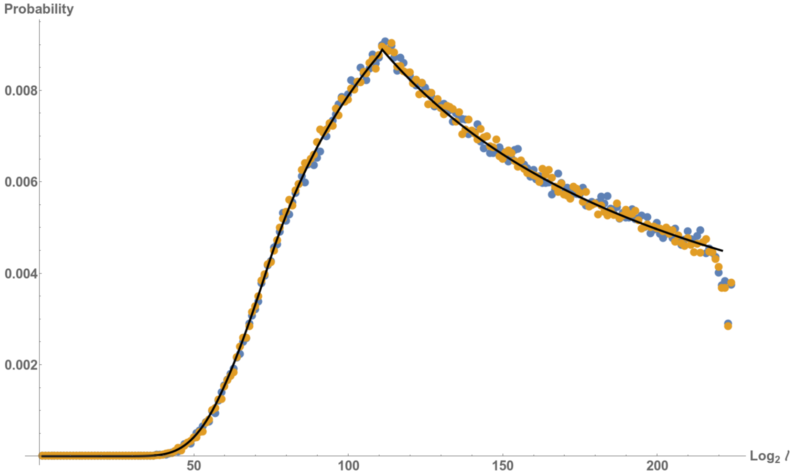

Figure 9.2: Largest prime factor versus the probability. Blue: The regular curves E. Orange: The twists of the curves E. Black: The Dickman estimate. Orange is more visible than blue because orange is plotted on top of blue.

for the number of integers in the Hasse interval whose largest prime factor has exactly k bits:

$$\Psi(p+1+2\sqrt{p}, 2^k) - \Psi(p+1-2\sqrt{p}, 2^k) - \Psi(p+1+2\sqrt{p}, 2^{k-1}) + \Psi(p+1-2\sqrt{p}, 2^{k-1}).$$

We divided this by $4\sqrt{p}$ (the size of the Hasse interval) to obtain the black graph in Figure 9.2. We also experimentally computed (for a somewhat smaller sample than in Section 9.1.4) the fraction of curves where ℓ has k bits, and the fraction of curves where ℓ' has k bits, shown as blue and orange dots in Figure 9.2. The dots are below the right end of the graph in Figure 9.2 for the reasons explained in Section 9.1.4; for smaller values of ℓ the estimate closely matches the experimental data.

About 20% of the 224-bit curves have $\ell < 2^{100}$, producing a tolerable rho attack cost, around 2^{50} elliptic-curve operations. However, $\rho(u)$ drops rapidly as u increases (it is roughly $1/u^u$), so the chance of achieving this reasonable cost also drops rapidly as the curve size increases. For 256-bit curves the chance is $\rho(2.56) \approx 0.12 \approx 2^{-3}$. For 384-bit curves the chance is $\rho(3.84) \approx 0.0073 \approx 2^{-7}$. For 512-bit curves the chance is $\rho(5.12) \approx 0.00025 \approx 2^{-12}$.

We now switch from considering rho attacks against arbitrary curves to considering twist attacks against curves with cofactor 1. For a twist attack to fit into 2^{50} elliptic-curve operations, the largest prime ℓ' dividing $p+1+t$ must be below 2^{100}, but also the *second-largest* prime dividing $p+1+t$ must be below 2^{50}; see generally [BL15a]. In other words, $p+1+t$ must be $(2^{100}, 2^{50})$-semismooth. Recall that an integer is defined to be (y, z)-semismooth if none of its prime factors is larger than y and at most one of its prime factors is larger than z. The portion of the twist attack corresponding to the second-largest prime is difficult to batch across multiple targets, so it is reasonable to consider even smaller limits for that prime.

p	$k = 30$	$k = 40$	$k = 50$	$k = 60$	$k = 70$	$k = 80$
P-224 prime	$2^{-15.74}$	$2^{-8.382}$	$2^{-4.752}$	$2^{-2.743}$	$2^{-1.560}$	$2^{-0.8601}$
P-256 prime	$2^{-20.47}$	$2^{-11.37}$	$2^{-6.730}$	$2^{-4.132}$	$2^{-2.551}$	$2^{-1.557}$
P-384 prime	$2^{-42.10}$	$2^{-25.51}$	$2^{-16.65}$	$2^{-11.37}$	$2^{-7.977}$	$2^{-5.708}$
P-521 prime	$2^{-68.64}$	$2^{-43.34}$	$2^{-29.57}$	$2^{-21.16}$	$2^{-15.63}$	$2^{-11.81}$

Table 9.1: Estimated probability that an elliptic curve modulo p has largest twist prime at most 2^{2k} and second largest twist prime at most 2^k, i.e., that an elliptic curve modulo p is vulnerable to a twist attack using approximately 2^k operations. Estimates rely on the method of [BP96] to compute asymptotic semismoothness probabilities.

We estimated this semismoothness probability using the same approach as for rho attacks. First, estimate the semismoothness probability for $p + 1 + t$ as the semismoothness probability for a uniform random integer in the Hasse interval. Second, estimate the semismoothness probability for a uniform random integer using a known two-variable generalization of ρ. Third, compute this generalization using a method of Bach and Peralta [BP96]. The results range from $2^{-6.730}$ for 256-bit curves down to $2^{-29.57}$ for 521-bit curves. Table 9.1 shows the results of similar computations for several sizes of primes and several limits on feasible attack costs.

To summarize, feasible attacks in the public literature have a broad range of success probabilities against curves not designed to resist those attacks; probabilities listed above include 2^{-4}, 2^{-8}, 2^{-11}, 2^{-16}, and 2^{-25}. It is thus reasonable to consider a similarly broad range of possibilities for ϵ, the probability that a curve passing public security criteria is vulnerable to Jerry's secret attack.

9.2 Manipulating curves

Here we target users with minimal acceptability criteria: i.e., we assume that $A(E, P, S)$ checks only the public security criteria for (E, P) described in Section 9.1. The auxiliary data S might be used to communicate, e.g., a precomputed $|E(\mathbb{F}_p)|$ to be verified by the user, but is not used to constrain the choice of (E, P). Curves that pass the acceptability criteria are safe against known attacks, but have no protection against Jerry's secret vulnerability.

9.2.1 Curves without public justification

Here are two examples of standard curves distributed without any justification for how they were chosen. These examples suggest that there are many ECC users who do in fact have minimal acceptability criteria.

As a first example, we look at the FRP256V1 standard [ANS11] published in 2011 by the Agence nationale de la sécurité des systèmes d'information (ANSSI). This curve is $y^2 = x^3 - 3x + b$ over $\mathbb{F}_p$, where

$$\begin{aligned} b &= \texttt{0xEE353FCA5428A9300D4ABA754A44C00FDFEC0C9AE4B1A1803075ED967B7BB73F}, \\ p &= \texttt{0xF1FD178C0B3AD58F10126DE8CE42435B3961ADBCABC8CA6DE8FCF353D86E9C03}. \end{aligned}$$

Another example is a curve published by the Office of State Commercial Cryptography Administration (OSCCA) in China along with the SM2 algorithms in 2010 (cf. [OSC10b; OSC10a]). The curve is of the same form as the ANSSI one with

$$\begin{aligned} b &= \texttt{0x28E9FA9E9D9F5E344D5A9E4BCF6509A7F39789F515AB8F92DDBCBD414D940E93}, \\ p &= \texttt{0xFFFFFFFEFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00000000FFFFFFFFFFFFFFFF}. \end{aligned}$$

Each curve E is also accompanied by a point P. The curves meet the ECDLP requirements[7] reviewed in Section 9.1. The only further data provided with these curves is data that could also have been computed efficiently by users from the above information. Nothing in the curve documentation suggests any verification that would have further limited the choice of curves.

9.2.2 The attack

The attack is straightforward. Since the only things that users check are the public security criteria, Jerry can continue generating curves for a fixed p (either randomly or not) that satisfy the public criteria until he gets one that is vulnerable to his secret attack. Alternatively, Jerry can generate curves vulnerable to his secret attack and check them against the public security criteria. Every attack (publicly) known so far allows efficient computation of vulnerable curves, so it seems likely that the same will be true for Jerry's secret vulnerability. After finding a vulnerable curve, Jerry simply publishes it.

Of course, Jerry's vulnerability must not depend on any properties excluded by the public security criteria, and there must be enough vulnerable curves. Enumerating 2^7 vulnerable curves over $\mathbb{F}_p$ is likely to be sufficient if Jerry can ignore twist-security, and enumerating 2^{15} vulnerable curves over $\mathbb{F}_p$ is likely to be sufficient even if twist-security is required. See Section 9.1.

Even if Jerry's curves are less frequent, Jerry can increase his chances by also varying the prime p. To simplify our analysis we do not take advantage of this flexibility in this section: we assume that Jerry is forced to reuse a particular standard prime such as a NIST prime or the ANSSI prime. We emphasize, however, that the standard security requirements do not seriously scrutinize the choice of prime, and existing standards vary in their choices of primes. Any allowed variability in p would also improve the attack in Section 9.4, and we do vary p in Section 9.5.

9.2.3 Implementation

We implemented the attack to demonstrate that it is really feasible in practice. In our implementation the same setting as above is adopted and even made more restrictive: the resulting curve should be of the form $y^2 = x^3 - 3x + b$ over $\mathbb{F}_p$, where p is the same as for the ANSSI curve. The public security criteria we consider are all the

[7] But not the SafeCurves requirements. Specifically, FRP256V1 has twist security 2^{79}, and the OSCCA curve has twist security 2^{96}.

```
p = 0xF1FD178C0B3AD58F10126DE8CE42435B3961ADBCABC8CA6DE8FCF353D86E9C03 # standard ANSSI prime
k = GF(p)

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n.is_prime() and (2*p+2-n).is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1
    and Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1)

A = p-3 # standard -3 modulo p
B = 0xBADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BD48
if secure(A,B):
  print 'p',hex(p).upper()
  print 'A',hex(A).upper()
  print 'B',hex(B).upper()

# output:
# p F1FD178C0B3AD58F10126DE8CE42435B3961ADBCABC8CA6DE8FCF353D86E9C03
# A F1FD178C0B3AD58F10126DE8CE42435B3961ADBCABC8CA6DE8FCF353D86E9C00
# B BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BD48
```

Figure 9.3: A procedure to generate the new BADA55-R-256 curve.

standard ECDLP security criteria plus twist security, and we further require that both cofactors are 1.

Of course, as explained in the introduction, we will not include any actual secret vulnerability in this white paper. We instead use a highly structured parameter b as an artificial model of a secret vulnerability. We show that we can construct a curve with such a b that passes all the public criteria. In reality, Jerry would select a curve with a secret vulnerability rather than a curve with our artificial model of a vulnerability, and would use a trustworthy curve name such as TrustedCurve-R-256.

Our attack is implemented using the Sage computer algebra system [Ste+15]. We took `0x5AFEBADA55ECC5AFEBADA55ECC5AFEBADA55ECC5AFEBADA55ECC5AFEBADA55EC` as the start value for b and incremented b until we found a curve that meets the public security criteria. This corresponds to Jerry iteratively checking whether curves that are vulnerable to the secret attack fulfill the public criteria.

As a result we found a desired curve, which we call BADA55-R-256, with b = `0x5AFEBADA55ECC5AFEBADA55ECC5AFEBADA55ECC5AFEBADA55ECC5AFEBADA5A57` after 1131 increments within 78 minutes on a single core of an AMD CPU.[8] One can easily check using a computer-algebra system that the curve does meet all the public criteria. It is thus clear that users who only verify public security criteria can be very easily attacked, and Jerry has an easy time if he is working for or is trusted by ANSSI, OSCCA, or a similar organization.

[8] Note that a *lucky* attacker starting from `0xBADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA` is able to find the following secure parameter already within 43 minutes after only 622 increments: b = `0xBADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BADA55BD48`.

9.3 Manipulating seeds

Section 9.2 deals with the easiest case for Jerry that the users are satisfied verifying public security criteria. However some audiences might demand justifications for the curve choices. In this section, we consider users who are suspicious that the curve parameters might be maliciously chosen to enable a secret attack. Empirically many users are satisfied if they get a *hash verification routine* as justification; see, e.g., ANSI X9.62 [ANS99], IEEE P1363 [IEE00], SEC 2 [Cer10], or NIST FIPS 186-2 [NIS00]. Hash verification routines mean that Jerry cannot use a very small set of vulnerable curves, but we will show below that he has good chances to get vulnerable curves deployed if they are just somewhat more common.

9.3.1 Hash verification routine

As the name implies, a hash verification routine involves a cryptographic hash function. The inputs to the routine are the curve parameters and a seed that is published along with the curve. Usually the seed is hashed to compute a curve parameter or point coordinate. The ways of computing the parameters differ but the public justification is that these bind the curve parameters to the hash value, making them hard to manipulate since the hash function is preimage resistant[9]. In addition the user verifies a set of public security criteria. We focus on the obstacle that Jerry faces and call curves that can be verified with such routines *verifiably hashed curves*.

For Jerry's marketing we do not recommend the phrase "verifiably hashed": it is better to claim that the curves are totally random (even though this is not what is being verified) and that these curves could not possibly be manipulated (even though Jerry is in fact quite free to manipulate them). For example, ANSI X9.62 [ANS99, page 31] speaks of "selecting an elliptic curve verifiably at random"; SEC 2 [Cer10, copy and paste: page 8 and page 18] claims that "verifiably random parameters offer some additional conservative features" and "that the parameters cannot be predetermined". NIST's marketing in [NIS00] is not as good: NIST uses the term "pseudo-random curves".

Below we recall the curve verification routine for the NIST P-curves. The routine is specified in NIST FIPS 186-2 [NIS00].

Each NIST P-curve is of the form $y^2 = x^3 - 3x + b$ over a prime field $\mathbb{F}_p$ and is published with a seed s. The hash function SHA-1 is denoted as `SHA1`; recall that SHA-1 produces a 160-bit hash value. The bit length of p is denoted by m. We use $\text{bin}(i)$ to denote the 20-byte big-endian representation of some integer i and use $\text{int}(j)$ to denote the integer with binary expansion j. For given parameters b, p, and s, the verification routine is:

1. Let $z \leftarrow \text{int}(s)$. Compute $h_i \leftarrow \texttt{SHA1}(s_i)$ for $0 \leq i \leq v$, where $s_i \leftarrow \text{bin}((z + i) \bmod 2^{160})$ and $v = \lfloor (m-1)/160 \rfloor$.

2. Let h be the rightmost $m - 1$ bits of $h_0||h_1||\cdots||h_v$. Let $c \leftarrow \text{int}(h)$.

[9] If Jerry has a back door in the hash function this situation is no different than in Section 9.2, so we will not assume this feature.

3. Verify that $b^2c = -27$ in $\mathbb{F}_p$.

To generate a verifiably hashed curve one starts with a seed and then follows the same steps 1 and 2 as above. Instead of step 3 one tries to solve for b given c; this succeeds for about 50% of all choices for s. The public perception is that this process is repeated with fresh seeds until the first resulting curve satisfies all public security criteria.

9.3.2 Acceptability criteria

One might think that the public acceptability criteria are defined by the NIST verification routine stated above: i.e., $A(E, P, s) = 1$ if and only if (E, P) passes the public security criteria from Section 9.1 and (E, s) passes the verification routine stated above with seed s and E defined as $y^2 = x^3 - 3x + b$.

However, the public acceptability criteria are not actually so strict. P1363 allows $y^2 = x^3 + ax + b$ without the requirement $a = -3$. P1363 does require $b^2c = a^3$ where c is a hash as above, but neither P1363 nor NIST gives a justification for the relation $b^2c = a^3$, and it is clear that the public will accept different relations. For example, the Brainpool curves (see Section 9.4) use the simpler relations $a = g$ and $b = h$ where g and h are separate hashes. One can equivalently view the Brainpool curves as following the P1363 procedure but using a different hash for c, namely computing c as g^3/h^2 where again g and h are separate hashes. Furthermore, even though NIST and Brainpool both use SHA-1, SHA-1 is not the only acceptable hash function; for example, Jerry can easily argue that SHA-1 is outdated and should be replaced by SHA-2 or SHA-3.

We do not claim that the public would accept *any* relation, or that the public would accept *any* choice of "hash function", allowing Jerry just as much freedom as in Section 9.2. The exact boundaries of public acceptability are complicated and not immediately obvious. We have determined approximations to these boundaries by extrapolating from existing data (see, e.g., Section 9.4), and we encourage Jerry to carry out large-scale scientific surveys, while taking care to prevent leaks to the public.

9.3.3 The attack

Jerry begins the attack by defining a public hash verification routine. As explained above, Jerry has some flexibility to modify this routine. This flexibility is not *necessary* for the rest of the attack in this section (for example, Jerry can use exactly the NIST verification routine) but a more favorable routine does improve the *efficiency* of the attack. Our cost analysis below makes a particularly efficient choice of routine.

Jerry then tries one seed after another until finding a seed for which the verifiably hashed curve (1) passes the public security criteria but (2) is subject to his secret vulnerability. Jerry publishes this seed and the resulting curve, pretending that the seed was the first random seed that passed the public security criteria.

9.3.4 Optimizing the attack

Assume that the curves vulnerable to Jerry's secret attack are randomly distributed over the curves satisfying the public security criteria. Then the success probability that a seed leads to a suitable curve is the probability that a curve is vulnerable to the secret attack times the probability that a curve satisfies the public security criteria. Depending on which condition is easier to check Jerry runs many hash computations to compute candidate b's, checks them for the easier criterion and only checks the surviving choices for the other criterion. The hash computations and security checks for each seed are independent from other seeds; thus, this procedure can be parallelized with an arbitrary number of parallel computing instances.

We generated a family of curves to show the power of this method and highlight the computing power of hardware accelerators (such as GPUs or Xeon Phis). We began by defining our own curve verification routine and implementing the corresponding secret generation routine. The hash function we use is Keccak with 256-bit output instead of SHA-1. The hash value is $c = \text{int}(\text{Keccak}(s))$, and the relation is simply $b = c$ in $\mathbb{F}_p$. All choices are easily justified: Keccak is the winner of the SHA-3 competition and much more secure than SHA-1; using a hash function with a long output removes the weird order of hashed components that smells suspicious and similarly $b = c$ is as simple and unsuspicious as it can get. In reality, however, these choices greatly benefit the attack: the GPUs efficiently search through many seeds in parallel, one single computation of Keccak has a much easier data flow than in the method above, and having b computed without any expensive number-theoretic computation (such as square roots) means that the curve can be tested already on the GPUs and only the fraction that satisfies the first test is passed on to the next stage. Of course, for a real vulnerability we would have to add the cost of checking for that vulnerability, but minimizing overhead is still useful.

Except for the differences stated above, we followed closely the setting of the NIST P-curves. The target is to generate curves of the form $y^2 = x^3 - 3x + b$ over $\mathbb{F}_p$, and we consider 3 choices of p: the NIST P-224, P-256, and P-384 primes. (For P-384 we switched to Keccak with 384-bit output.) As a placeholder "vulnerability" we define E to be vulnerable if b starts with the hex-string `BADA55EC`. This fixes 8 hex digits, i.e., it simulates a 1-out-of-2^{32} attack. In addition we require that the curves meet the standard ECDLP criteria plus twist security and have both cofactors equal to 1.

9.3.5 Implementation

Our implementation uses NVIDIA's CUDA framework for parallel programming on GPUs. A high-end GPU today allows several thousand threads to run in parallel, though at a frequency slightly lower than high-end CPUs. We let each thread start with its own random seed. The threads then hash the seeds in parallel. After hashing, each thread outputs the hash value if it starts with the hex-string `BADA55EC`. To restart, each seed is simply increased by 1, so no new source of randomness is required. Checking whether outputs from GPUs also satisfy the public security criteria is done by running a Sage [Ste+15] script on CPUs. Since only 1 out of 2^{32} curves has the desired pattern, the CPU computation is totally hidden by GPU computation. Longer

```
import binascii
import simplesha3
hash = simplesha3.keccakc512 # SHA-3 winner with 256-bit output

p = 2^224 - 2^96 + 1 # standard NIST P-224 prime
k = GF(p)

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n.is_prime() and (2*p+2-n).is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1
    and Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1)

def str2int(seed):
  return Integer(seed.encode('hex'),16)

A = p-3
S = '3CC520E9434349DF680A8F4BCADDA648D693B2907B216EE55CB4853DB68F9165'
B = str2int(hash(binascii.unhexlify(S))) # verifiably random
if secure(A,B):
  print 'p',hex(p).upper()
  print 'A',hex(A).upper()
  print 'B',hex(B).upper()

# output:
# p FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF000000000000000000000001
# A FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFFFFFFFFFE
# B BADA55ECFD9CA54C0738B8A6FB8CF4CCF84E916D83D6DA1B78B622351E11AB4E
```

Figure 9.4: A procedure to generate the new "verifiably random" BADA55-VR-224 curve. Since the hash output is more than 256 bits, we implicitly reduce it modulo p.

strings, corresponding to less likely vulnerabilities, make GPUs even more powerful for our attack scheme.

In the end we found 3 "vulnerable" verifiably hashed curves: BADA55-VR-224, BADA55-VR-256, and BADA55-VR-384, each corresponding to one of the three NIST P-curves. See Figures 9.4, 9.5, and 9.6. Of course, as in Section 9.2, Jerry would use a secret vulnerability rather than our artificial "vulnerability", and would use the name TrustedCurve-VR rather than BADA55-VR.

As an example, BADA55-VR-256 was found within 7 hours, using a cluster of 41 NVIDIA GTX780 GPUs (`http://blog.cr.yp.to/20140602-saber.html`). Each GPU is able to carry out 170 million 256-bit-output Keccak hashes in a second. Most of the instructions are bitwise logic instructions. On average each core performs 0.58 bitwise logic instructions per cycle while the theoretical maximum throughput is 0.83. We have two explanations for the gap: first, each thread uses many registers, which makes the number of active warps too small to fully hide the instruction latency; second, there is not quite enough instruction-level parallelism to fully utilize the cores in this GPU architecture. We also tested our implementation on K10 GPUs. Each of them carries out only 61 million hashes per second. This is likely to be caused by register spilling: the K10 GPUs have only 63 registers per thread instead of the 255 registers of the GTX780. Using a sufficient amount of computing power easily allows Jerry to deal with secret vulnerabilities that have smaller probabilities of occurrence than 2^{-32}.

```
import binascii
import simplesha3
hash = simplesha3.keccakc512 # SHA-3 winner with 256-bit output

p = 2^256 - 2^224 + 2^192 + 2^96 - 1 # standard NIST P-256 prime
k = GF(p)

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n.is_prime() and (2*p+2-n).is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1
    and Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1)

def str2int(seed):
  return Integer(seed.encode('hex'),16)

A = p-3
S = '3ADCC48E36F1D1926701417F101A75F000118A739D4686E77278325A825AA3C6'
B = str2int(hash(binascii.unhexlify(S))) # verifiably random
if secure(A,B):
  print 'p',hex(p).upper()
  print 'A',hex(A).upper()
  print 'B',hex(B).upper()

# output:
# p FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF
# A FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFC
# B BADA55ECD8BBEAD3ADD6C534F92197DEB47FCEB9BE7E0E702A8D1DD56B5D0B0C
```

Figure 9.5: A procedure to generate the new "verifiably random" BADA55-VR-256 curve.

9.4 Manipulating nothing-up-my-sleeve numbers

There are some particularly pesky researchers who do not shut up even when provided with a verification routine as in the previous section. These researchers might even think of the powerful attack presented in the previous section.

In 1999, M. Scott complained about the choice of unexplained seeds for the NIST curves [Sco99] and concluded "Do they want to be distrusted?":

> `[...] Consider now the possibility that one in a million of all curves have an exploitable structure that "they" know about, but we don't. Then "they" simply generate a million random seeds until they find one that generates one of "their" curves. Then they get us to use them. And remember the standard paranoia assumptions apply - "they" have computing power way beyond what we can muster. So maybe that could be 1 billion.`
>
> `A much simpler approach would generate more trust. Simply select B as an integer formed from the maximum number of digits of pi that provide a number B which is less that p. Then keep incrementing B until the number of points on the curve is prime. Such a curve will be accepted as "random" as all would accept that the decimal digits of pi have no`

```
import binascii
import simplesha3
hash = simplesha3.keccakc768 # SHA-3 winner with 384-bit output

p = 2^384 - 2^128 - 2^96 + 2^32 - 1 # standard NIST P-384 prime
k = GF(p)

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n.is_prime() and (2*p+2-n).is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1
    and Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1)

def str2int(seed):
  return Integer(seed.encode('hex'),16)

A = p-3
S = 'CA9EBD338A9EE0E6862FD329062ABC06A793575A1C744F0EC24503A525F5D06E'
B = str2int(hash(binascii.unhexlify(S))) # verifiably random
if secure(A,B):
  print 'p',hex(p).upper()
  print 'A',hex(A).upper()
  print 'B',hex(B).upper()

# output:
# p FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
#   FFFFFFFFFFFFFFFEFFFFFFFF0000000000000000FFFFFFFF
# A FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
#   FFFFFFFFFFFFFFFEFFFFFFFF0000000000000000FFFFFFFC
# B BADA55EC3BE2AD1F9EEEA5881ECF95BBF3AC392526F01D4C
#   D13E684C63A17CC4D5F271642AD83899113817A61006413D
```

Figure 9.6: A procedure to generate the new "verifiably random" BADA55-VR-384 curve.

> unfortunate interaction with elliptic curves. We would all accept that such a curve had not been specially "cooked".
>
> So, sigh, why didn't they do it that way? Do they want to be distrusted?

In the same vein the German ECC Brainpool consortium expressed skepticism [Bra05, Introduction] and suggested using natural constants in place of random seeds. They coined the term "verifiably pseudorandom" for this method of generating seeds. Others speak of "nothing-up-my-sleeves numbers", a nice reference to magicians which we will take as an inspiration to our endeavor to show how Jerry can play this system. We comment that "nothing-up-my-sleeves numbers" also appear in other areas of cryptography and can be manipulated in similar ways, but this chapter focuses on manipulation of elliptic curves.

9.4.1 The Brainpool procedure

Brainpool requires that "curves shall be generated in a pseudo-random manner using seeds that are generated in a systematic and comprehensive way". Brainpool produces each curve coefficient by hashing a seed extracted from the bits of $e = \exp(1)$. This first curve cannot be expected to meet Brainpool's security criteria, so Brainpool counts systematically upwards from this initial seed until finding a curve that does meet the security criteria. Brainpool uses a similar procedure to generate primes.

We have written a Sage implementation, emphasizing simplicity and clarity, of the prime-generation and curve-generation procedures specified in the Brainpool standard [Bra05, Section 5]. For example, Figure 9.7 (designed to be shown to the public) uses Brainpool's procedure to generate a 224-bit curve. The output consists of the following "verifiably pseudorandom" integers p, a, b defining an elliptic curve $y^2 = x^3 + ax + b$ over $\mathbb{F}_p$:

$$p = \texttt{0xD7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF}$$

$$a = \texttt{0x2B98B906DC245F2916C03A2F953EA9AE565C3253E}\underline{\texttt{8AEC4BFE84C659E}}$$

$$b = \texttt{0x6}\underline{\texttt{8AEC4BFE84C659E}}\texttt{BB8B81DC39355A2EBFA3870D98976FA2F17D2D8D}$$

We have added underlines to point out an embarrassing collision of substrings, obviously quite different from what one expects in "pseudorandom" strings.

What happened here is that the Brainpool procedure generates each of a and b as truncations of concatenations of various hash outputs (since the selected hash function, SHA-1, produces only 160-bit outputs), and there was a collision in the hash inputs. Specifically, Brainpool uses the same seed-increment function for three purposes: searching for a suitable a; moving from a to b; and moving within the concatenations. The first hash used in the concatenation for a was fed through this increment function to obtain the second hash, and was fed through the same increment function to obtain the first hash used in the concatenation for b, producing the overlap visible above.

A reader who checks the Brainpool standard [Bra05] will find that the 224-bit curve listed there does not have the same (a, b), and does not have this overlap. The reason for this is that, astonishingly, the 224-bit standard Brainpool curve was not actually produced by the standard Brainpool procedure. In fact, although the reader will find overlaps in the standard 192-bit, 256-bit, 384-bit, and 512-bit Brainpool curves, *none* of the standard Brainpool curves below 512 bits were produced by the standard Brainpool procedure. In the case of the 160-bit, 224-bit, 320-bit, and 384-bit Brainpool curves, one can immediately demonstrate this discrepancy by observing that the gap listed between "seed A" and "seed B" in [Bra05, Section 11] is larger than 1, while the standard procedure always produces a gap of exactly 1.

A procedure that actually *does* generate the Brainpool curves appeared a few years later in the Brainpool RFC [LM10] and is reimplemented in Figure 9.8. For readers who do not enjoy playing a "spot the differences" game between Figures 9.7 and 9.8, we explain how the procedures differ:

- The procedure in [LM10] assigns seeds to an $(a^*ab^*b)^*$ pattern. It tries consecutive seeds for a until finding that $-3/a$ is a 4th power, then tries further seeds for b until finding that b is not a square, then checks whether the resulting curve meets Brainpool's security criteria. If this fails, it goes back to trying further seeds for a etc.

- The original procedure in [Bra05] assigns seeds to an $(a^*ab)^*$ pattern. It tries consecutive seeds for a until finding that $-3/a$ is a 4th power, then uses the next seed for b, then checks whether b is a non-square and whether the curve meets Brainpool's security criteria. If this fails, it goes back to trying further seeds for a etc.

Figure 9.9 shows our implementation of the procedure from [LM10] for all output sizes, including both Brainpool prime generation and Brainpool curve generation. The subroutine `secure` in this implementation also includes an "early abort" (using "division polynomials"), improving performance by an order of magnitude without changing the output; Figure 9.7 omits this speedup for simplicity. Our implementations also skip checking a few security criteria that have negligible probability of failing, such as having large CM field discriminant (see Section 9.1); these criteria are trivially verified after the fact.

We were surprised to discover the failure of the Brainpool standard procedure to generate the Brainpool standard curves. We have not found this failure discussed, or even mentioned, anywhere in the Brainpool RFCs or on the Brainpool web pages. We have also not found any updates or errata to the Brainpool standard after [Bra05]. One would expect that having a "verifiably pseudorandom" curve not actually produced by the specified procedure would draw more public attention, unless the public never actually tried verifying the curves, an interesting possibility for Jerry. We do not explore this line of thought further: we make the worst-case assumption that future curves will be verified by the public, using tools that Jerry is unable to corrupt.

The Brainpool standard also includes the following statement [Bra05, page 2]: "It is envisioned to provide additional curves on a regular basis for users who wish to change curve parameters regularly, cf. Annex H2 of [X9.62], paragraph 'Elliptic curve domain parameter cryptoperiod considerations'." However, the procedure for generating further "verifiably pseudorandom" curves is not discussed. One possibility is to continue the original procedure past the first (a, b) pair, but this makes new curves more and more expensive to verify. Another possibility is to replace e by a different natural constant.

9.4.2 The BADA55-VPR-224 procedure

We now present a new and improved verifiably pseudorandom 224-bit curve, BADA55-VPR-224. BADA55-VPR-224 uses the standard NIST P-224 prime, i.e., $p = 2^{224} - 2^{96} + 1$.

To avoid Brainpool's complications of concatenating hash outputs, we upgrade from the deprecated SHA-1 hash function to the state-of-the-art maximum-security SHA3-512 hash function. We also upgrade to requiring maximum twist security: i.e., both the cofactor and the twist cofactor are required to be 1.

Brainpool already generates seeds using $\exp(1) = e$ and generates primes using $\arctan(1) = \pi/4$, and MD5 already uses $\sin(1)$, so we use $\cos(1)$. We eliminate Brainpool's contrived, complicated[10] search pattern for a: we simply count upwards, trying every seed for a, until finding the first secure (a, b). The full 160-bit seed for a is the 32-bit counter followed by $\cos(1)$. We complement this seed to obtain the seed for b, ensuring maximal difference between the two seeds.

[10] As shown in Section 9.4.1, even Brainpool didn't get these details right.

Figure 9.11 is a Sage script implementing the BADA55-VPR-224 generation procedure. This procedure is simpler and more natural than the Brainpool procedure in Figure 9.8. Here is the resulting curve:

$$\begin{aligned} a &= \texttt{0x7144BA12CE8A0C3BEFA053ED}\underline{\texttt{BADA55}}\texttt{5A42391AC64F052376E041C7D4AF23195E} \\ &\ \ \texttt{BD8D83625321D452E8A0C3BB0A048A26115704E45DCEB346A9F4BD9741D14D49,} \\ b &= \texttt{0x5C32EC7FC48CE1802D9B70DBC3FA574EAF015FCE4E99B43EBE3468D6EFB2276B} \\ &\ \ \texttt{A3669AFF6FFC0F4C6AE4AE2E5D74C3C0AF97DCE17147688DDA89E734B56944A2.} \end{aligned}$$

9.4.3 How BADA55-VPR-224 was generated: exploring the space of acceptable procedures

The surprising collision of Brainpool substrings had an easy explanation: two hashes in the Brainpool procedure were visibly given the same input. The surprising appearance of the 24-bit string `BADA55` in a above has no such easy explanation. There are 128 hexadecimal digits in a, so one expects this substring to appear anywhere within a with probability $123/2^{24} \approx 2^{-17}$.

The actual explanation is as follows. We decided in advance that we would force `BADA55` to appear somewhere in a as our artificial model of a “vulnerability”. We then identified millions of natural-sounding “verifiably pseudorandom” procedures, and enumerated (using a few hours on our cluster) approximately 2^{20} of these procedures. The space of “verifiably pseudorandom” procedures has many dimensions analyzed below, such as the choice of hash function, the length of the input seed, the update function between seeds, and the initial constant for deriving the seed: i.e., each procedure is defined by a combination of hash function, seed length, etc. The exact number of choices available in any particular dimension is relatively unimportant; what is important is the exponential effect from combining many dimensions.

Since 2^{20} is far above 2^{17}, it is unsurprising that our “vulnerability” appeared in quite a few of these procedures. We selected one of those procedures and presented it as Section 9.4.2 as an example of what could be shown to the public. See Figure 9.12 for another example[11] of such a procedure, generating a BADA55-VPR2-224 curve, starting from e instead of $\cos(1)$. We could have easily chosen a more restrictive “vulnerability”.

The structure of this attack means that Jerry can use the same attack to target a real vulnerability that has probability 2^{-17}, or (with reasonable success chance) even 2^{-20}, perhaps even reusing our database of curves. As in Section 9.2 and Section 9.3, Jerry should use the name TrustedCurve-VPR rather than BADA55-VPR.

In this section we do not manipulate the choice of prime, the choice of curve shape, the choice of cofactor criterion, etc. Taking advantage of this flexibility (see Section 9.5) would increase the number of natural-sounding Brainpool-like procedures above 2^{30}.

[11] Presenting two examples with the same string `BADA55` gives the reader of this chapter some assurance that we did, in fact, choose this string in advance. Otherwise we could have tried to fool the reader as follows: generate a relatively small number of curves, search for an interesting-sounding string in the results, write the previous sections of this chapter to target that string (rather than `BADA55`), and pretend that we had chosen this string in advance.

Our experience is that Alice and Bob, when faced with a *single* procedure such as Section 9.4.2 (or Section 9.4.1), find it extremely difficult to envision the entire space of possible procedures (they typically see just a few dimensions of flexibility), and find it inconceivable that the space could have size as large as 2^{20}, never mind 2^{30}. This is obviously a helpful phenomenon for Jerry.

9.4.4 Manipulating bit-extraction procedures

Consider the problem of extracting a fixed-length string of bits from (e.g.) the constant $e = \exp(1) = 2.71828\ldots = (10.10110111\ldots)_2$. Here are several plausible options for the starting bit position:

- Start with the most significant bit: i.e., take bits of e at bit positions 2^1, 2^0, 2^{-1}, 2^{-2}, etc.
- Start immediately after the binary point: i.e., take bits of e at bit positions 2^{-1}, 2^{-2}, etc. For some constants this is identical to the first option: consider, e.g., the first MD5 constant $\sin(1) = 0.84\ldots$.
- Start with the most significant *nibble*: i.e., take bits of e at bit positions 2^3, 2^2, 2^1, 2^0, 2^{-1}, 2^{-2}, etc.
- Start with the most significant *byte*: i.e., take bits of e at bit positions 2^7, 2^6, 2^5, etc.
- Start with the byte at position 0. In the case of e this is the same as the fourth option. In the case of $\sin(1)$ this means prepending 8 zero bits to the fourth option.

These options can be viewed as using different maps from real numbers x to real numbers y with $0 \le y < 1$: the first map takes x to $|x|/2^{\lfloor \log_2 |x| \rfloor}$, the second map takes x to $x - \lfloor x \rfloor$, the third map takes x to $|x|/16^{\lfloor \log_{16} |x| \rfloor}$, etc. Brainpool used the third of these options, describing it as using "the hexadecimal representation" of e. Jerry can use similarly brief descriptions for any of the options without drawing the public's attention to the existence of other options. We implemented the first, second, and fourth options; for an average constant this produced slightly more than 2 distinct possibilities for real numbers y.

Jerry can easily get away with extracting a k-bit integer from y by truncation (i.e., $\lfloor 2^k y \rfloor$) or by rounding (i.e., $\lceil 2^k y \rfloor$). Jerry can defend truncation (which has fundamentally lower accuracy) as simpler, and can defend rounding as being quite standard in mathematics and the physical sciences; but we see no reason to believe that Jerry would be challenged in the first place. We implemented both options, gaining a further factor of 1.5.

Actually, Brainpool uses the bit position indicated above only for the low-security 160-bit Brainpool curve (which Jerry can disregard as already being a non-problem for Eve). As shown in Figure 9.9, Brainpool shifts to subsequent bits of e for the 192-bit curve, then to further bits for the 224-bit curve, etc. Brainpool uses 160 bits for each curve (see below), so the seed for the 256-bit curve (which Jerry can reasonably guess would be the most commonly used curve) is shifted by 480 bits. This number 480 depends on how many lower security levels are allocated (an obvious target of manipulation), and on exactly how many bits are allocated to those seeds. A further option, pointed out in [Mer14] by Merkle (Brainpool RFC co-author), is to reverse the order of curve sizes; the number 480 then depends on how many *higher* security levels are allocated. Yet another option is to put curve sizes in claimed order of usage. We did not implement any of the options described in this paragraph.

9.4.5 Manipulating choices of hash functions

The latest (July 2013) revision of the NIST ECDSA standard [NIS13, Section 6.1.1] specifically requires that "the security strength of a hash function used [for curve generation] **shall** meet or exceed the security strength associated with the bit length". The original NIST curves are exempted from this rule by [NIS13, footnote 2], but this rule prohibits SHA-1 for (e.g.) new 224-bit curves. On the other hand, a more recent Brainpool-related curve-selection document [Mer14] states that "For a PRNG, SHA-1 was (and still is) sufficiently secure."

Jerry has at least 10 plausible options for standard hash functions used to generate (e.g.) 256-bit curves:

- SHA-1. "We follow the Brainpool standard. What matters is preimage resistance, and SHA-1 still provides more than 2^{128} preimage resistance."
- SHA-256. "The trusted, widely deployed SHA-256 standard."
- SHA-384. "SHA-2 at the security level required to handle both sizes of Suite B curves."
- SHA-512. "The maximum-security SHA-512 standard."
- SHA-512/256. "NIST's standard wide-pipe hash function."
- SHA3-256. "The state-of-the-art SHA-3 standard at a 2^{128} security level."
- SHA3-384. "The state-of-the-art SHA-3 standard, at the security level required to handle both sizes of Suite B curves."
- SHA3-512. "The maximum-security state-of-the-art SHA3-512 standard."
- SHAKE128. "The state-of-the-art SHA-3 standard at a 2^{128} security level, providing flexible output sizes."
- SHAKE256. "The state-of-the-art SHA-3 standard at a 2^{256} security level, providing flexible output sizes."

There are also several non-NIST hash functions with longer track records than SHA-3. Any of RIPEMD-128, RIPEMD-160, RIPEMD-256, RIPEMD-320, Tiger, Tiger/128, Tiger/160, and Whirlpool would have been easily justifiable as a choice of hash function before 2006. MD5 and all versions of Haval would have been similarly justifiable before 2004.

Since we targeted a 224-bit curve we had even more standard NIST hash-function options. For simplicity we implemented just 10 hash-function options, namely the following variants of Keccak, the SHA-3 competition winner: Keccak-224, Keccak-256, Keccak-384, Keccak-512, "default" Keccak ("capacity" $c = 576$, 128 output bytes), Keccak-128 (capacity $c = 256$, 168 output bytes), SHA3-224 (which has different input padding from Keccak-224, changing the output), SHA3-256, SHA3-384, and SHA3-512. All of these Keccak/SHA-3 choices can be implemented efficiently with a single code base and variable input parameters.

9.4.6 Manipulating counter sizes

The simplest way to obtain a 160-bit "verifiably pseudorandom" output with SHA-1 is to hash the empty string. Curve generation needs many more outputs (since most curves do not pass the public security criteria), but the simplest way to obtain 2^β "verifiably pseudorandom" outputs is to hash all β-bit inputs.

Hash-function implementations are often limited to byte-aligned inputs, so it is natural to restrict β to a multiple of 8. If each output has chance 2^{-15} of producing an acceptable curve (see Section 9.1) then $\beta = 16$ finds an acceptable curve with chance nearly 90% ("this is retroactively justified by our successfully finding a curve, so there was no need for us to consider backup plans"); $\beta = 24$ fails with negligible probability ("we chose the smallest β for which the probability of failure was negligible"); $\beta = 32$ is easily justified by reference to 32-bit machines; $\beta = 64$ is easily justified by reference to 64-bit machines.

Obviously Brainpool takes a more complicated approach, using bits of some natural constant to further "randomize" its outputs. The standard way to randomize a hash is to concatenate the randomness (e.g., bits of e) with the input being hashed (the counter). Brainpool instead *adds* the randomness to the input being hashed. The Brainpool choice is not secure as a general-purpose randomized hash, although these security problems are of no relevance to curve generation. There is no evidence of public objections to Brainpool's use of addition here (and to the overall complication introduced by the extra randomization), so there is also no reason to think that the public would object to the more standard concatenation approach.

Overall there are 13 plausible possibilities here: the 4 choices of β above, with the counter on the left of the randomness; the 4 choices of β above, with the counter on the right of the randomness; the counter being added to the randomness; and 4 further possibilities in which the randomness is partitioned into an initial value for a counter (for the top bits) and the remaining seed (for the bottom bits). We implemented the first 9 of these 13 possibilities.

9.4.7 Manipulating hash input sizes

ANSI X9.62 requires $\geq$160 input bits for its hash input. One way for Jerry to advertise a long input is that it allows many people to randomly generate curves with a low risk of collision. For example, Jerry can advertise

- a 160-bit input as allowing 2^{64} curves with only a 2^{-32} risk of collision;

- a 256-bit input as allowing 2^{64} curves with only a 2^{-128} risk of collision; or
- a 384-bit input as allowing 2^{128} curves with only a 2^{-128} risk of collision.

All of these numbers sound perfectly natural. Of course, what Jerry is actually producing is a single standard for many people to use, so multiple-curve collision probabilities are of no relevance, but (in the unlikely event of being questioned) Jerry can simply say that the input length was chosen for "compatibility" with having users generate their own curves.

Jerry can advertise longer input lengths as providing "curve coverage". A 512-bit input will cover a large fraction of curves, even for primes as large as 512 bits. A 1024-bit input is practically guaranteed to cover all curves, and to produce probabilities indistinguishable from uniform. Jerry can also advertise, as input length, the "natural input block length of the hash function".

We implemented all 6 possibilities listed above. We gained a further factor of 2 by storing the seed (and counter) in big-endian format ("standard network byte order") or little-endian format ("standard CPU byte order").

9.4.8 Manipulating the (a, b) hash pattern

It should be obvious from Section 9.4.1 that there are many degrees of freedom in the details of how a and b are generated: how to distribute seeds between a and b; whether to require $-3/a$ to be a 4th power in $\mathbb{F}_p$; whether to require b to be a non-square in $\mathbb{F}_p$; whether to concatenate hash outputs from left to right or right to left; exactly how many bits to truncate hash outputs to (Brainpool uses one bit fewer than the prime; Jerry can argue for the same length as the prime "for coverage", or more bits "for indistinguishability"); whether to truncate to rightmost bits (as in Brainpool) or leftmost bits (as in various NIST requirements; see [NIS13]); et al.

For simplicity we eliminated the concatenation and truncation, always using a hash function long enough for the target 224-bit prime. We also eliminated the options regarding squares etc. We implemented a total of just 8 choices here. These choices vary in (1) whether to allocate seeds primarily to a or primarily to b and (2) how to obtain the alternate seed (e.g., the seed for a) from the primary seed (e.g., the seed for b): plausible options include complement, rotate 1 byte left, rotate 1 byte right, and four standard versions of 1-bit rotations.

9.4.9 Manipulating natural constants

As noted at the beginning of this chapter, the public has accepted dozens of "natural" constants in various cryptographic functions, and sometimes reciprocals of those constants, without complaint. Our implementation started with just 17 natural constants: π, e, Euler gamma, $\sqrt{2}$, $\sqrt{3}$, $\sqrt{5}$, $\sqrt{7}$, $\log(2)$, $(1+\sqrt{5})/2$, $\zeta(3)$, $\zeta(5)$, $\sin(1)$, $\sin(2)$, $\cos(1)$, $\cos(2)$, $\tan(1)$, and $\tan(2)$. We gained an extra factor of almost 2 by including reciprocals.

Jerry could be creative and use previously unused numbers such as numbers derived from some historical document or newspaper, personal information of, e.g., arbitrary celebrities in an arbitrary order, arbitrary collections of natural or physical constants and even a combination of several sources. For example, NewDES [Wik15a] derives its S-Box from the United States Declaration of Independence. If the public accepts numbers with such flimsy justifications as "nothing-up-my-sleeves numbers" then Jerry obviously has as much flexibility as in Section 9.3. We make the worst-case assumption that the public is not quite as easily fooled, and similarly that the public would not accept $703e^{(\sqrt[8]{30}+4\pi)/9\sin(\sqrt[3]{16})}$ as a "nothing-up-my-sleeve number".

9.4.10 Implementation

Any combination of the above manipulations defines a "systematic" curve-generation procedure. This procedure outputs the first curve parameters (using the specified update function) that result in a "secure" curve according to the public security tests. However, performing all public security tests for each set of parameters considered by each procedure is very costly. Instead, we split the attack into two steps:

1. For a given procedure f_i we iterate over the seeds $s_{i,k}$ using the specific update function of f_i. We check each parameter candidate from seed $s_{i,k}$ for our secret `BADA55` vulnerability. After a certain number of update steps the probability that we passed valid, secure parameters is very high; thus, we discard the procedure and start over with another one. If we find a candidate exhibiting the vulnerability, we perform the public security tests on this particular candidate. If the `BADA55` candidate passes, we proceed to step 2.

2. We perform the whole public procedure f_i starting with seed $s_{i,0}$ and check whether there is any valid parameter set passing the public security checks already before the `BADA55` parameters are reached. If there is such an earlier parameter set, we return to step 1 with the next procedure f_{i+1}.

The largest workload in our attack scenario is step 2, the re-checking for earlier safe curve parameters before `BADA55` candidates. The public security tests are not well suited for GPU parallelization; the first step of the attack procedure is relatively cheap and a GPU parallelization of this step does not have a remarkable impact on the overall runtime. Therefore, we implemented the whole attack only for the CPUs of the Saber cluster and left the GPUs idle.

We initially chose 8000 as the limit for the update counter to have a very good chance that the first secure twist-secure curve starting from the seed is the curve with our vulnerability. For example, BADA55-VPR-224 was found with counter just 184, and there was only a tiny risk of a smaller counter producing a secure twist-secure curve (which we checked later, in the second step). In total $\approx 2^{33}$ curves were covered by this limited computation; more than 2^{18} were secure and twist-secure. We then pushed the 8000 limit higher, performing more computation and finding more curves. This gradually increased the risk of the counter not being minimal, something that we would have had to address by the techniques of Section 9.5; but this issue still did not affect, e.g., BADA55-VPR2-224, which was found with counter 28025.

9.5 Manipulating minimality

Instead of supporting “verifiably pseudorandom” curves as in Section 9.4, some researchers have advocated choosing “verifiably deterministic” curves.

Both approaches involve specifying a “systematic” procedure that outputs a curve. The difference is that in a “verifiably pseudorandom” curve the curve coefficient is the output of a hash function for the *first hash input* that meets specified curve criteria, while a “verifiably deterministic” curve uses the *first curve coefficient* that meets specified curve criteria. Typically the curve uses a “verifiably deterministic” prime, which is the *first prime* that meets specified prime criteria.

Eliminating the hash function and hash input makes life harder for Jerry: it eliminates the main techniques that we used in previous sections to manipulate curve choices. However, as we explain in detail in this section, Jerry still has many degrees of freedom. Jerry can manipulate the concept of “first curve coefficient”, can manipulate the concept of “first prime”, can manipulate the curve criteria, and can manipulate the prime criteria, with public justifications claiming that the selected criteria provide convenience, ease of implementation, speed of implementation, and security.

In Section 9.4 we did not manipulate the choice of prime: we obtained a satisfactory level of flexibility in other ways. In this section, the choice of prime is an important component of Jerry’s flexibility. It should be clear to the reader that the techniques in this section to manipulate the prime, the curve criteria, etc. can be backported to the setting of Section 9.4, adding to the flexibility there.

We briefly review a recent proposal that fits into this category and then proceed to work out how much flexibility is left for Jerry.

9.5.1 NUMS curves

In early 2014, Bos, Costello, Longa, and Naehrig [BCL$^+$15] proposed 13 Weierstrass and 13 Edwards curves, spread over 3 different security levels. Each curve was generated following a deterministic procedure (similar to the procedure proposed in [BHK$^+$13]). Given that there are up to 10 different procedures per security level we cannot review all of them here but [BCL$^+$15] is a treasure trove of arguments to justify different prime and curve properties and we will use this to our benefit below.

The same authors together with Black proposed a set of 6 of these curves as an Internet-Draft [BBC$^+$14] referring to these curves as “Nothing Up My Sleeve (NUMS) Curves”. Note that this does not match the common use of “nothing up my sleeves”; see, e.g., the Wikipedia page [Wik15b]. These curves are claimed in [LC14] to have “independently-verifiable provenance”, as if they were not subject to any possible manipulation; and are claimed in [BBC$^+$15] to be selected “without any hidden parameters, reliance on randomness or any other processes offering opportunities for manipulation of the resulting curves”. What we analyze in this section is the extent to which Jerry can manipulate the resulting curves.

9.5.2 Choice of security level

Jerry may propose curves aiming for multiple security levels. To quote the Brainpool-curves RFC [LM10] “The level of security provided by symmetric ciphers and hash functions used in conjunction with the elliptic curve domain parameters specified in this RFC should roughly match or exceed the level provided by the domain parameters.” Table 1 in that document justifies security levels of 80, 96, 112, 128, 160, 192, and 256 bits. We consider the highest five to be easy sells. For the smaller ones Jerry will need to be more creative and, e.g., evoke the high cost of energy for small devices.

9.5.3 Choice of prime

There are several parts to choosing a prime once the security level is fixed.

Choice of prime size

For a fixed security level α it should take about 2^α operations to break the DLP. The definition of “operation” leaves some flexibility. The choices for the bit length r of the prime are:

- Exactly 2α, see e.g., [BCL+15].
- Exactly $2\alpha - 1$, see e.g., [BCL+15].
- Exactly $2\alpha - 2$, see e.g., [BCL+15].
- Exactly $2\alpha + 1$ to make up for the loss of $\sqrt{\pi/4}$ in the Pollard-rho complexity.
- Exactly $2\alpha + 2$ to *really* make up for the loss of $\sqrt{\pi/4}$ in the Pollard-rho complexity.

  $\vdots$

- Exactly $2\alpha + \beta$ to make up for the loss through precomputations for multi-target attacks.
- Exactly $2\alpha - 3$ to make arithmetic easier and because each elliptic-curve operation takes at least 3 bit operations.
- Exactly $2\alpha - 4$ to make arithmetic easier and because each elliptic-curve operation takes at least 4 bit operations.

  $\vdots$

- Exactly $2\alpha - \gamma$ to make arithmetic easier and because each elliptic-curve operation takes at least $2^{\gamma/2}$ bit operations.

These statements provide generic justifications for 8 options (actually even more, but we take a power of 2 to simplify). In the next two steps we show how to select different primes for each of these requirements. If the resulting p has additional beneficial properties these generic arguments might not be necessary, but they might be required if a competing (and by some measure superior) proposal can be excluded on the basis of not following the same selection criterion. If Jerry wants to highlight such benefits in his prime choice he may point to fast reduction or fast multiplication in a particular redundant representation with optimal limb size.

Choice of prime shape

The choices for the prime shape are:

- A random prime. This might seem somewhat hard to justify outside the scope of the previous section because arithmetic in $\mathbb{F}_p$ becomes slower, but members of the ECC Brainpool working group published several helpful arguments [LMS+14]. The most useful one is that random primes mean that the blinding factor in randomizing scalars against differential side-channel attacks can be chosen smaller.

- A pseudo-Mersenne prime, i.e. a prime of the shape $2^r \pm c$. The most common choice is to take c to be the smallest integer for a given r which leads to a prime because this makes reduction modulo the prime faster. (To reduce modulo $2^r \pm c$, divide by 2^r and add $\mp c$ times the dividend to the remainder.) See, e.g., [BCL+15]. Once r is fixed there are two choices for the two signs.

- A Solinas prime, i.e. a prime of the form $2^r \pm 2^v \pm 1$ as chosen for the Suite B curves [NSA05]. Also for these primes speed of modular reduction is the common argument. The difference $r - v$ is commonly chosen to be a multiple of the word size. Jerry can easily argue for multiples of 32 and 64. We skip this option in our count because it is partially subsumed in the following one.

- A "Montgomery-friendly" prime, i.e. a prime of the form $2^{r-v}(2^v - c) \pm 1$. These curves speed up reductions if elements in $\mathbb{F}_p$ are represented in Montgomery representation, $r - v$ is a multiple of the word size and c is less than the word size. Common word sizes are 32 and 64, giving two choices here. We ignore the flexibility of the ± because that determines p modulo 4, which is considered separately.

There are of course infinitely many random primes; in order to keep the number of options reasonable we take 4 as an approximation of how many prime shapes can be easily justified, making this a total of 8 options.

Choice of prime congruence

Jerry can get an additional bit of freedom by choosing whether to require $p \equiv 1 \pmod 4$ or to require $p \equiv 3 \pmod 4$. A common justification for the latter is that computations of square roots are particularly fast which could be useful for compression of points, see, e.g., [Bra05; BCL+15]. (In fact one can also compute square roots efficiently for $p \equiv 1 \pmod 4$, in particular for $p \equiv 5 \pmod 8$, but Jerry does not need to admit this.) To instead justify $p \equiv 1 \pmod 4$, Jerry can point to various benefits of having $\sqrt{-1}$ in the field: for example, twisted Edwards curves are fastest when $a = -1$, but completeness for $a = -1$ requires $p \equiv 1 \pmod 4$.

If Jerry chooses twisted Hessian curves he can justify restricting to $p \equiv 1 \pmod 3$ to obtain complete curve arithmetic.

9.5.4 Choice of ordering of field elements

The following curve shapes each have one free parameter. It is easy to justify choosing this parameter as the smallest parameter under some side conditions. Here smallest can be chosen to mean smallest in $\mathbb{N}$ or as the smallest power of some fixed generator g of $\mathbb{F}_p^*$. The second option is used in, e.g., a recent ANSSI curve-selection document [FPR$^+$15, Section 2.6.2]: "we define ... g as the smallest generator of the multiplicative group ... We then iterate over ... $b = g^n$ for $n = 1, \ldots,$ until a suitable curve is found." Each choice below can be filled with these two options.

9.5.5 Choice of curve shape and cofactor requirement

Jerry can justify the following curve shapes:

1. Weierstrass curves, the most general curve shape. The usual choice is $y^2 = x^3 - 3x + b$, leaving one parameter b free. For simplicity we do not discuss the possibility of choosing values other than -3.

2. Edwards curves, the speed leader in fixed-base scalar multiplication offering complete addition laws. The usual choices are $ax^2+y^2 = 1+dx^2y^2$, for $a \in \{\pm 1\}$, leaving one parameter d free. The group order of an Edwards curve is divisible by 4.

3. Montgomery curves, the speed leader for variable-base scalar multiplication and the simplest to implement correctly. The usual choices are $y^2 = x^3 + Ax^2 + x$, leaving one parameter A free. The group order of a Montgomery curve is divisible by 4.

4. Hessian curves, a cubic curve shape with complete addition laws (for twisted Hessian). The usual choices are $ax^3+y^3+1 = dxy$, where a is a small non-cube, leaving one parameter d free. The group order of a Hessian curve is divisible by 3, making twisted Hessian curves the curves with the smallest cofactor while having complete addition.

The following choices depend on the chosen curve shape, hence we consider them separately.

Weierstrass curves

Most standards expect the point format to be (x, y) on Weierstrass curves. Even when computations want to use the faster Edwards and Hessian formulas, Jerry can easily justify specifying the curve in Weierstrass form. This also ensures backwards compatibility with existing implementations that can only use the Weierstrass form.

The following are examples of justifiable choices for the cofactor h of the curve:

- Require cofactor exactly 1, as in Suite B and Brainpool.

- Require cofactor exactly 2, the minimum cofactor that allows the techniques of [BHK+13] to transmit curve points as uniform random binary strings for censorship circumvention.
- Require cofactor exactly 3, the minimum cofactor that allows Hessian arithmetic.
- Require cofactor exactly 4, the minimum cofactor that allows Edwards arithmetic.
- Require cofactor exactly 12, the minimum cofactor that allows both Hessian arithmetic and Edwards arithmetic.
- Take the first curve having cofactor below $2^{\alpha/8}$. This cofactor limit is standardized in [Cer10] and [NIS13]. (This cofactor will almost always be larger than 12.)
- Take the first curve having cofactor below $2^{\alpha/8}$ and a multiple of 3.
- Take the first curve having cofactor below $2^{\alpha/8}$ and a multiple of 4.
- Take the first curve having cofactor below $2^{\alpha/8}$ and a multiple of 12.
- Replace "cofactor below $2^{\alpha/8}$" with the SafeCurves requirement of a largest prime factor above 2^{200}.

On average these choices produce slightly more than 8 options; the last few options sometimes coincide.

The curve is defined as $y^2 = x^3 - 3x + b$ where b is minimal under the chosen criterion. Changing from positive b to negative b changes from a curve to its twist if $p \equiv 3 \pmod 4$, and (as illustrated by additive transfers) this change does not necessarily preserve security. However, this option makes only a small difference in our final total, so for simplicity we skip it.

Hessian curves

A curve given in Hessian form (and chosen minimal there) can be required to have minimal cofactor, minimal cofactor while being compatible with Edwards form, cofactor smaller than $2^{\alpha/8}$, or largest prime factor larger than 2^u. This leads to 8 options considering positive and negative values of d. Of course other restrictions on the cofactor are possible.

Edwards curves

For Edwards curves we need to split up the consideration further:

Edwards curves with $p \equiv 3 \pmod 4$

Curves with $a = -1$ are attractive for speed but are not complete in this case. Nevertheless [BCL+15] argues for this option, so we have additionally the choice between aiming for a complete or an $a = -1$ curve.

A curve given in (twisted) Edwards form (and chosen minimal there) can be required to have minimal cofactor, minimal cofactor while being compatible with Hessian form, cofactor smaller than $2^{\alpha/8}$, or largest prime factor larger than 2^u (and the latter in combination with Hessian if desired). This leads to at least 8 choices considering completeness; for minimal cofactors [BCL+15] shows that minimal choices for positive and negative values of d are not independent. To stay on the safe side we count these as 8 options only.

Edwards curves with $p \equiv 1 \pmod 4$

The curves $x^2 + y^2 = 1 + dx^2y^2$ and $-x^2 + y^2 = 1 - dx^2y^2$ are isomorphic because -1 is a square, hence taking the smallest positive value for d finds the same curve as taking the smallest negative value for the other sign of a. Jerry can however insist or not insist on completeness. Justifying non-completeness if the smallest option is complete however seems a hard sell.

Because $2p+2 \equiv 4 \pmod 8$ one of the curve and its twist will have order divisible by 8 while the other one has remainder 4 modulo 8. Jerry can require cofactor 4, as the minimal cofactor, or cofactor 8 if he chooses the twist with minimal cofactor as well and is concerned that protocols will only multiply by the cofactor of the curve rather than by that of the twist. The other options are the same as above. Again, to stay on the safe side, we count this as 8 options only.

Montgomery curves

There is a significant overlap between choosing the smallest Edwards curve and the smallest Montgomery curve. In order to ease counting and avoid overcounting we omit further Montgomery options.

Summary of curve choice

We have shown that Jerry can argue for $8 + 8 + 8 = 24$ options.

9.5.6 Choice of twist security

We make the worst-case assumption, as discussed in Section 9.1, that future standards will be required to include twist security. However, Jerry can play twist security to his advantage in changing the details of the twist-security requirements. Here are three obvious choices:

- Choose the cofactor of the twist as small as possible. Justification: This offers maximal protection.

- Choose the cofactor of the twist to be secure under the SEC recommendation, i.e. $h' < 2^{\alpha/8}$. Justification: This is considered secure enough for the main curve, so it is certainly enough for the twist.

- Choose the curve such that the curve passes the SafeCurves requirement of 2^{100} security against twist attacks. Justification: Attacks on the twist cannot use Pollard rho but need to do a brute-force search in the subgroups. The SafeCurves requirement captures the actual hardness of the attack.

Jerry can easily justify changes to the bound of 2^{100} by pointing to a higher security level or reducing it because the computations in the brute-force part are more expensive. We do not use this flexibility in the counting.

9.5.7 Choice of global vs. local curves

Jerry can take the first prime (satisfying some criteria), and then, for that prime, take the first curve coefficients (satisfying some criteria). Alternatively, Jerry can take the first possible curve coefficients, and then, for those curve coefficients, take the first prime. These two options are practically guaranteed to produce different curves. For example, in the Weierstrass case, Jerry can take the curve $y^2 = x^3 - 3x + 1$, and then search for the first prime p so that this curve over $\mathbb{F}_p$ satisfies the requirements on cofactor and twist security. If Jerry instead takes $y^2 = x^3 - 3x + g$ as in [FPR+15, Section 2.6.2], p must also meet the requirement that g be primitive in $\mathbb{F}_p$.

In mathematical terminology, the second option specifies a curve over a "global field" such as the rationals $\mathbb{Q}$, and then reduces the curve modulo suitable primes. This approach is particularly attractive when presented as a family of curves, all derived from the same global curve.

9.5.8 More choices

Brainpool [Bra05] requires that the number of points on the curve is less than p but also presents an argument for the opposite choice:

> To avoid overruns in implementations we require that $\#E(GF(p)) < p$. In connection with digital signature schemes some authors propose to use $q > p$ for security reasons, but the attacks described e.g. in [BRS] appear infeasible in a thoroughly designed PKI.

So Jerry can choose to insist on $p < |E(\mathbb{F}_p)|$ or on $p > |E(\mathbb{F}_p)|$.

9.5.9 Overall count

We have shown that Jerry can easily argue for 4 (security level) ·8 (prime size) ·8 (prime shape) ·2 (congruence) ·2 (definition of first) ·24 (curve choice) ·3 (twist conditions) ·2 (global/local) ·2 ($p \lesseqgtr |E(\mathbb{F}_p)|$) = 294912 choices.

9.5.10 Example

The artificial "vulnerability" that we have used throughout this chapter, namely BADA55 appearing in a curve coefficient, is obviously incompatible with taking that coefficient to be minimal in the usual ordering. We would be happy to accept the following type of challenge as an alternative: a third party provides us with a nonstructured prime number $n > 2^{50}$; we find a curve so that the hexadecimal representation of ℓ modulo n ends in BAD, a condition having probability 2^{-12}.

9.6 Manipulating security criteria

An unfortunate recent trend is to introduce top performance as a selection requirement. This means that Alice and Bob accept only *the fastest curve*, as demonstrated by benchmarks across a range of platforms. The most widely known example of this approach is Bernstein's Curve25519, the curve $y^2 = x^3 + 486662x^2 + x$ modulo the particularly efficient prime $2^{255} - 19$, which over the past ten years has set speed records for conservative ECC on space-constrained ASICs, Xilinx FPGAs, 8-bit AVR microcontrollers, 16-bit MSP430X microcontrollers, 32-bit ARM Cortex-M0 microcontrollers, larger 32-bit ARM smartphone processors, the Cell processor, NVIDIA and AMD GPUs, and several generations of 32-bit and 64-bit Intel and AMD CPUs, using implementations from 23 authors. See [Ber06; GT07; CS09; BDL$^+$12; BS12; LM13; MC14; SG14; Cho15; DHH$^+$15; HSS$^+$15].

The annoyance for Jerry in this scenario is that, in order to make a case for his curve, he needs to present implementions of the curve arithmetic on a variety of devices, showing that his curve is fastest across platforms. Jerry could try to falsify his speed reports, but it is increasingly common for the public to demand verifiable benchmarks using open-source software.

Jerry can hope that some platforms will favor one curve while other platforms will favor another curve; Jerry can then use arguments for a "reasonable" weighting of platforms as a mechanism to choose one curve or the other. However, it seems difficult to outperform Curve25519 even on *one* platform. The prime $2^{255} - 19$ is particularly efficient, as is the Montgomery curve shape $y^2 = x^3 + 486662x^2 + x$. The same curve is also expressible as a complete Edwards curve, allowing fast additions without the overhead of checking for exceptional cases. Twist security removes the overhead of checking for invalid inputs. Replacing 486662 with a larger curve coefficient produces identical performance on many platforms but loses a measurable amount of performance on some platforms, violating the "top performance" requirement.

In Section 9.5, Jerry was free to, e.g., claim that $p \equiv 3 \pmod 4$ provides "simple square-root computations" and thus replace $2^{255} - 19$ with $2^{255} - 765$; claim that "compatibility" requires curves of the form $y^2 = x^3 - 3x + b$; etc. The new difficulty in this section is that Jerry is facing "top performance" fanatics who reject $2^{255} - 765$ as not providing top performance; who reject $y^2 = x^3 - 3x + b$ as not providing top performance; etc.

Fortunately, Jerry still has some flexibility in defining what security requirements to take into account. Taking "the fastest curve" actually means taking the fastest curve *meeting specified security requirements*, and the list of security requirements is a target of manipulation.

Most importantly, Jerry can argue for any size of ℓ. However, if there is a faster curve with a larger ℓ satisfying the same criteria, then Jerry's curve will be rejected. Furthermore, if Jerry's curve is only marginally larger than a significantly faster curve, then Jerry will have to argue that a tiny difference in security levels (e.g., one curve broken with 0.7× or 0.5× as much effort as another) is meaningful, or else the top-performance fanatics will insist on the significantly faster curve.

The choice of prime has the biggest impact on speed and closely rules the size of ℓ. For pseudo-Mersenne primes larger than 2^{224} the only possibly competitive ones are: $2^{226}-5, 2^{228}+3, 2^{233}-3, 2^{235}-15, 2^{243}-9, 2^{251}-9, 2^{255}-19, 2^{263}+9, 2^{266}-3, 2^{273}+5, 2^{285}-9, 2^{291}-19, 2^{292}+13, 2^{295}+9, 2^{301}+27, 2^{308}+27, 2^{310}+15, 2^{317}+9, 2^{319}+9, 2^{320}+27, 2^{321}-9, 2^{327}+9, 2^{328}+15, 2^{336}-3, 2^{341}+5, 2^{342}+15, 2^{359}+23, 2^{369}-25, 2^{379}-19, 2^{390}+3, 2^{395}+29, 2^{401}-31, 2^{409}+29, 2^{414}-17, 2^{438}+25, 2^{444}-17, 2^{452}-3, 2^{456}+21, 2^{465}+29, 2^{468}-17, 2^{488}-17, 2^{489}-21, 2^{492}+21, 2^{495}-31, 2^{508}+15, 2^{521}-1$. Preliminary implementation work shows that the Mersenne prime $2^{521}-1$ has such efficient reduction that it outperforms, e.g., the prime $2^{512}-569$ from [BCL$^+$15]; perhaps it even outperforms primes below 2^{500}. We would expect implementation work to also show, e.g., that $2^{319}+9$ is significantly faster than $2^{320}+27$, and Jerry will have a hard time arguing for $2^{320}+27$ on security grounds. Considering other classes of primes, such as Montgomery-friendly primes, might identify as many as 100 possibly competitive primes, but it is safe to estimate that fewer than 80 of these primes will satisfy the top-performance fanatics, and further implementation work is likely to reduce the list even more. Note that in this section, unlike other sections, we take a count that is optimistic for Jerry.

Beyond the choice of prime, Jerry can use different choices of security criteria. However, most of the flexibility in Section 9.5 consists of speed claims, compatibility claims, etc., few of which can be sold as security criteria. Jerry *can* use the different twist conditions, the choice whether $p < |E(\mathbb{F}_p)|$ or $p > |E(\mathbb{F}_p)|$, and possibly two choices of cofactor requirements. Jerry can also choose to require completeness as a security criterion, but this does not affect curve choice in this section: the complete formulas for twisted Hessian and Edwards curves are *faster* than the incomplete formulas for Weierstrass curves. The bottom line is that multiplying fewer than 80 primes by 12 choices of security criteria produces fewer than 960 curves. The main difficulty in pinpointing an exact number is carrying out detailed implementation work for each prime; we leave this to future work.

9.7 Afterword: removing the hat

This chapter, outside this section, systematically adopts the attacker's perspective. In this section, to avoid any chance of confusion, we drop the attacker's perspective and address a few questions that we have been asked.

First, in case this is not obvious to the reader, we do not actually endorse the attacker's perspective. Our goal in analyzing the security of systems is to prevent attacks.

Second, this chapter *analyzes* the *possibilities* of backdooring curves under various conditions. We are not making any statements about whether such an attack has actually been carried out.

Third, we have been asked how to eliminate Jerry's flexibility in choosing curves. We are not aware of any proposal that reduces the flexibility to just one curve.

```
import hashlib   # for a PRNG, SHA-1 is standard and sufficiently secure
def hash(seed): h = hashlib.sha1(); h.update(seed); return h.digest()
seedbytes = 20   # 160-bit size for seed, determined by SHA-1 output size

# 224-bit prime p produced by very similar procedure, shown in separate file
p = 0xD7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF
k = GF(p); R.<x> = k[]

def secure(A,B):
  if k(B).is_square(): return False
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n < p and n.is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1)

def int2str(seed,bytes):   # standard big-endian encoding of integer seed
  return ''.join([chr((seed//256^i)%256) for i in reversed(range(bytes))])

def str2int(seed):
  return Integer(seed.encode('hex'),16)

def update(seed):  # add 1 to seed, viewed as integer
  return int2str(str2int(seed) + 1,len(seed))

def fullhash(seed):
  return str2int(hash(seed) + hash(update(seed))) % 2^223

def real2str(seed,bytes): # most significant bits of real number between 0 and 1
  return int2str(Integer(floor(RealField(8*bytes+8)(seed)*256^bytes)),bytes)

nums = real2str(exp(1)/16,7*seedbytes) # enough bits for all curve sizes
S = nums[2*seedbytes:3*seedbytes] # previous bytes are used for 160 and 192
while True:
  A = fullhash(S)
  if not (k(A)*x^4+3).roots(): S = update(S); continue
  S = update(S)
  B = fullhash(S)
  if not secure(A,B): S = update(S); continue
  print 'p',hex(p).upper()
  print 'A',hex(A).upper()
  print 'B',hex(B).upper()
  break

# output:
# p D7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF
# A 2B98B906DC245F2916C03A2F953EA9AE565C3253E8AEC4BFE84C659E
# B 68AEC4BFE84C659EBB8B81DC39355A2EBFA3870D98976FA2F17D2D8D
```

Figure 9.7: An implementation of the Brainpool standard procedure [Bra05, Section 5] to generate a 224-bit curve.

```
import hashlib   # for a PRNG, SHA-1 is standard and sufficiently secure
def hash(seed): h = hashlib.sha1(); h.update(seed); return h.digest()
seedbytes = 20   # 160-bit size for seed, determined by SHA-1 output size

# 224-bit prime p produced by very similar procedure, shown in separate file
p = 0xD7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF
k = GF(p); R.<x> = k[]

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n < p and n.is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1)

def int2str(seed,bytes):   # standard big-endian encoding of integer seed
  return ''.join([chr((seed//256^i)%256) for i in reversed(range(bytes))])

def str2int(seed):
  return Integer(seed.encode('hex'),16)

def update(seed):  # add 1 to seed, viewed as integer
  return int2str(str2int(seed) + 1,len(seed))

def fullhash(seed):
  return str2int(hash(seed) + hash(update(seed))) % 2^223

def real2str(seed,bytes): # most significant bits of real number between 0 and 1
  return int2str(Integer(floor(RealField(8*bytes+8)(seed)*256^bytes)),bytes)

nums = real2str(exp(1)/16,7*seedbytes) # enough bits for all curve sizes
S = nums[2*seedbytes:3*seedbytes] # previous bytes are used for 160 and 192
while True:
  A = fullhash(S)
  if not (k(A)*x^4+3).roots(): S = update(S); continue
  while True:
    S = update(S)
    B = fullhash(S)
    if not k(B).is_square(): break
  if not secure(A,B): S = update(S); continue
  print 'p',hex(p).upper()
  print 'A',hex(A).upper()
  print 'B',hex(B).upper()
  break

# output:
# p D7C134AA264366862A18302575D1D787B09F075797DA89F57EC8C0FF
# A 68A5E62CA9CE6C1C299803A6C1530B514E182AD8B0042A59CAD29F43
# B 2580F63CCFE44138870713B1A92369E33E2135D266DBB372386C400B
```

Figure 9.8: An implementation of a procedure that, unlike Figure 9.7, actually generates the `brainpool224r1` curve.

```
import sys

import hashlib   # for a PRNG, SHA-1 is standard and sufficiently secure
def hash(seed): h = hashlib.sha1(); h.update(seed); return h.digest()
seedbytes = 20   # 160-bit size for seed, determined by SHA-1 output size

def int2str(seed,bytes):   # standard big-endian encoding of integer seed
  return ’’.join([chr((seed//256^i)%256) for i in reversed(range(bytes))])

def str2int(seed):
  return Integer(seed.encode(’hex’),16)

def update(seed):  # add 1 to seed, viewed as integer
  return int2str(str2int(seed) + 1,len(seed))

def real2str(seed,bytes): # most significant bits of real number between 0 and 1
  return int2str(Integer(floor(RealField(8*bytes+8)(seed)*256^bytes)),bytes)

sizes = [160,192,224,256,320,384,512]
S = real2str(pi/16,len(sizes)*seedbytes)
primeseeds = [S[i:i+seedbytes] for i in range(0,len(S),seedbytes)]
S = real2str(exp(1)/16,len(sizes)*seedbytes)
curveseeds = [S[i:i+seedbytes] for i in range(0,len(S),seedbytes)]

for j in range(len(sizes)):
  L,S = sizes[j],primeseeds[j]
  v = (L-1)//160

  def fullhash(seed,bits):
    h = hash(seed)
    for i in range(v): seed = update(seed); h += hash(seed)
    return str2int(h) % 2^bits
```

Figure 9.9: Part 1 of 2: A complete procedure to generate the Brainpool standard curves. Continued in Figure 9.10.

```
while True:
  p = fullhash(S,L)
  while not (p % 4 == 3 and p.is_prime()): p += 1
  if 2^(L-1) - 1 < p and p < 2^L: break
  S = update(S)

k = GF(p)
R.<x> = k[]

def secure(A,B):
  E = EllipticCurve([k(A),k(B)])
  for q in [2,3,5,7]:
    # quick check whether q divides n, without computing n
    for r,e in E.division_polynomial(q).roots():
      if E.is_x_coord(r): return False
  n = E.cardinality()
  return (n < p and n.is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1)

S = curveseeds[j]
while True:
  A = fullhash(S,L-1)
  if not (k(A)*x^4+3).roots(): S = update(S); continue
  while True:
    S = update(S)
    B = fullhash(S,L-1)
    if not k(B).is_square(): break
  if not secure(A,B): S = update(S); continue
  print ’p’,hex(p).upper()
  print ’A’,hex(A).upper()
  print ’B’,hex(B).upper()
  sys.stdout.flush()
  break
```

Figure 9.10: Part 2 of 2: A complete procedure to generate the Brainpool standard curves. Continued from Figure 9.9.

```
import simplesha3
hash = simplesha3.sha3512   # SHA-3 standard at maximum security level

p = 2^224 - 2^96 + 1   # standard NIST P-224 prime
k = GF(p)
seedbytes = 20   # standard 160-bit size for seed

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n.is_prime() and (2*p+2-n).is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1
    and Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1)

def int2str(seed,bytes):   # standard big-endian encoding of integer seed
  return ''.join([chr((seed//256^i)%256) for i in reversed(range(bytes))])

def str2int(seed):
  return Integer(seed.encode('hex'),16)

def complement(seed):   # change all bits, eliminating Brainpool-type collisions
  return ''.join([chr(255-ord(s)) for s in seed])

def real2str(seed,bytes): # most significant bits of real number between 0 and 1
  return int2str(Integer(RealField(8*bytes)(seed)*256^bytes),bytes)

sizeofint = 4   # number of bytes in a 32-bit integer
nums = real2str(cos(1),seedbytes - sizeofint)
for counter in xrange(0,256^sizeofint):
  S = int2str(counter,sizeofint) + nums
  T = complement(S)
  A = str2int(hash(S))
  B = str2int(hash(T))
  if secure(A,B):
    print 'p',hex(p).upper()
    print 'A',hex(A).upper()
    print 'B',hex(B).upper()
    break

# output:
# p FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF000000000000000000000001
# A 7144BA12CE8A0C3BEFA053EDBADA555A42391FC64F052376E041C7D4AF23195EBD8D83625321D452E8A0C3BB0
#   A048A26115704E45DCEB346A9F4BD9741D14D49
# B 5C32EC7FC48CE1802D9B70DBC3FA574EAF015FCE4E99B43EBE3468D6EFB2276BA3669AFF6FFC0F4C6AE4AE2E5
#   D74C3C0AF97DCE17147688DDA89E734B56944A2
```

Figure 9.11: A procedure to generate the new “verifiably pseudorandom” BADA55-VPR-224 curve. Compare Figure 9.8.

```
import simplesha3   # Keccak, the SHA-3 winner
hash = simplesha3.keccakc1024   # maximum security level: 512-bit output
seedbytes = 64   # maximum-security 512-bit seed, same size as output

p = 2^224 - 2^96 + 1   # standard NIST P-224 prime
k = GF(p)

def secure(A,B):
  n = EllipticCurve([k(A),k(B)]).cardinality()
  return (n.is_prime() and (2*p+2-n).is_prime()
    and Integers(n)(p).multiplicative_order() * 100 >= n-1
    and Integers(2*p+2-n)(p).multiplicative_order() * 100 >= 2*p+2-n-1)

def int2str(seed,bytes):   # standard little-endian encoding of integer seed
  return ''.join([chr((seed//256^i)%256) for i in range(bytes)])

def str2int(seed):
  return sum([ord(seed[i])*256^i for i in range(len(seed))])

def rotate(seed):  # rotate seed by 1 bit, eliminating Brainpool-like collisions
  x = str2int(seed)
  x = 2*x + (x >> (8*len(seed)-1))
  return int2str(x,len(seed))

def real2str(seed,bytes): # most significant bits of real number between 0 and 1
  return int2str(Integer(floor(RealField(8*bytes+8)(seed)*256^bytes)),bytes)

counterbytes = 3   # minimum number of bytes needed to guarantee success
nums = real2str(exp(1)/4,seedbytes - counterbytes)
for counter in xrange(0,256^counterbytes):
  S = int2str(counter,counterbytes) + nums
  R = rotate(S)
  A = str2int(hash(R))
  B = str2int(hash(S))
  if secure(A,B):
    print 'p',hex(p).upper()
    print 'A',hex(A).upper()
    print 'B',hex(B).upper()
    break

# output:
# p FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF000000000000000000000001
# A 8F0FF20E1E3CF4905D492E04110683948BFC236790BBB59E6E6B33F24F348ED2E16C64EE79F9FD27E9A367FF6
#   415B41189E4FB6BADA555455DC44C4F87011EEF
# B  E85067A95547E30661C854A43ED80F36289043FFC73DA78A97E37FB96A2717009088656B948865A660FF3959
#   330D8A1CA1E4DE31B7B7D496A4CDE555E57D05C
```

Figure 9.12: A procedure to generate the new "verifiably pseudorandom" BADA55-VPR2-224 curve. Compare Figure 9.11.

Part IV

Multivariate System Solving with XL

10 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]. 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].

We describe a parallel implementation of an algorithm for solving quadratic systems that was first suggested by Lazard in [Laz83]. Later it was reinvented by Courtois, Klimov, Patarin, and Shamir and published in [CKP$^+$00]; they call the algorithm *XL* as an acronym for *extended linearization*: XL *extends* a quadratic system by multiplying all equations 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 in the computer algebra system Magma, and is often used as standard benchmark by cryptographers.

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 for generic systems, we describe 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 clusters of up to 8 nodes and shared-memory systems of up to 64 cores. A previous implementation of XL is PWXL, a parallel implementation of XL with block Wiedemann described in [MDK⁺10]. PWXL supports onl $\mathbb{F}_2$, while our implementation supports $\mathbb{F}_2$, $\mathbb{F}_{16}$, and $\mathbb{F}_{31}$. Furthermore, our implementation is modular and can be extended to other fields. Comparisons on performance of PWXL and our work will be shown in Sec. 10.5.3. We are planning to make our implementation available to the public.

This chapter is structured as follows: The XL algorithm is introduced in Sec. 10.1. The parallel implementation of XL using the block Wiedemann algorithm is described in Sec. 10.4. Section 10.5 gives runtime measurements and performance values that are achieved by our implementation for a set of parameters on several parallel systems as well as comparisons to PWXL and to the implementation of F_4 in Magma.

10.1 The XL algorithm

The original description of XL for multivariate quadratic systems can be found in the paper [CKP⁺00]; a more general definition of XL for systems of higher degree is given in [Cou03]. The following gives an 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 $D \in \mathbb{N}$ as $D := \min\{d : ((1-\lambda)^{m-n-1}(1+\lambda)^m)[d] \leq 0\}$ (see [YC05, Eq. (7)], [Moh01; Die04]), where $f[i]$ denotes the coefficient of the degree-i term in the expansion of a polynomial $f(\lambda)$ e.g., $(\lambda+2)^3[2] = (\lambda^3+6\lambda^2+12\lambda+8)[2] = 6$. XL extends the quadratic 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 matrix of $\mathcal{M}$ 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, giving solutions for all monomials and particularly for $x_1, x_2, \ldots, x_n$. Note that the matrix corresponding to the linear system $\mathcal{M}$ is the Macaulay matrix of degree D for the polynomial system $\mathcal{A}$ (see [Mac16], e.g., defined in [FPP⁺12]).

10.1.1 The Block Wiedemann algorithm

The computationally most expensive task in XL is to find a solution for the sparse linear system $\mathcal{M}$ 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. The following paragraphs give a brief introduction to the block Wiedemann algorithm. For more details please refer to [Nie12, Sec. 4.2] and [Cop94].

The basic idea of Coppersmith's block Wiedemann algorithm for finding a solution $\bar{x} \neq 0$ of $B\bar{x} = 0$ for $B \in K^{N \times N}$, $\bar{x} \in K^N$ (where B corresponds to the system matrix of $\mathcal{M}$ when computing XL) is the same as in the original Wiedemann algorithm: Assume that the characteristic polynomial $f(\lambda) = \sum_{0 \leq i} f[i]\lambda^i$ of B is known. Since B is singular, it has an eigenvalue 0, thus $f(B) = 0$ and $f[0] = 0$. We have:

$$f(B)\bar{z} = \sum_{i>0} f[i]B^i\bar{z} = B\sum_{i>0} f[i]B^{i-1}\bar{z} = 0,$$

for any vector $\bar{z} \in K^N$. Therefore, $\bar{x} = \sum_{i>0} f[i]B^{i-1}\bar{z}$, $\bar{z} \neq 0$ is a (hopefully nonzero) kernel vector and thus a solution of the linear equation system. In fact it is possible to use any *annihilating* polynomial $f(\lambda)$ of B, i.e., a polynomial $f(\lambda) \neq 0$ such that $f(B) = 0$.

Wiedemann suggests to use the Berlekamp–Massey algorithm for the computation of $f(\lambda)$. Given a linearly recurrent sequence $\{a^{(i)}\}_{i=0}^{\infty}$, the algorithm computes $c_1, \ldots, c_d$ for some d such that $c_1a^{(d-1)} + c_2a^{(d-2)} + \cdots + c_da^{(0)} = 0$. Choosing $a^{(i)} = \bar{x}^TBB^i\bar{z}$ with random vectors $\bar{x}$ and $\bar{z}$ (as delegates for BB^i) as input and $f[i] = c_{d-i}$, $0 \leq i < d$ as output returns $f(\lambda)$ as an annihilating polynomial of B with high probability.

Coppersmith [Cop94] proposed a modification of the Wiedemann algorithm that makes it more suitable for modern computer architectures by operating in parallel on a *block* of $\tilde{n}$ column vectors $\bar{z}_i, 0 \leq i < \tilde{n}$, of a matrix $z \in K^{N \times \tilde{n}}$. His block Wiedemann algorithm computes kernel vectors in three steps which are called BW1, BW2, and BW3 for the remainder of this chapter. The *block sizes* of the block Wiedemann algorithm are the integers $\tilde{m}$ and $\tilde{n}$. They can be chosen freely for the implementation such that they give the best performance on the target architecture for matrix and vector operations, e.g., depending on the size of cache lines or vector registers. Step BW1 computes the first $N/\tilde{m} + N/\tilde{n} + (1)$ elements of a sequence $\{a^{(i)}\}_{i=0}^{\infty}$, $a_i = \left(x \cdot (B \cdot B^iz)\right)^T \in K^{\tilde{n} \times \tilde{m}}$ using random matrices $x \in K^{\tilde{m} \times N}$ and $z \in K^{N \times \tilde{n}}$. This sequence is the input for the second step BW2, a block variant of the Berlekamp–Massey algorithm. It returns a matrix polynomial $f(\lambda)$ with coefficients $f[j] \in K^{\tilde{n} \times \tilde{n}}$, that is used by step BW3 to compute up to $\tilde{n}$ solution vectors in a blocked fashion similar as described above for the original Wiedemann algorithm.

10.2 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.

Algorithm 3 Gaussian elimination in Coppersmith's Berlekamp–Massey algorithm

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

10.2.1 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 3. 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.

10.2.2 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.

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 10.1 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 10.1, this

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 10.1: 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.

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 10.1. This case occurs if in iteration $j+1$ no coefficient is reassigned to node $i-1$ due to load balancing.

- In case $k = \min(S_i^{(j)}) = \max(S_{i-1}^{(j+1)})$, $i \neq 1$, the top n rows of $(P^{(j)}f^{(j)})[k]$ are sent from node i to node $i-1$. The example in Figure 10.1 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.

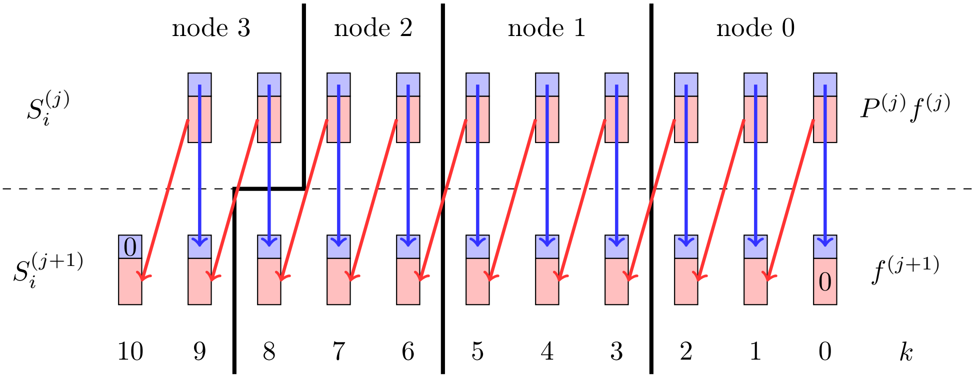

Figure 10.1: Example for the communication between 4 nodes. The top n rows of the coefficients are colored in blue, the bottom m rows are colored in red.

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.

10.3 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.

10.3.1 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.

10.3.2 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.

10.4 Implementation of XL

Stage BW1 of the block Wiedemann algorithm computes $a^{(i)} = \big(x \cdot (B \cdot B^i z)\big)^T$, $0 \leq i \leq N/\tilde{m} + N/\tilde{n} + (1)$. We do this efficiently using two sparse-matrix multiplications by making the random matrices x and z deliberately sparse. We compute a sequence $\{t^{(i)}\}_{i=0}^{\infty}$ of matrices $t^{(i)} \in K^{N \times n}$ defined as

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

Thus, $a^{(i)}$ can be computed as $a^{(i)} = (xt^{(i)})^T$. In step BW3 we evaluate the annihilating polynomial $f(\lambda)$ by applying Horner's scheme, again using two sparse-matrix multiplications by computing

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

For details on the steps BW1, BW2, and BW3 please refer to [Nie12, Sec. 4.2].

Assuming that $\tilde{m} = c{\cdot}\tilde{n}$ for some constant $c \geq 1$, the asymptotic time complexity of step BW1 and BW2 can be written as $\big(N^2 \cdot w_B\big)$, where w_B is the average number of nonzero entries per row of B. Note that BW3 actually requires about half of the time of BW1 since it requires only about halve as many iterations. The asymptotic time complexity of Coppersmith's version of the Berlekamp–Massey algorithm in step BW2 is $\big(N^2 \cdot \tilde{n}\big)$. Thomé presents an improved version of Coppersmith's block Berlekamp–Massey algorithm in [Tho02]. Thomé's version is asymptotically faster: It reduces the complexity of BW2 from $(N^2 \cdot \tilde{n})$ to $\big(N \cdot \log^2(N) \cdot \tilde{n}\big)$. The subquadratic complexity is achieved by converting the block Berlekamp–Massey algorithm into a recursive divide-and-conquer process.

Since steps BW1 and BW3 have a higher asymptotic time complexity than Thomé's version of step BW2, we do not describe our implementation, optimization, and parallelization of Coppersmith's and Thomé's versions of step BW2 in detail in this chapter for the sake of brevity. The interested reader is referred to [Nie12, Chap. 4] for details. However, we discuss the performance of our implementations in Sec. 10.5.

Since the system matrix $\mathcal{M}$ has more rows than columns, some rows must be dropped randomly to obtain a square matrix B. Observe that due to the extension step of XL the entries of the original quadratic system $\mathcal{A}$ appear repeatedly in the matrix B at well-defined positions based on the enumeration scheme. Therefore, it is possible to generate the entries of B on demand spending a negligible amount of memory. However, the computation of the entry positions requires additional time; to avoid this computational overhead, we store the Macaulay matrix B in a compact memory format (see [Nie12, Sec. 4.5.3]). This gives a significant speedup in the computation time—given that the matrix B fits into available memory.

10.4.1 Efficient matrix multiplication

All matrix multiplications of the shape $D = EF$ that we perform during XL are either multiplications of a sparse matrix by a dense matrix, or multiplications of a dense matrix by a dense matrix with matrices of a small size. For these cases, schoolbook multiplication is more efficient than the *asymptotically* more efficient Strassen algorithm or the Coppersmith–Winograd algorithm.

However, when computing in finite fields, the cost of matrix multiplications can be significantly reduced by trading expensive multiplications for cheap additions—if the field size is significantly larger than the row weight of E. This is the case for small fields like, for example, $\mathbb{F}_{16}$ or $\mathbb{F}_{31}$. We reduce the number of actual multiplications for a row r of E by summing up all row vectors of F which are to be multiplied by the same field element and performing the multiplication on all of them together. A temporary buffer $b_\alpha \in K^n, \alpha \in K$ of vectors of length n is used to collect the sum of row vectors that ought to be multiplied by α. For all entries $E_{r,c}$, row c of F is added to $b_{E_{r,c}}$. Finally, b can be reduced by computing $\sum \alpha \cdot b_\alpha, \alpha \neq 0, \alpha \in K$, which gives the result for row r of the matrix D.

With the strategy explained so far, computing the result for one row of E takes $w_E + |K| - 2$ additions and $|K| - 2$ scalar multiplications (there is no need for the multiplication by 0 and 1, and for the addition of 0). The number of actual multiplications can be further reduced by exploiting the distributivity of the scalar multiplication of vectors: Assume in the following that $K = \mathbb{F}_{p^k} = \mathbb{F}_p[x]/(f(x))$, with p prime and $f(x)$ an irreducible polynomial with $\deg(f) = k$. When $k = 1$, the natural mapping from K to $\{0, 1, \ldots, p-1\} \subset \mathbb{N}$ induces an order of the elements. The order can be extended for $k > 1$ by $\forall \beta, \gamma \in K : \beta > \gamma \iff \beta[i] > \gamma[i], i = \max(\{j : \beta[j] \neq \gamma[j]\})$. We decompose each scalar factor $\alpha \in K \setminus \{0, 1, x^1, \ldots, x^{k-1}\}$ of a multiplication $\alpha \cdot b_\alpha$ into two components $\beta, \gamma \in K$ such that $\beta, \gamma < \alpha$ and $\beta + \gamma = \alpha$. Starting with the largest α, iteratively add b_α to b_β and b_γ and drop buffer b_α. The algorithm terminates when all buffers b_α, $\alpha \in K \setminus \{0, 1, x^1, \ldots, x^{k-1}\}$ have been dropped. Finally, the remaining buffers b_α, $\alpha \in \{1, x^1, \ldots, x^{k-1}\}$ are multiplied by their respective scalar factor (except b_1) and summed up to the final result. This reduces the number of multiplications to $k-1$. All in all the computation on one row of E (with row weight w_E) costs $w_E + 2(|K| - k - 1) + k - 1$ additions and $k - 1$ scalar multiplications. For example the computations in $\mathbb{F}_{16}$ require $w_E + 25$ additions and 3 multiplications per row of a matrix E.

10.4.2 Parallel Macaulay matrix multiplication

The most expensive part in the computation of steps BW1 and BW3 of XL is a repetitive multiplication of the shape $t_{new} = B \cdot t_{old}$, where $t_{new}, t_{old} \in K^{N \times \tilde{n}}$ are dense matrices and $B \in K^{N \times N}$ is a sparse Macaulay matrix with an average row weight w_B.

For generic systems, the Macaulay matrix B has an expected number of non-zero entries per row of $(|K| - 1)/|K| \cdot \binom{n+2}{2}$. However, in our memory efficient data format for the Macaulay matrix we also store the zero entries from the original system. This results in a fixed row weight $w_B = |K| \cdot \binom{n+2}{2}$. This is highly efficient in terms of memory consumption and computation time for $\mathbb{F}_{16}$, $\mathbb{F}_{31}$, and larger fields (see [Nie12, Chap. 4]). Since there is a guaranteed number of entries per row (i.e. the row weight w_B) we compute the Macaulay matrix multiplication in row order in a big loop over all row indices as described in the previous section.

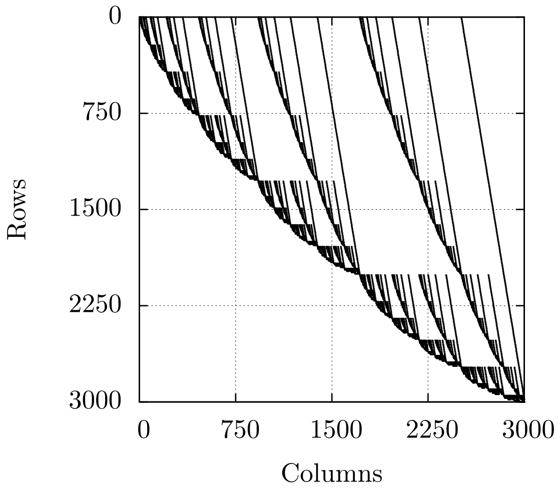

Figure 10.2: Plot of a Macaulay matrix for a system with 8 variables, 10 equations, using graded reverse lexicographical (grevlex) monomial order.

The parallelization of the Macaulay matrix multiplication of steps BW1 and BW3 is implemented in two ways: On multi-core architectures OpenMP is used to keep all cores busy; on cluster architectures the Message Passing Interface (MPI) and InfiniBand verbs are used to communicate between the cluster nodes. Both approaches can be combined for clusters of multi-core nodes.

The strategy of the workload distribution is similar on both multi-core systems and cluster systems. Figure 10.2 shows an example of a Macaulay matrix. Our approach for efficient matrix multiplications (described in the previous section) trades multiplications for additions. The approach is most efficient, if the original number of scalar multiplications per row is much higher than the order of the field. Since the row weight of the Macaulay matrix is quite small, splitting the rows between computing nodes reduces the efficiency of our approach. Therefore, the workload is distributed by assigning blocks of rows of the Macaulay matrix to the computing units.

Parallelization for Shared-Memory Systems:

We parallelize the data-independent loop over the rows of the Macaulay matrix using OpenMP with the directive "`#pragma omp parallel for`". The OpenMP parallelization on UMA systems encounters no additional communication cost although the pressure on shared caches may be increased. On NUMA systems the best performance is achieved if the data is distributed over the NUMA nodes in a way that takes the higher cost of remote memory access into account. However, the access pattern to t_{old} is very irregular due to the structure of the Macaulay matrix: In particular, the access pattern of each core does not necessarily fully cover memory pages. Furthermore, the same memory page is usually touched by several cores. The same is true for

t_{new}, since after each iteration t_{new} and t_{old} are swapped by switching their respective memory regions. Therefore, we obtained the shortest runtime by distributing the memory pages interleaved (in a round-robin fashion) over the nodes.

Parallelization for Cluster Systems:

The computation on one row of the Macaulay matrix depends on many rows of the matrix t_{old}. A straightforward approach is to make the full matrix t_{old} available on all cluster nodes. This can be achieved by an all-to-all communication step after each iteration of 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, allowing the CPU to continue computations alongside communication. It is possible to split the computation of t_{new} into two column blocks; during computation on one block, 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 bounded by communication cost.

Apart from hiding the communication overhead it is also possible to totally avoid all communication by splitting t_{old} and t_{new} into independent column blocks for each cluster node. However, splitting t_{old} and t_{new} has an impact either on the runtime of BW1 and BW3 (if the block size becomes too small for efficient computation) or on the runtime of BW2 (since the block size has a strong impact on its runtime and memory demand).

We implemented both approaches since they can be combined to give best performance on a target system architecture. The following paragraphs explain the two approaches in detail:

a) Operating on Two Shared Column Blocks of t_{old} and t_{new}: For this approach, the matrices t_{old} and t_{new} are split 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 mentioned 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.

However, looking at the structure of the Macaulay matrix (an example is shown in Fig. 10.2) 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 matrices $t_{new,0}$ 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.

This communication pattern requires to send several data blocks to individual cluster nodes in parallel to ongoing computation. This can not be done efficiently using MPI. Therefore, we circumvent the MPI API and program the network hardware directly. Our implementation uses an InfiniBand network; the same approach can be used for other high-performance networks. We access the InfiniBand hardware 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 reduces communication to the smallest amount possible for the cost of only a negligibly small initialization overhead.

This approach of splitting t_{old} and t_{new} into two shared column blocks has the disadvantage 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 when working on a single column block. However, these memory accesses are sequential. It is therefore likely that the access pattern can be detected by the memory logic and that the data is prefetched into the caches.

b) Operating on Independent Column Blocks of t_{old} and t_{new}: Any communication during steps 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 steps.

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,

Table 10.2: Computer architectures used for the experiments

	NUMA	Cluster
CPU		
Name	AMD Opteron 6276	Intel Xeon E5620
Microarchitecture	Bulldozer Interlagos	Nehalem
Frequency	2300 MHz	2400 MHz
Number of CPUs per socket	2	1
Number of cores per socket	16 (2 x 8)	4
Level 1 data-cache size	16 × 48 KB	4 × 32 KB
Level 2 data-cache size	8 × 2 MB	4 × 256 KB
Level 3 data-cache size	2 × 8 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	64	64
Network interconnect	—	InfiniBand MT26428 2 ports of 4×QDR, 32 Gbit/s
Memory		
Memory per CPU	32 GB	18 GB
Memory per cluster node	—	36 GB
Total memory	256 GB	288 GB

the efficiency of the parallelization declines significantly. Another disadvantage of this approach is that since the nodes compute on the whole Macaulay matrix, all nodes must store the whole matrix in their memory. For large systems this is may not be feasible.

Both approaches for parallelization have advantages and disadvantages; the ideal approach can only be found by testing each approach on the target hardware. For small clusters approach b) 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) depends heavily on the network configuration and the ratio between computation time and communication time. Both approaches can be combined by splitting the cluster into independent partitions; the workload is distributed over the partitions using approach b) and over the nodes within one partition using approach a).

10.5 Experimental results

This section gives an overview of the performance and the scalability of our XL implementation for generic systems. Experiments have been carried out on two computer systems: a 64-core NUMA system and an eight node InfiniBand cluster. Table 10.2 lists the key features of these systems.

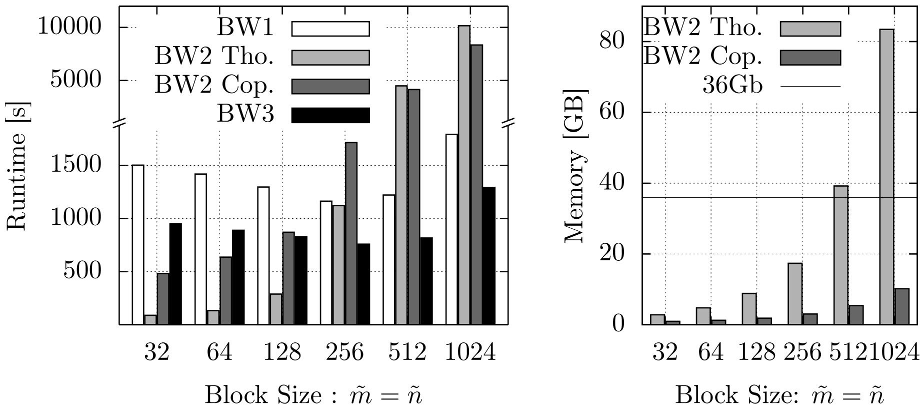

Figure 10.3: Runtime and memory consumption of XL 16-14 over different block sizes on a single cluster node with two CPUs (8 cores in total) and 36 GB RAM.

10.5.1 Impact of the block size

We measured the impact of the block size of the block Wiedemann algorithm on the performance of the implementation on a single cluster node (without cluster communication). We used a quadratic system with 16 equations and 14 variables over $\mathbb{F}_{16}$. In this case, the degree D for the linearization is 9. The input for the algorithm is a Macaulay matrix B with $N = 817190$ rows (and columns) and row weight $w_B = 120$. To reduce the parameter space, we fix $\tilde{m}$ to $\tilde{m} = \tilde{n}$.

Figure 10.3 shows the runtime for block sizes 32, 64, 128, 256, 512, and 1024. Given the fixed size of the Macaulay matrix and $\tilde{m} = \tilde{n}$, the number of field operations for BW1 and BW2 is roughly the same for different choices of the block size $\tilde{n}$ since the number of iterations is proportional to $1/\tilde{n}$ and number of field operations per iteration is roughly proportional to $\tilde{n}$. However, the runtime of the computation varies depending on $\tilde{n}$.

During the i-th iteration step of BW1 and BW3, the Macaulay matrix is multiplied with a matrix $t^{(i-1)} \in \mathbb{F}_{16}^{N \times \tilde{n}}$. For $\mathbb{F}_{16}$ each row of $t^{(i-1)}$ requires $\tilde{n}/2$ bytes of memory. In the cases $\tilde{m} = \tilde{n} = 32$ and $\tilde{m} = \tilde{n} = 64$ each row thus occupies less than one cache line of 64 bytes. This explains why the best performance in BW1 and BW3 is achieved for larger values of $\tilde{n}$. The runtime of BW1 and BW3 is minimal for block sizes $\tilde{m} = \tilde{n} = 256$. In this case one row of $t^{(i-1)}$ occupies two cache lines. The reason why this case gives a better performance than $\tilde{m} = \tilde{n} = 128$ might be that the memory controller is able to prefetch the second cache line. For larger values of $\tilde{m}$ and $\tilde{n}$ 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 $\tilde{n}$. However, this turns out to be the case only for moderate sizes of $\tilde{n}$; note the different scale of the graph in Fig. 10.3 for a runtime of more than 2000 seconds. For $\tilde{m} = \tilde{n} = 256$ the runtime of Coppersmith's version of BW2 is already larger than that of BW1 and BW3, for $\tilde{m} = \tilde{n} = 512$ and $\tilde{m} = \tilde{n} = 1024$ both versions of BW2 dominate the total runtime of the computation. Thomé's version is faster than Coppersmith's version for small and moderate block sizes. However, by doubling the block size, the memory demand of BW2 roughly doubles as well; Figure 10.3 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 $\tilde{m} = \tilde{n} = 512$ 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.

10.5.2 Scalability experiments

The scalability was measured using a quadratic system with 18 equations and 16 variables over $\mathbb{F}_{16}$. The degree D for this system is 10. The Macaulay matrix B has a size of $N = 5\,311\,735$ rows and columns; the row weight is $w_B = 153$. Since this experiment is not concerned about peak performance but about scalability, a block size of $\tilde{m} = \tilde{n} = 256$ is used. 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. The approach a) (two shared column blocks) was used on the cluster system for all node counts.

Given the runtime T_1 for one computing node and T_p for p computing nodes, the parallel efficiency E_p on the p nodes is defined as $E_p = T_1/pT_p$. Figure 10.4 shows 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 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 systems it is even smaller due to its smaller asymptotic time complexity compared to steps BW1 and BW3. Thus, a lower scalability than BW1 and BW3 can be tolerated for BW2.

For this problem size, our parallel implementation of BW1 and BW3 scales very well for up to eight nodes. However, at some point the communication time is going to catch up with computation time: The computation time roughly halves with every doubling of the number of cluster nodes, while the communication demand per node shrinks with a smaller slope. Therefore, at a certain number of nodes communication time and computation time are about the same and the parallel efficiency declines for any larger number of nodes. We do not have access to a cluster with a fast network interconnect and a sufficient amount of nodes to measure when this point is reached, thus we can only give an estimation: Figure 10.5 shows the expected time of computation and communication for larger cluster sizes. We computed the amount of data that an individual node sends and receives depending on the number of computing nodes. We use the maximum of the outgoing data for the estimation of the communication time. For this particular problem size, we expect that for a cluster of around 16 nodes communication time is about as long as computation time

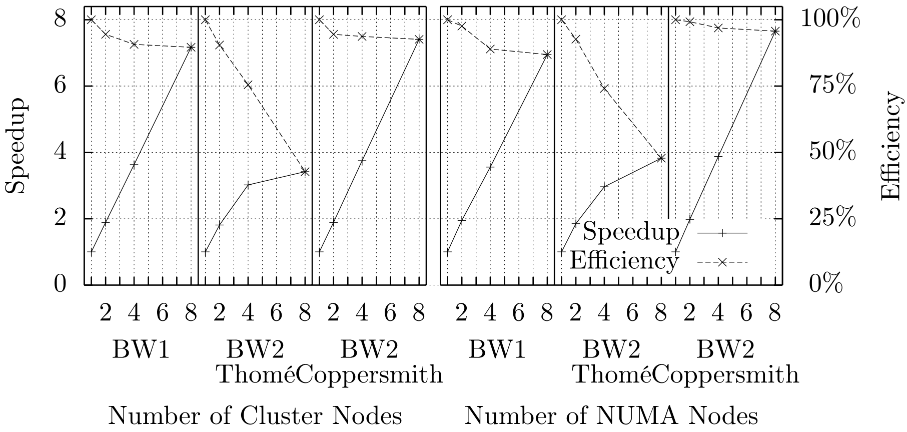

Figure 10.4: Speedup and efficiency of BW1 and BW2

and that the parallel efficiency is going to decline for larger clusters.

On the NUMA system, the scalability is similar to the cluster system. BW1 achieves an efficiency of over 85% 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 (16 threads) the implementation achieves a high efficiency of over 95% since a memory controller on the same socket is used for remote memory access and the remote memory access has only moderate cost. When using more than one NUMA node, the efficiency declines to around 85% 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 50% 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 overhead for communication is sufficiently small that it does not have much impact on the parallel efficiency of up to 8 NUMA nodes.

Our experiments show that the shape of the Macaulay matrix has a large impact on the performance and the scalability of XL. Currently, we are using graded reverse lexicographical order for the Macaulay matrix. However, as opposed to Gröbner basis solvers like F_4 and F_5, for XL there is no algorithmic or mathematic requirement for any particular ordering. In our upcoming research, we are going to examine if another monomial order or a redistribution of columns and rows of the Macaulay matrix has a positive impact on the performance of our implementation.

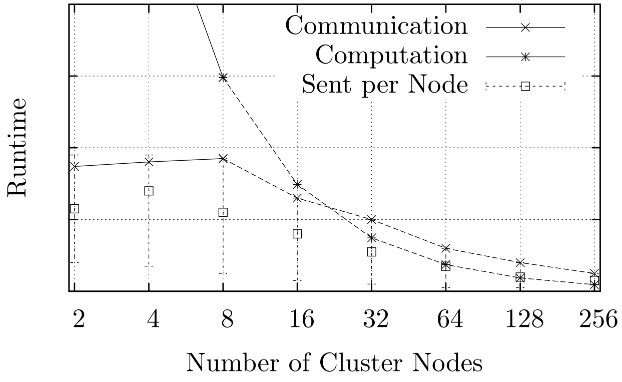

Figure 10.5: Estimation of computation time vs. communication time on a cluster system. The numbers for 2, 4, and 8 nodes are measurements, the numbers for larger cluster sizes are estimations. The amount of data sent per node varies; we show the maximum, minimum, and average.

10.5.3 Comparison with PWXL and Magma F_4

To put our numbers into context, we compare our work with two other Gröbner basis solvers in this section: with PWXL, a parallel implementation of XL with block Wiedemann for $\mathbb{F}_2$ described in [MDK+10], and with the implementation of Faugère's F_4 algorithm [Fau99] in the computational algebra system Magma.

Comparison with PWXL:

Figure 10.6 compares the runtime of PWXL and our implementation for systems in $\mathbb{F}_2$ with $m = n$. We ran our XL implementation on our cluster system (see Table 10.2) while PWXL was running on a machine with four six-core AMD Opteron 8435 CPUs, running at 2.6 GHz.

Our implementation outperforms PWXL for the largest cases given in the paper, e.g., for $n = 33$ our implementation is 24 times faster running on 8 cluster nodes (64 CPU cores) and still 6 times faster when scaling to 16 CPU cores. This significant speedup may be explained by the fact that PWXL is a modification of the block-Wiedemann solver for factoring RSA-768 used in [KAF+10]. Therefore, the code may not be well optimized for the structure of Macaulay matrices. However, these numbers show that our implementation achieves high performance for computations in $\mathbb{F}_2$.

Comparison with F_4:

Figure 10.7 compares time and memory consumption of the F_4 implementation in Magma V2.17-12 and our implementation of XL for systems in $\mathbb{F}_{16}$ with $m = 2n$. When solving the systems in Magma we coerce the systems into $\mathbb{F}_{256}$, because for $\mathbb{F}_{256}$ Magma performs faster than when using $\mathbb{F}_{16}$ directly. The computer used to run F_4 has an 8 core Xeon X7550 CPU running at 2.0 GHz; however, F_4 uses only one core

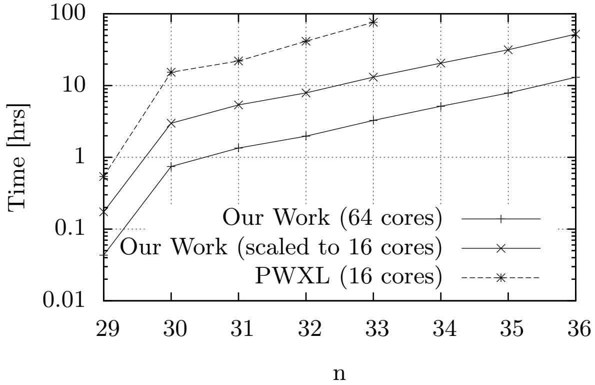

Figure 10.6: Comparison of the runtime of our work and PWXL, $m = n$, $\mathbb{F}_2$

of it. We ran XL on our NUMA system using all 64 CPU cores. For this comparison we use Coppersmith's version of BW2 since it is more memory efficient than Thomé's version.

Note that there is a jump in the graph when going from $n = 21$ to $n = 22$ for XL our implementation, similarly when going from $n = 23$ to $n = 24$ for F_4. This is due to an increment of the degree D from 5 to 6, which happens earlier for XL. Therefore, F_4 takes advantage of a lower degree in cases such as $n = 22, 23$. Other XL-based algorithms like Mutant-XL [MMD$^+$08] may be able to fill this gap. In this chapter we omit a discussion of the difference between the degrees of XL and $\mathrm{F}_4/\mathrm{F}_5$. However, in cases where the degrees are the same for both algorithms, our implementation of XL is better in terms of runtime and memory consumption.

For $n = 25$, the memory consumption of XL is less than 2% of that of F_4. In this case, XL runs 338 times faster on 64 cores than F_4 on one single core, which means XL is still faster when the runtime is normalized to single-core performance by multiplying the runtime by 64.

10.5.4 Performance for computation on large systems

Table 10.3 presents detailed statistics of some of the largest systems we are able to solve in a moderate amount of time (within at most one week). In the tables the time (BW1, BW2, BW3, and total) is measured in seconds, and the memory is measured in GB. Note that for the cluster we give the memory usage for a single cluster node. While all the fields that we have implemented so far are presented in the table, we point out that the most optimization has been done for $\mathbb{F}_{16}$.

The system with $n = 32$ variables and $m = 64$ equations over $\mathbb{F}_{16}$ listed in Table 10.3 is the largest case we have tested. The system was solved in 5 days on the cluster using block sizes $\tilde{m} = 256$ and $\tilde{n} = 128$. With $n = 32$ and $D = 7$ we have $N = \binom{n+D}{D} = \binom{32+7}{7} = 15\,380\,937$ and $w_B = \binom{n+2}{2} = \binom{32+2}{2} = 561$. There are roughly $N/\tilde{n} + N/\tilde{m}$ iterations in BW1 and $N/\tilde{n}$ iterations in BW3. This leads to $2N/\tilde{n} + N/\tilde{m}$ Macaulay matrix multiplications, each takes about $N \cdot (w_B + 25) \cdot \tilde{n}$

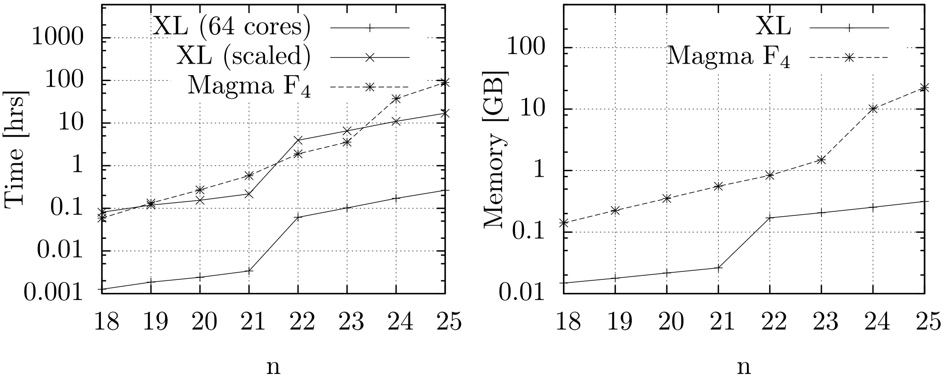

Figure 10.7: Comparison of runtime and memory demand of our implementation of XL and Magma's implementation of F$_4$, $m = 2n$

additions and $N \cdot 3 \cdot \tilde{n}$ multiplications in $\mathbb{F}_{16}$ (see Sec. 10.4.2). Operations performed in BW2 are not taken into account, because BW2 requires only a negligible amount of time. Therefore, solving the system using XL corresponds to computing about $(2 \cdot 15\,380\,937/128 + 15\,380\,937/256) \cdot 15\,380\,937 \cdot (561 + 25) \cdot 128 \approx 2^{58.3}$ additions and about $2^{50.7}$ multiplications in $\mathbb{F}_{16}$. Since one addition in $\mathbb{F}_{16}$ requires 4 bit operations, this roughly corresponds to the computation of $4 \cdot 2^{58.3} \approx 2^{60.3}$ bit operations.

Field	Machine	m	n	D	Time in [sec]				Memory	Block Size
					BW1	BW2	BW3	total	in [GB]	$\tilde{m}$, $\tilde{n}$
$\mathbb{F}_2$	Cluster	32	32	7	3830	1259	2008	7116	2.4	512, 512
	Cluster	33	33	7	6315	2135	3303	11778	3.0	512, 512
	Cluster	34	34	7	10301	2742	5439	18515	3.8	512, 512
	Cluster	35	35	7	16546	3142	8609	28387	4.6	512, 512
	Cluster	36	36	7	26235	5244	15357	46944	5.6	512, 512
$\mathbb{F}_{16}$	NUMA	56	28	6	1866	330	984	3183	3.9	128,128
	Cluster				1004	238	548	1795	1.3	256,256
	NUMA	58	29	6	2836	373	1506	4719	4.6	128,128
	Cluster				1541	316	842	2707	1.6	256,256
	NUMA	60	30	7	91228	5346	64688	161287	68.8	256,128
	Cluster				53706	3023	38052	94831	10.2	256,128
	NUMA	62	31	7	145693	7640	105084	258518	76.7	256,128
	Cluster				89059	3505	67864	160489	12.1	256,128
	NUMA	64	32	7	232865	8558	163091	404551	100.3	256,128
	Cluster				141619	3672	97924	244338	15.3	256,128
$\mathbb{F}_{31}$	NUMA	50	25	6	1729	610	935	3277	0.3	64,64
	Cluster				1170	443	648	2265	0.7	128,128
	NUMA	52	26	6	2756	888	1483	5129	0.4	64,64
	Cluster				1839	656	1013	3513	0.9	128,128
	NUMA	54	27	6	4348	1321	2340	8013	0.5	64,64
	Cluster				2896	962	1590	5453	1.0	128,128
	NUMA	56	28	6	6775	1923	3610	12313	0.6	64,64
	Cluster				4497	1397	2458	8358	1.2	128,128
	NUMA	58	29	6	10377	2737	5521	18640	0.7	64,64
	Cluster				6931	2011	3764	12713	1.5	128,128

Table 10.3: Statistics of XL with block Wiedemann for $\mathbb{F}_2$ and $\mathbb{F}_{16}$ using Thomé's BW2, and $\mathbb{F}_{31}$ using Coppersmith's BW2

Bibliography

[ABN10] Michel Abdalla, Mihir Bellare, and Gregory Neven. "Robust encryption". In: *Theory of Cryptography, 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9-11, 2010. Proceedings.* 2010, pp. 480–497. DOI: 10.1007/978-3-642-11799-2_28 (cit. on p. 78).

[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, 2004, pp. 157–167 (cit. on p. 153).

[AHK$^+$01] Kazumaro Aoki, Fumitaka Hoshino, Tetsutaro Kobayashi, and Hiroaki Oguro. "Elliptic curve arithmetic using SIMD". In: *Information security, 4th international conference, ISC 2001, Malaga, Spain, October 1–3, 2001, proceedings.* Ed. by George I. Davida and Yair Frankel. Vol. 2200. Lecture Notes in Computer Science. Springer, 2001, pp. 235–247. ISBN: 978-3-540-42662-2 (cit. on p. 20).

[AKS83] Miklós Ajtai, János Komlós, and Endre Szemerédi. "An $O(n \log n)$ sorting network". In: *Proceedings of the 15th annual ACM symposium on theory of computing, 25–27 April, 1983, Boston, Massachusetts, USA*. Ed. by David S. Johnson, Ronald Fagin, Michael L. Fredman, David Harel, Richard M. Karp, Nancy A. Lynch, Christos H. Papadimitriou, Ronald L. Rivest, Walter L. Ruzzo, and Joel I. Seiferas. Association for Computing Machinery, 1983, pp. 1–9 (cit. on p. 31).

[ALS$^+$13] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. "More efficient oblivious transfer and extensions for faster secure computation". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer communications security.* ACM. 2013, pp. 535–548 (cit. on pp. 79, 90).

[ALS$^+$15] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. "More efficient oblivious transfer extensions with security for malicious adversaries". 2015. IACR Cryptology ePrint archive: 2015/061 (cit. on pp. 79, 80).

[And05] Sean Eron Anderson. "Bit Twiddling Hacks". 1997-2005. URL: https://graphics.stanford.edu/~seander/bithacks.html (cit. on p. 48).

[ANS01] Accredited Standards Committee X9. "American national standard X9.63-2001, public key cryptography for the financial services industry: key agreement and key transport using elliptic curve cryptography". 2001. URL: `http://grouper.ieee.org/groups/1363/Research/Other.html` (cit. on pp. 108, 111).

[ANS11] Agence nationale de la sécurité des systèmes d'information. "Publication d'un paramétrage de courbe elliptique visant des applications de passeport électronique et de l'administration électronique française". 2011. URL: `https://tinyurl.com/nhog26h` (cit. on pp. 108, 111, 118).

[ANS99] Accredited Standards Committee X9. "American national standard X9.62-1999, public key cryptography for the financial services industry: the elliptic curve digital signature algorithm (ECDSA)". 1999. URL: `http://grouper.ieee.org/groups/1363/Research/Other.html` (cit. on pp. 108, 111, 121).

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson. "Lucky thirteen: breaking the TLS and DTLS record protocols". In: *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013.* IEEE Computer Society, 2013, pp. 526–540. ISBN: 978-1-4673-6166-8. DOI: `10.1109/SP.2013.42`. URL: `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6547086` (cit. on p. 8).

[Aum15] Jean-Philippe Aumasson. "Generator of "nothing-up-my-sleeve" (NUMS) constants". 2015. URL: `https://github.com/veorq/numsgen/blob/master/numsgen.py` (cit. on p. 111).

[Bat68] Kenneth E. Batcher. "Sorting networks and their applications". In: *AFIPS conference proceedings, volume 32: 1968 Spring Joint Computer Conference, Reston, Virginia.* Thompson Book Company, 1968, pp. 307–314. URL: `http://www.cs.kent.edu/~batcher/conf.html` (cit. on p. 31).

[BBC$^+$14] Benjamin Black, Joppe W. Bos, Craig Costello, Patrick Longa, and Michael Naehrig. "Elliptic curve cryptography (ECC) nothing up my sleeve (NUMS) curves and curve generation". 2014. URL: `https://tools.ietf.org/html/draft-black-numscurves-00` (cit. on pp. 94, 135).

[BBC$^+$15] Benjamin Black, Joppe W. Bos, Craig Costello, Adam Langley, Patrick Longa, and Michael Naehrig. "Rigid parameter generation for elliptic curve cryptography". 2015. URL: `https://tools.ietf.org/html/draft-black-rpgecc-01` (cit. on p. 135).

[BBJ$^+$08] Daniel J Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. "Twisted edwards curves". In: *Progress in Cryptology–AFRICACRYPT 2008.* Springer, 2008, pp. 389–405 (cit. on p. 85).

[BC13] Guido Bertoni and Jean-Sébastien Coron, eds. "Cryptographic hardware and embedded systems—CHES 2013—15th international workshop, Santa Barbara, CA, USA, August 20–23, 2013, proceedings". Vol. 8086. Lecture Notes in Computer Science. Springer, 2013. ISBN: 978-3-642-40348-4.

[BC14] Daniel J. Bernstein and Tung Chou. “Faster binary-field multiplication and faster binary-field macs”. In: *Selected Areas in Cryptography - SAC 2014 - 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers*. Ed. by Antoine Joux and Amr M. Youssef. Vol. 8781. Lecture Notes in Computer Science. Springer, 2014, pp. 92–111. ISBN: 978-3-319-13050-7. DOI: `10.1007/978-3-319-13051-4_6` (cit. on p. 3).

[BCC$^+$15] Daniel J. Bernstein, Tung Chou, Chitchanok Chuengsatiansup, Andreas Hülsing, Eran Lambooij, Tanja Lange, Ruben Niederhagen, and Christine van Vredendaal. “How to manipulate curve standards: A white paper for the black hat http: //bada55.cr.yp.to”. In: *Security Standardisation Research - Second International Conference, SSR 2015, Tokyo, Japan, December 15-16, 2015, Proceedings*. Ed. by Liqun Chen and Shin’ichiro Matsuo. Vol. 9497. Lecture Notes in Computer Science. Full version at `http://bada55.cr.yp.to`. Springer, 2015, pp. 109–139. ISBN: 978-3-319-27151-4. DOI: `10.1007/978-3-319-27152-1` (cit. on p. 3).

[BCH$^+$13] Joppe W. Bos, Craig Costello, Hüseyin Hisil, and Kristin E. Lauter. “Fast cryptography in genus 2”. In: *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*. Ed. by Thomas Johansson and Phong Q. Nguyen. Vol. 7881. Lecture Notes in Computer Science. Springer, 2013, pp. 194–210. ISBN: 978-3-642-38347-2. DOI: `10.1007/978-3-642-38348-9_12` (cit. on p. 95).

[BCL$^+$14] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Peter Schwabe. “Kummer strikes back: new DH speed records”. In: *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*. Ed. by Palash Sarkar and Tetsu Iwata. Vol. 8873. Lecture Notes in Computer Science. Springer, 2014, pp. 317–337. ISBN: 978-3-662-45610-1. DOI: `10.1007/978-3-662-45611-8_17` (cit. on pp. 94, 95).

[BCL$^+$15] Joppe W. Bos, Craig Costello, Patrick Longa, and Michael Naehrig. “Selecting elliptic curves for cryptography: an efficiency and security analysis”. In: *Journal of Cryptographic Engineering* (2015), pp. 1–28. IACR Cryptology ePrint archive: 2014/130 (cit. on pp. 111, 135–137, 140, 143).

[BCS13] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. “McBits: fast constant-time code-based cryptography”. In: *Cryptographic hardware and embedded systems—CHES 2013—15th international workshop, Santa Barbara, CA, USA, August 20–23, 2013, proceedings*. Ed. by Guido Bertoni and Jean-Sébastien Coron. Vol. 8086. Lecture Notes in Computer Science. Springer, 2013, pp. 250–272. ISBN: 978-3-642-40348-4. IACR Cryptology ePrint archive: 2015/610 (cit. on pp. 2, 41, 44, 45, 53, 59, 62).

[BD08] Johannes Buchmann and Jintai Ding, eds. “Post-quantum cryptography, second international workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17–19, 2008, proceedings”. Vol. 5299. Lecture Notes in Computer Science. Springer, 2008.

[BDL$^+$11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. “High-speed high-security signatures”. In: *Cryptographic hardware and embedded systems—CHES 2011, 13th international workshop, Nara, Japan, September 28–October 1, 2011, proceedings.* Ed. by Bart Preneel and Tsuyoshi Takagi. Vol. Springer. Lecture Notes in Computer Science. 2011. ISBN: 978-3-642-23950-2. IACR Cryptology ePrint archive: 2011/368 (cit. on pp. 23, 79, 85, 86, 88–91, 93–97, 99, 102–106).

[BDL$^+$12] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. “High-speed high-security signatures”. In: *Journal of Cryptographic Engineering* 2 (2012), pp. 77–89. IACR Cryptology ePrint archive: 2011/368 (cit. on p. 142).

[BDP$^+$11] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. “The KECCAK reference”. Jan. 2011. URL: `http://keccak.noekeon.org/` (cit. on p. 87).

[BDP$^+$13] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. “Keccak and the SHA-3 standardization”. 2013 (cit. on pp. 34, 44).

[Bea96] Donald Beaver. “Correlated pseudorandomness and the complexity of private computations”. In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996.* 1996, pp. 479–488. DOI: `10.1145/237814.237996` (cit. on p. 79).

[Ben65] Václav E. Beneš. “Mathematical theory of connecting networks and telephone traffic”. Academic Press, 1965 (cit. on p. 32).

[Ber00] Daniel J. Bernstein. “Fast multiplication”. 2000. URL: `http://cr.yp.to/talks.html#2000.08.14` (cit. on p. 58).

[Ber05] Daniel J. Bernstein. “The Poly1305-AES message-authentication code”. In: *Fast software encryption: 12th international workshop, FSE 2005, Paris, France, February 21–23, 2005, revised selected papers.* Ed. by Henri Gilbert and Helena Handschuh. Vol. 3557. Lecture Notes in Computer Science. Springer, 2005, pp. 32–49. ISBN: 3-540-26541-4. URL: `http://cr.yp.to/papers.html#poly1305` (cit. on pp. 34, 44, 57).

[Ber06] Daniel J. Bernstein. “Curve25519: new Diffie–Hellman speed records”. In: *PKC 2006.* Ed. by Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin. Vol. 3958. Lecture Notes in Computer Science. Springer, 2006, pp. 207–228. DOI: `10.1007/11745853_14`. URL: `http://cr.yp.to/papers.html#curve25519` (cit. on pp. 79, 101, 102, 142).

[Ber07a] Daniel J. Bernstein. “Polynomial evaluation and message authentication”. 2007. URL: `http://cr.yp.to/papers.html#pema` (cit. on pp. 66, 67).

[Ber07b] Daniel J. Bernstein. “qhasm software package”. 2007 (cit. on pp. 9, 29, 104).

[Ber08a] Daniel J. Bernstein. “Fast multiplication and its applications”. In: *Surveys in algorithmic number theory*. Ed. by Joe P. Buhler and Peter Stevenhagen. Vol. 44. Mathematical Sciences Research Institute Publications. New York: Cambridge University Press, 2008, pp. 325–384. URL: http://cr.yp.to/papers.html#multapps (cit. on p. 67).

[Ber08b] Daniel J. Bernstein. “The Salsa20 family of stream ciphers”. In: *New stream cipher designs*. Ed. by Matthew Robshaw and Olivier Billet. Vol. 4986. Lecture Notes in Computer Science. Springer, 2008, pp. 84–97. ISBN: 978-3-540-68350-6. URL: http://cr.yp.to/papers.html#salsafamily (cit. on pp. 34, 44).

[Ber09a] Daniel J. Bernstein. “Batch binary Edwards”. In: *Advances in cryptology—CRYPTO 2009, 29th annual international cryptology conference, Santa Barbara, CA, USA, August 16–20, 2009, proceedings*. Ed. by Shai Halevi. Vol. 5677. Lecture Notes in Computer Science. Springer, 2009, pp. 317–336. ISBN: 978-3-642-03355-1. URL: http://cr.yp.to/papers.html#bbe (cit. on pp. 20, 21, 24, 58).

[Ber09b] Daniel J. Bernstein. “Minimum number of bit operations for multiplication”. 2009. URL: http://binary.cr.yp.to/m.html (cit. on pp. 58, 60, 65).

[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*. 2009, pp. 3–18. URL: http://cr.yp.to/papers.html#linearmod2 (cit. on p. 61).

[Ber11] Daniel J. Bernstein. “Simplified high-speed high-distance list decoding for alternant codes”. In: *Post-quantum cryptography, fourth international workhop, PQCrypto, Taipei, Taiwan, November 29–December 02, 2011*. Ed. by Bo-Yin Yang. Vol. 7071. Lecture Notes in Computer Science. Springer, 2011, pp. 200–216. URL: http://cr.yp.to/papers.html#simplelist (cit. on p. 36).

[Ber68] Elwyn R. Berlekamp. “Algebraic coding theory”. McGraw-Hill, 1968 (cit. on p. 35).

[Ber70] Elwyn R. Berlekamp. “Factoring polynomials over large finite fields”. In: *Mathematics of Computation* 24 (1970), pp. 713–715. URL: http://www.ams.org/journals/mcom/1970-24-111/S0025-5718-1970-0276200-X/S0025-5718-1970-0276200-X.pdf (cit. on p. 26).

[BHK$^+$13] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. “Elligator: elliptic-curve points indistinguishable from uniform random strings”. In: *ACM CCS’13*. Ed. by Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung. ACM, 2013, pp. 967–980. DOI: 10.1145/2508859.2516734. URL: http://elligator.cr.yp.to/ (cit. on pp. 135, 139).

[BHK+99] John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway. "UMAC: fast and secure message authentication". In: *Advances in cryptology—CRYPTO '99*. Ed. by Michael Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 216–233. ISBN: 3-5540-66347-9. URL: `http://www.cs.ucdavis.edu/~rogaway/umac/` (cit. on pp. 57, 66).

[Bih97a] Eli Biham. "A fast new DES implementation in software". In: *Fast software encryption, 4th international workshop, FSE '97, Haifa, Israel, January 20–22, 1997, proceedings*. Ed. by Eli Biham. Vol. 1267. Lecture Notes in Computer Science. Springer, 1997, pp. 260–272. ISBN: 3-540-63247-6 (cit. on p. 20).

[Bih97b] Eli Biham, ed. "Fast software encryption, 4th international workshop, FSE '97, Haifa, Israel, January 20–22, 1997, proceedings". Vol. 1267. Lecture Notes in Computer Science. Springer, 1997. ISBN: 3-540-63247-6.

[BJ02] Eric Brier and Marc Joye. "Weierstraß elliptic curves and side-channel attacks". In: *Public Key Cryptography*. Ed. by David Naccache and Pascal Paillier. Vol. 2274. Lecture Notes in Computer Science. Springer, 2002, pp. 335–345. ISBN: 3-540-43168-3. URL: `http://joye.site88.net/papers/BJ02espa.pdf` (cit. on p. 113).

[BKN+10] Joppe Bos, Thorsten Kleinjung, Ruben Niederhagen, and Peter Schwabe. "ECC2K-130 on Cell CPUs". In: *Progress in cryptology—AFRICACRYPT 2010, third international conference on cryptology in Africa, Stellenbosch, South Africa, May 3–6, 2010, proceedings*. Ed. by Daniel J. Bernstein and Tanja Lange. Vol. 6055. Lecture Notes in Computer Science. Springer, 2010, pp. 225–242. IACR Cryptology ePrint archive: 2010/077 (cit. on p. 20).

[BL13] Daniel J. Bernstein and Tanja Lange, eds. "eBACS: ECRYPT Benchmarking of Cryptographic Systems". Note: accessed 10 June 2013. 2013 (cit. on pp. 22, 40, 93–95).

[BL15a] Daniel J. Bernstein and Tanja Lange. "SafeCurves: choosing safe curves for elliptic-curve cryptography". (accessed 27 September 2015). 2015. URL: `http://safecurves.cr.yp.to` (cit. on pp. 79, 112, 113, 117).

[BL15b] Daniel J Bernstein and Tanja Lange. "eBACS: ecrypt benchmarking of cryptographic systems". accessed 16 March 2015. URL: `http://bench.cr.yp.to` (cit. on p. 90).

[BLN+15] Sai Sheshank Burra, Enrique Larraia, Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart. "High performance multi-party computation for binary circuits based on oblivious transfer". 2015. IACR Cryptology ePrint archive: 2015/472 (cit. on p. 77).

[BLP08] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. "Attacking and defending the McEliece cryptosystem". In: *Post-quantum cryptography, second international workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17–19, 2008, proceedings.* Ed. by Johannes Buchmann and Jintai Ding. Vol. 5299. Lecture Notes in Computer Science. Springer, 2008, pp. 31–46. IACR Cryptology ePrint archive: 2008/318 (cit. on pp. 19, 22).

[BM74] Allan Borodin and Robert T. Moenck. "Fast modular transforms". In: *Journal of Computer and System Sciences* 8 (1974). Note: older version, not a subset, in [MB72], pp. 366–386. ISSN: 0022–0000 (cit. on pp. 25, 190).

[BM89] Mihir Bellare and Silvio Micali. "Non-interactive oblivious transfer and spplications". In: *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings.* 1989, pp. 547–557. DOI: `10.1007/0-387-34805-0_48` (cit. on p. 79).

[Bor56] J. L. Bordewijk. "Inter-reciprocity applied to electrical networks". In: *Applied Scientific Research B: Electrophysics, Acoustics, Optics, Mathematical Methods* 6 (1956), pp. 1–74 (cit. on p. 28).

[BP96] Eric Bach and René Peralta. "Asymptotic semismoothness probabilities". In: *Math. Comput.* 65.216 (1996), pp. 1701–1715. DOI: `10.1090/S0025-5718-96-00775-2`. URL: `http://www.ams.org/journals/mcom/1996-65-216/S0025-5718-96-00775-2/S0025-5718-96-00775-2.pdf` (cit. on p. 118).

[Bra05] ECC Brainpool. "ECC Brainpool standard curves and curve generation". 2005. URL: `http://www.ecc-brainpool.org/download/Domain-parameters.pdf` (cit. on pp. 108, 111, 126–128, 137, 141, 145).

[Bro] Nicolai Brown. "IANIX". URL: `https://ianix.com/` (cit. on p. 93).

[BS08] Bhaskar Biswas and Nicolas Sendrier. "McEliece cryptosystem implementation: theory and practice". In: *Post-quantum cryptography, second international workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17–19, 2008, proceedings.* Ed. by Johannes Buchmann and Jintai Ding. Vol. 5299. Lecture Notes in Computer Science. Springer, 2008, pp. 47–62 (cit. on pp. 20, 22, 23).

[BS12] Daniel J. Bernstein and Peter Schwabe. "NEON crypto". In: *CHES 2012.* Ed. by Emmanuel Prouff and Patrick Schaumont. Vol. 7428. Lecture Notes in Computer Science. Springer, 2012, pp. 320–339. ISBN: 978-3-642-33026-1. DOI: `10.1007/978-3-642-33027-8_19`. URL: `http://cr.yp.to/papers.html#neoncrypto` (cit. on pp. 21, 93, 94, 96, 97, 99, 102, 103, 142).

[BSP$^+$05] Martin Boesgaard, Ove Scavenius, Thomas Pedersen, Thomas Christensen, and Erik Zenner. "Badger—a fast and provably secure MAC". In: *Applied cryptography and network security, third international conference, ACNS 2005, New York, NY, USA, June 7-10, 2005, proceedings.* Vol. 3531. Lecture Notes in Computer Science. 2005, pp. 176–191. ISBN: 3-540-26223-7 (cit. on p. 66).

[Can01] Ran Canetti. "Universally composable security: A new paradigm for cryptographic protocols". In: *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA.* 2001, pp. 136–145. DOI: 10.1109/SFCS.2001.959888 (cit. on pp. 79, 82).

[Can89] David G. Cantor. "On arithmetical algorithms over finite fields". In: *Journal of Combinatorial Theory, Series A* 50 (1989), pp. 285–300 (cit. on p. 11).

[CCN$^+$12] Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, and Bo-Yin Yang. "Solving quadratic equations with XL on parallel architectures". In: *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings.* Ed. by Emmanuel Prouff and Patrick Schaumont. Vol. 7428. Lecture Notes in Computer Science. Springer, 2012, pp. 356–373. ISBN: 978-3-642-33026-1. DOI: 10.1007/978-3-642-33027-8_21 (cit. on p. 3).

[Cer00a] Certicom Research. "SEC 2: recommended elliptic curve domain parameters, version 1.0". 2000. URL: http://www.secg.org/SEC2-Ver-1.0.pdf (cit. on pp. 108, 111).

[Cer00b] Certicom Research. "SEC 1: elliptic curve cryptography, version 1.0". 2000. URL: http://www.secg.org/SEC1-Ver-1.0.pdf (cit. on pp. 111, 112).

[Cer09] Certicom Research. "SEC 1: elliptic curve cryptography, version 2.0". 2009. URL: http://www.secg.org/sec1-v2.pdf (cit. on pp. 111, 112).

[Cer10] Certicom Research. "SEC 2: recommended elliptic curve domain parameters, version 2.0". 2010. URL: http://www.secg.org/sec2-v2.pdf (cit. on pp. 111, 121, 139).

[CF01] Ran Canetti and Marc Fischlin. "Universally composable commitments". In: (2001). IACR Cryptology ePrint archive: 2001/055 (cit. on p. 79).

[CFN$^+$14] Stephen Checkoway, Matthew Fredrikson, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, and Hovav Shacham. "On the practical exploitability of Dual EC in TLS implementations". In: *23rd USENIX Security Symposium (USENIX Security 14).* San Diego, CA: USENIX Association, 2014. URL: https://projectbullrun.org/dual-ec/index.html (cit. on pp. 110, 113).

[CFS01] Nicolas Courtois, Matthieu Finiasz, and Nicolas Sendrier. "How to achieve a McEliece-based digital signature scheme". In: *Advances in cryptology—ASIACRYPT 2001, proceedings of the 7th international conference on the theory and application of cryptology and information security held on the Gold Coast, December 9–13, 2001*. Ed. by Colin Boyd. Vol. 2248. Lecture Notes in Computer Science. Springer, 2001, pp. 157–174. ISBN: 3-540-42987-5. URL: `http://hal.inria.fr/docs/00/07/25/11/PDF/RR-4118.pdf` (cit. on p. 36).

[Cho15] Tung Chou. "Sandy2x: fastest Curve25519 implementation ever". 2015. URL: `http://csrc.nist.gov/groups/ST/ecc-workshop-2015/presentations/session6-chou-tung.pdf` (cit. on pp. 3, 142).

[Cho16] Tung Chou. "Qcbits: constant-time small-key code-based cryptography". In: (2016) (cit. on p. 3).

[CHS14] Craig Costello, Hüseyin Hisil, and Benjamin Smith. "Faster compact diffie-hellman: endomorphisms on the x-line". In: *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*. Ed. by Phong Q. Nguyen and Elisabeth Oswald. Vol. 8441. Lecture Notes in Computer Science. Springer, 2014, pp. 183–200. ISBN: 978-3-642-55219-9. DOI: `10.1007/978-3-642-55220-5_11` (cit. on p. 94).

[CKP+00] Nicolas Tadeusz Courtois, Alexnader 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, 2000, pp. 392–407 (cit. on pp. 153, 154).

[CKP+05] Nam Su Chang, Chang Han Kim, Young-Ho Park, and Jongin Lim. "A non-redundant and efficient architecture for Karatsuba-Ofman algorithm". In: *Information security, 8th international conference, ISC 2005, Singapore, September 20–23, 2005, proceedings*. Ed. by Jianying Zhou, Javier Lopez, Robert H. Deng, and Feng Bao. Vol. 3650. Lecture Notes in Computer Science. Springer, 2005, pp. 288–299. ISBN: 3-540-29001-X (cit. on p. 58).

[CO15] Tung Chou and Claudio Orlandi. "The simplest protocol for oblivious transfer". In: *Progress in Cryptology - LATINCRYPT 2015 - 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings*. Ed. by Kristin E. Lauter and Francisco Rodríguez-Henríquez. Vol. 9230. Lecture Notes in Computer Science. Springer, 2015, pp. 40–58. ISBN: 978-3-319-22173-1. DOI: `10.1007/978-3-319-22174-8_3` (cit. on p. 3).

[Cop94] Don Coppersmith. "Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm". In: *Mathematics of Computation* 62.205 (1994), pp. 333–350 (cit. on pp. 154, 155, 157).

[Cou03] Nicolas Tadeusz 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, 2003, pp. 182–199 (cit. on p. 154).

[CP02] Nicolas Tadeusz 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, 2002, pp. 267–287 (cit. on p. 153).

[CS09] Neil Costigan and Peter Schwabe. “Fast elliptic-curve cryptography on the Cell Broadband Engine”. In: *Africacrypt 2009*. 2009, pp. 368–385. URL: `https://cryptojedi.org/papers/celldh-20090331.pdf` (cit. on pp. 93, 94, 142).

[DHH$^+$15] Michael Düll, Björn Haase, Gesine Hinterwälder, Michael Hutter, Christof Paar, Ana Helena Sánchez, and Peter Schwabe. “High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers”. In: *Designs, Codes and Cryptography* (2015). URL: `http://link.springer.com/article/10.1007/s10623-015-0087-1/fulltext.html` (cit. on pp. 93, 142).

[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, 2004, pp. 146–159 (cit. on p. 154).

[DNO08] Ivan Damgård, Jesper Buus Nielsen, and Claudio Orlandi. “Essentially optimal universally composable oblivious transfer”. In: *Information Security and Cryptology - ICISC 2008, 11th International Conference, Seoul, Korea, December 3-5, 2008, Revised Selected Papers*. 2008, pp. 318–335. DOI: `10.1007/978-3-642-00730-9_20` (cit. on pp. 79, 82).

[DSV13] Davide D’Angella, Chiara Valentina Schiavo, and Andrea Visconti. “Tight upper bounds for polynomial multiplication”. In: *Proceedings of the 6th WSEAS world congress: applied computing conference (ACC 2013)*. WSEAS, 2013. URL: `http://www.wseas.us/e-library/conferences/2013/Nanjing/ACCIS/ACCIS-03.pdf` (cit. on pp. 58, 65).

[EGL85] Shimon Even, Oded Goldreich, and Abraham Lempel. “A randomized protocol for signing contracts”. In: *Commun. ACM* 28.6 (1985), pp. 637–647. DOI: `10.1145/3812.3818` (cit. on p. 79).

[Fau02] Jean-Charles Faugère. “A new efficient algorithm for computing Gröbner bases without reduction to zero (F_5)”. In: *International Symposium on Symbolic and Algebraic Computation – ISSAC 2002*. ACM, 2002, pp. 75–83 (cit. on p. 153).

[Fau99] Jean-Charles Faugère. “A new efficient algorithm for computing Gröbner bases (F_4)”. In: *Journal of Pure and Applied Algebra* 139.1–3 (1999), pp. 61–88 (cit. on pp. 153, 171).

[Fid72] Charles M. Fiduccia. “On obtaining upper bounds on the complexity of matrix multiplication”. In: *Complexity of computer computations*. Ed. by Raymond E. Miller and James W. Thatcher. Plenum Press, 1972, pp. 31–40. ISBN: 0306307073 (cit. on p. 28).

[Fid73] Charles M. Fiduccia. “On the algebraic complexity of matrix multiplication”. PhD thesis. 1973 (cit. on p. 28).

[Fin11] Matthieu Finiasz. “Parallel-CFS—strengthening the CFS McEliece-based signature scheme”. In: *Selected areas in cryptography—17th international workshop, SAC 2010, Waterloo, Ontario, Canada, August 12–13, 2010, revised selected papers*. Ed. by Alex Biryukov, Guang Gong, and Douglas R. Stinson. Vol. 6544. Lecture Notes in Computer Science. Springer, 2011, pp. 159–170. ISBN: 978-3-642-19573-0. URL: `https://www.rocq.inria.fr/secret/Matthieu.Finiasz/research/2010/finiasz-sac10.pdf` (cit. on p. 36).

[FLP$^+$13] Pooya Farshim, Benoît Libert, Kenneth G. Paterson, and Elizabeth A. Quaglia. “Robust encryption, revisited”. In: *Public-Key Cryptography - PKC 2013 - 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26 - March 1, 2013. Proceedings*. 2013, pp. 352–368. DOI: `10.1007/978-3-642-36362-7_22` (cit. on p. 78).

[Fog14] Agner Fog. “Instruction tables”. 2014. URL: `http://www.agner.org/optimize/instruction_tables.pdf` (cit. on pp. 59, 89, 99).

[FPP$^+$12] Jean-Charles Faugère, Ludovic Perret, Christophe Petit, and Guénaël Renault. “Improving the complexity of index calculus algorithms in elliptic curves over binary fields”. In: *Advances in Cryptology – EUROCRYPT 2012*. Ed. by David Pointcheval and Thomas Johansson. Vol. 7237. Lecture Notes in Computer Science. Springer, 2012, pp. 27–44 (cit. on p. 154).

[FPR$^+$15] Jean-Pierre Flori, Jérôme Plût, Jean-René Reinhard, and Martin Ekerå. “Diversity and transparency for ECC”. 2015. URL: `http://csrc.nist.gov/groups/ST/ecc-workshop-2015/papers/session4-flori-jean-pierre.pdf` (cit. on pp. 138, 141).

[FS10] Luca De Feo and Éric Schost. “`transalpyne`: a language for automatic transposition”. 2010 (cit. on p. 29).

[GG96] Joachim von zur Gathen and Jürgen Gerhard. “Arithmetic and factorization of polynomials over $\mathbb{F}_2$ (extended abstract)”. In: *Proceedings of the 1996 international symposium on symbolic and algebraic computation, ISSAC ’96, Zurich, Switzerland, July 24–26, 1996*. Ed. by Erwin Engeler, B. F. Caviness, and Yagati N. Lakshman. Association for Computing Machinery, 1996, pp. 1–9. ISBN: 0-89791-796-0 (cit. on p. 11).

[GM00] Steven D. Galbraith and James McKee. “The probability that the number of points on an elliptic curve over a finite field is prime”. In: *Journal of the London Mathematical Society* 62 (2000), pp. 671–684. URL: `https://www.math.auckland.ac.nz/~sgal018/cm.pdf` (cit. on p. 115).

[GM10] Shuhong Gao and Todd Mateer. "Additive fast Fourier transforms over finite fields". In: *IEEE Transactions on Information Theory* 56 (2010), pp. 6265–6272. URL: `http://www.math.clemson.edu/~sgao/pub.html` (cit. on pp. 11–13, 62).

[Gra08] Andrew Granville. "Smooth numbers: computational number theory and beyond". In: *Algorithmic Number Theory: Lattices, Number Fields, Curves and Cryptography*. Cambridge University Press, 2008, pp. 267–323. ISBN: 9780521808545. URL: `http://en.scientificcommons.org/43534098` (cit. on p. 116).

[GS06] Joachim von zur Gathen and Jamshid Shokrollahi. "Fast arithmetic for polynomials over $\mathbf{F}_2$ in hardware". In: *Information theory workshop, 2006. ITW '06 Chengdu*. IEEE, 2006, pp. 107–111 (cit. on p. 58).

[GT07] Pierrick Gaudry and Emmanuel Thomé. "The mpFq library and implementing curve-based key exchanges". In: *SPEED: software performance enhancement for encryption and decryption*. 2007, pp. 49–64. URL: `http://www.loria.fr/~gaudry/papers.en.html` (cit. on p. 142).

[Gue13] Shay Gueron. "AES-GCM software performance on the current high end CPUs as a performance baseline for CAESAR". 2013. URL: `http://2013.diac.cr.yp.to/slides/gueron.pdf` (cit. on p. 59).

[Hal07] Shai Halevi. "Invertible universal hashing and the TET encryption mode". In: *Advances in cryptology—CRYPTO 2007, 27th annual international cryptology conference, Santa Barbara, CA, USA, August 19–23, 2007, proceedings*. Ed. by Alfred Menezes. Vol. 4622. Lecture Notes in Computer Science. Springer, 2007, pp. 412–429. ISBN: 978-3-540-74142-8. IACR Cryptology ePrint archive: 2007/014 (cit. on p. 66).

[HG12] Stefan Heyse and Tim Güneysu. "Towards one cycle per bit asymmetric encryption: code-based cryptography on reconfigurable hardware". In: *Cryptographic hardware and embedded systems—CHES 2012—14th international workshop, Leuven, Belgium, September 9–12, 2012, proceedings*. Ed. by Emmanuel Prouff and Patrick Schaumont. Vol. 7428. Lecture Notes in Computer Science. Springer, 2012, pp. 340–355. ISBN: 978-3-642-33026-1 (cit. on pp. 22, 23).

[HK97] Shai Halevi and Hugo Krawczyk. "MMH: software message authentication in the Gbit/second rates". In: *Fast software encryption, 4th international workshop, FSE '97, Haifa, Israel, January 20–22, 1997, proceedings*. Ed. by Eli Biham. Vol. 1267. Lecture Notes in Computer Science. Springer, 1997, pp. 172–189. ISBN: 3-540-63247-6. URL: `http://www.research.ibm.com/people/s/shaih/pubs/mmh.html` (cit. on p. 66).

[HL10] Carmit Hazay and Yehuda Lindell. "Efficient Secure Two-Party Protocols - Techniques and Constructions". Information Security and Cryptography. Springer, 2010. ISBN: 978-3-642-14302-1. DOI: `10.1007/978-3-642-14303-8` (cit. on pp. 79, 83).

[HMG13] Stefan Heyse, Ingo von Maurich, and Tim Güneysu. "Smaller Keys for Code-Based Cryptography: QC-MDPC McEliece Implementations on Embedded Devices". In: *Cryptographic hardware and embedded systems—CHES 2013—15th international workshop, Santa Barbara, CA, USA, August 20–23, 2013, proceedings.* Ed. by Guido Bertoni and Jean-Sébastien Coron. Vol. 8086. Lecture Notes in Computer Science. Springer, 2013, pp. 273–292. ISBN: 978-3-642-40348-4. IACR Cryptology ePrint archive: 2015/425.pdf (cit. on pp. 39, 40, 42, 43, 46, 54).

[HP08] Helena Handschuh and Bart Preneel. "Key-recovery attacks on universal hash function based MAC algorithms". In: *Advances in cryptology—CRYPTO 2008, 28th annual international cryptology conference, Santa Barbara, CA, USA, August 17–21, 2008, proceedings.* Ed. by David Wagner. Vol. 5157. Lecture Notes in Computer Science. Springer, 2008, pp. 144–161. ISBN: 978-3-540-85173-8 (cit. on p. 66).

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. "Ntru: a ring-based public key cryptosystem". In: *Algorithmic number theory.* Springer, 1998, pp. 267–288 (cit. on p. 2).

[HSS+15] Michael Hutter, Jürgen Schilling, Peter Schwabe, and Wolfgang Wieser. "NaCl's `crypto_box` in hardware". In: *CHES 2015.* Ed. by Tim Güneysu and Helena Handschuh. Vol. 9293. Lecture Notes in Computer Science. Springer, 2015, pp. 81–101. ISBN: 978-3-662-48323-7. DOI: 10.1007/978-3-662-48324-4_5. URL: https://cryptojedi.org/papers/naclhw-20150616.pdf (cit. on p. 142).

[HVP10] Jens Hermans, Frederik Vercauteren, and Bart Preneel. "Speed records for NTRU". In: *Topics in cryptology—CT-RSA 2010, the cryptographers' track at the RSA Conference 2010, San Francisco, CA, USA, March 1–5, 2010, proceedings.* Ed. by Josef Pieprzyk. Vol. 5985. Lecture Notes in Computer Science. Springer, 2010, pp. 73–88. ISBN: 978-3-642-11924-8. URL: http://homes.esat.kuleuven.be/~fvercaut/papers/ntru_gpu.pdf (cit. on p. 23).

[HWC+08] Huseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. "Twisted Edwards curves revisited". In: *Advances in Cryptology-ASIACRYPT 2008.* Springer, 2008, pp. 326–343 (cit. on pp. 79, 85, 103, 105).

[IEE00] Institute of Electrical and Electronics Engineers. "IEEE 1363-2000: standard specifications for public key cryptography". 2000. URL: http://grouper.ieee.org/groups/1363/P1363/draft.html (cit. on pp. 108, 111, 121).

[IHT06] José Luis Imaña, Román Hermida, and Francisco Tirado. "Low-complexity bit-parallel multipliers based on a class of irreducible pentanomials". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 14 (2006), pp. 1388–1393 (cit. on p. 61).

[IKN+03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. "Extending oblivious transfers efficiently". In: *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings.* 2003, pp. 145–161. DOI: `10.1007/978-3-540-45146-4_9` (cit. on p. 79).

[IOM12] Tetsu Iwata, Keisuke Ohashi, and Kazuhiko Minematsu. "Breaking and repairing GCM security proofs". In: *Advances in cryptology—CRYPTO 2012—32nd annual cryptology conference, Santa Barbara, CA, USA, August 19–23, 2012, proceedings.* Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. Lecture Notes in Computer Science. Springer, 2012, pp. 31–49. ISBN: 978-3-642-32008-8. IACR Cryptology ePrint archive: 2012/438 (cit. on p. 65).

[IR89] Russell Impagliazzo and Steven Rudich. "Limits on the provable consequences of one-way permutations". In: *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA.* 1989, pp. 44–61. DOI: `10.1145/73007.73012` (cit. on p. 79).

[Jab01] A. Kh. Al Jabri. "Cryptography and Coding, 8th IMA International Conference, Cirencester, UK, December 17-19, 2001, Proceeding". In: *Cryptography and Coding, 8th IMA International Conference, Cirencester, UK, December 17-19, 2001, Proceeding.* Ed. by Bahram Honary. Vol. 2260. Lecture Notes in Computer Science. Springer, 2001, pp. 1–8. ISBN: 3-540-43026-1 (cit. on p. 43).

[JSS15] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. "Practical invalid curve attacks on TLS-ECDH". In: *ESORICS 2015.* 2015. URL: `https://www.nds.rub.de/research/publications/ESORICS15/` (cit. on p. 113).

[KAF+10] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen Klaas Lenstra, Emmanuel Thomé, Joppe Willem Bos, Pierrick Gaudry, Alexander Kruppa, Peter Lawrence 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, 2010, pp. 333–350 (cit. on pp. 161, 171).

[Kel03] John Kelsey. "Choosing a DRBG algorithm". 2003? URL: `https://github.com/matthewdgreen/nistfoia/blob/master/6.4.2014%20production/011%20-%209.12%20Choosing%20a%20DRBG%20Algorithm.pdf` (cit. on p. 110).

[KI01] Kazukuni Kobara and Hideki Imai. "Semantically secure McEliece public-key cryptosystems—conversions for McEliece PKC". In: *Public key cryptography: proceedings of the 4th international workshop on practice and theory in public key cryptosystems (PKC 2001) held on Cheju Island, February 13–15, 2001.* Ed. by Kwangjo Kim. Vol. 1992. Lecture Notes in Computer Science. Springer, 2001, pp. 19–35 (cit. on p. 33).

[Kil88] Joe Kilian. “Founding cryptography on oblivious transfer”. In: *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*. 1988, pp. 20–31. DOI: 10.1145/62212.62215 (cit. on p. 77).

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. “Actively secure ot extension with optimal overhead”. CRYPTO. 2015 (cit. on p. 79).

[KR11] Ted Krovetz and Philip Rogaway. “The software performance of authenticated-encryption modes”. In: *Fast software encryption—18th international workshop, FSE 2011, Lyngby, Denmark, February 13–16, 2011, revised selected papers*. Ed. by Antoine Joux. Vol. 6733. Lecture Notes in Computer Science. Springer, 2011, pp. 306–327. ISBN: 978-3-642-21701-2. URL: http://www.cs.ucdavis.edu/~rogaway/papers/ae.pdf (cit. on p. 59).

[Kro07] Ted Krovetz. “Message authentication on 64-bit architectures ”. In: *Selected areas in cryptography, 13th international workshop, SAC 2006, Montreal, Canada, August 17–18, 2006, revised selected papers*. Ed. by Eli Biham and Amr M. Youssef. Vol. 4356. Lecture Notes in Computer Science. Springer, 2007, pp. 327–341. ISBN: 978–3–540–74461–0 (cit. on pp. 57, 66).

[KS09] Emilia Käsper and Peter Schwabe. “Faster and timing-attack resistant AES-GCM”. In: *Cryptographic hardware and embedded systems—CHES 2009, 11th international workshop, Lausanne, Switzerland, September 6–9, 2009, proceedings*. Ed. by Christophe Clavier and Kris Gaj. Vol. 5747. Lecture Notes in Computer Science. Springer, 2009, pp. 1–17. ISBN: 978-3-642-04137-2. IACR Cryptology ePrint archive: 2009/129 (cit. on pp. 20, 57–59).

[Lar14] Enrique Larraia. “Extending oblivious transfer efficiently, or - how to get active security with constant cryptographic overhead”. In: (2014). IACR Cryptology ePrint archive: 2014/692 (cit. on p. 79).

[Laz83] Daniel Lazard. “Gröbner-bases, Gaussian elimination and resolution of systems of algebraic equations”. In: *Computer Algebra – EUROCAL ’83*. Ed. by J. A. van Hulzen. Vol. 162. Lecture Notes in Computer Science. Springer, 1983, pp. 146–156 (cit. on p. 153).

[LC14] Brian LaMacchia and Craig Costello. “Deterministic generation of elliptic curves (a.k.a. “NUMS” curves)”. 2014. URL: https://www.ietf.org/proceedings/90/slides/slides-90-cfrg-5.pdf (cit. on p. 135).

[LM10] Manfred Lochter and Johannes Merkle. “RFC 5639: elliptic curve cryptography (ECC) Brainpool standard curves and curve generation”. 2010. URL: https://tools.ietf.org/html/rfc5639 (cit. on pp. 127, 128, 136).

[LM13] Adam Langley and Andrew Moon. “Implementations of a fast elliptic-curve digital signature algorithm”. 2013. URL: https://github.com/floodyberry/ed25519-donna (cit. on pp. 94, 95, 103, 106, 142).

[LMS$^+$14] Manfred Lochter, Johannes Merkle, Jörn-Marc Schmidt, and Torsten Schütze. "Requirements for standard elliptic curves". 2014. URL: http://www.ecc-brainpool.org/20141001_ECCBrainpool_PositionPaper.pdf (cit. on p. 137).

[LMS04] Florian Luca, David Jose Mireles, and Igor E. Shparlinski. "MOV attack in various subgroups on elliptic curves". In: *Illinois Journal of Mathematics* 48.3 (July 2004), pp. 1041–1052. URL: https://projecteuclid.org/euclid.ijm/1258131069 (cit. on p. 116).

[LS12a] Grégory Landais and Nicolas Sendrier. "CFS software implementation". Note: see also newer version [LS12b]. 2012. IACR Cryptology ePrint archive: 2012/132 (cit. on p. 190).

[LS12b] Grégory Landais and Nicolas Sendrier. "Implementing CFS". In: *Progress in cryptology—Indocrypt 2012—13th international conference on cryptology in India, Kolkata, India, December 9–12, 2012, proceedings.* Ed. by Steven Galbraith and Mridul Nandi. Vol. 7668. Lecture Notes in Computer Science. Note: see also older version [LS12a]. Springer, 2012, pp. 474–488 (cit. on pp. 36, 37, 190).

[LS12c] Patrick Longa and Francesco Sica. "Four-dimensional gallant-lambert-vanstone scalar multiplication". In: *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings.* Ed. by Xiaoyun Wang and Kazue Sako. Vol. 7658. Lecture Notes in Computer Science. Springer, 2012, pp. 718–739. ISBN: 978-3-642-34960-7. DOI: 10.1007/978-3-642-34961-4_43 (cit. on p. 94).

[LST12] Will Landecker, Thomas Shrimpton, and R. Seth Terashima. "Tweakable blockciphers with beyond birthday-bound security". In: *Advances in cryptology—CRYPTO 2012—32nd annual cryptology conference, Santa Barbara, CA, USA, August 19–23, 2012, proceedings.* Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. Lecture Notes in Computer Science. Springer, 2012, pp. 14–30. ISBN: 978-3-642-32008-8 (cit. on p. 66).

[Lup56] O. B. Lupanov. "On rectifier and contact-rectifier circuits". In: *Doklady Akademii Nauk SSSR* 111 (1956), pp. 1171–1174. ISSN: 0002–3264 (cit. on p. 28).

[LW15] Manfred Lochter and Andreas Wiemers. "Twist insecurity". 2015. IACR Cryptology ePrint archive: 2015/577 (cit. on p. 114).

[Mac16] Francis Sowerby Macaulay. "The Algebraic Theory of Modular Systems". Cambridge Tracts in Mathematics and Mathematical Physics 19. Cambridge University Press, 1916 (cit. on p. 154).

[MB72] Robert T. Moenck and Allan Borodin. "Fast modular transforms via division". In: *13th annual symposium on switching and automata theory.* Note: newer version, not a superset, in [BM74]. IEEE Computer Society, 1972, pp. 90–96 (cit. on p. 181).

[MC14] Eric M. Mahé and Jean-Marie Chauvet. “Fast GPGPU-based elliptic curve scalar multiplication”. 2014. IACR Cryptology ePrint archive: 2014/198.pdf (cit. on p. 142).

[McE78] Robert J. McEliece. “A public-key cryptosystem based on algebraic coding theory”. Note: JPL DSN Progress Report. 1978. URL: http://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF (cit. on pp. 2, 19, 39, 42).

[MDK$^+$10] Wael Said Abd Elmageed Mohamed, Jintai Ding, Thorsten Kleinjung, Stanislav Bulygin, and Johannes Buchmann. “PWXL: a parallel Wiedemann-XL algorithm for solving polynomial equations over GF(2)”. In: *International Conference on Symbolic Computation and Cryptography – SCC 2010*. Ed. by Carlos Cid and Jean-Charles Faugère. 2010, pp. 89–100 (cit. on pp. 154, 171).

[Mer14] Johannes Merkle. “Re: [Cfrg] ECC reboot (Was: When’s the decision?)” 2014. URL: https://www.ietf.org/mail-archive/web/cfrg/current/msg05353.html (cit. on p. 131).

[MG14a] Ingo von Maurich and Tim Güneysu. “Lightweight code-based cryptography: QC-MDPC McEliece encryption on reconfigurable devices”. In: *Design, Automation and Test in Europe Conference and Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*. Ed. by Gerhard Fettweis and Wolfgang Nebel. European Design and Automation Association, 2014, pp. 1–6. ISBN: 978-3-9815370-2-4. URL: https://www.sha.rub.de/media/sh/veroeffentlichungen/2014/02/11/Lightweight_Code-based_Cryptography.pdf (cit. on pp. 39, 40, 42, 43, 46, 54).

[MG14b] Ingo von Maurich and Tim Güneysu. “Towards Side-Channel Resistant Implementations of QC-MDPC McEliece Encryption on Constrained Devices”. In: *Post-quantum cryptography*. Ed. by Michele Mosca. Vol. 8772. Lecture Notes in Computer Science. Springer and Springer, 2014, pp. 266–282. URL: http://ei.rub.de/media/sh/veroeffentlichungen/2014/07/17/Side-Channel_Resistant_QC-MDPC_McEliece.pdf (cit. on pp. 39–43, 46, 54).

[MHG16] Ingo von Maurich, Lukas Heberle, and Tim Güneysu. “IND-CCA Secure Hybrid Encryption from QC-MDPC Niederreiter”. In: *Post-quantum cryptography*. Ed. by Tsuyoshi Takagi. Vol. 9606. Lecture Notes in Computer Science. Springer and Springer, 2016 (cit. on pp. 39–43, 45, 46, 54).

[MM12] Seiichi Matsuda and Shiho Moriai. “Lightweight cryptography for the cloud: exploit the power of bitslice implementation”. In: *Cryptographic hardware and embedded systems—CHES 2012—14th international workshop, Leuven, Belgium, September 9–12, 2012, proceedings*. Ed. by Emmanuel Prouff and Patrick Schaumont. Vol. 7428. Lecture Notes in Computer Science. Springer, 2012, pp. 408–425. ISBN: 978-3-642-33026-1 (cit. on p. 21).

[MMD+08] Mohamed Saied Emam Mohamed, Wael Said Abd Elmageed Mohamed, Jintai Ding, and Johannes Buchmann. "MXL2: solving polynomial equations over GF(2) using an improved mutant strategy". In: *Post-Quantum Cryptography – PQCrypto 2008*. Ed. by Johannes Buchmann and Jintai Ding. Vol. 5299. Lecture Notes in Computer Science. Springer, 2008, pp. 203–215 (cit. on p. 172).

[MN07] Mitsuru Matsui and Junko Nakajima. "On the power of bitslice implementation on Intel Core2 processor". In: *Cryptographic hardware and embedded systems—CHES 2007, 9th international workshop, Vienna, Austria, September 10–13, 2007, proceedings*. Ed. by Pascal Paillier and Ingrid Verbauwhede. Vol. 4727. Lecture Notes in Computer Science. Springer, 2007, pp. 121–134. ISBN: 978-3-540-74734-5. URL: `http://www.iacr.org/archive/ches2007/47270121/47270121.pdf` (cit. on p. 20).

[MOG15] Ingo von Maurich, Tobias Oder, and Tim Güneysu. "Implementing QC-MDPC McEliece Encryption". In: *ACM Transactions on Embedded Computing Systems* 14 (2015), p. 44 (cit. on pp. 39–46, 49, 54).

[Moh01] Tzuong-Tsieng Moh. "On the method of XL and its inefficiency to TTM". 2001. IACR Cryptology ePrint archive: 2001/047 (cit. on p. 154).

[Mon95] Peter Lawrence 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, 1995, pp. 106–120 (cit. on p. 154).

[Moo14a] Dustin Moody. "Development of FIPS 186: digital signatures (and elliptic curves)". 2014. URL: `http://csrc.nist.gov/groups/ST/crypto-review/documents/FIPS_186_and_Elliptic_Curves_052914.pdf` (cit. on p. 113).

[Moo14b] Andrew "Floodberry" Moon. "Optimized implementations of Poly1305, a fast message-authentication-code". 2014. URL: `https://github.com/floodyberry/poly1305-opt` (cit. on p. 57).

[Moo15] Andrew "Floodyberry" Moon. "Implementations of a fast elliptic-curve digital signature algorithm". accessed 16 March 2015. 2015. URL: `https://github.com/floodyberry/ed25519-donna` (cit. on pp. 89, 90).

[MR02] Sean Murphy and Matthew John Barton 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, 2002, pp. 1–16 (cit. on p. 153).

[MTS+13] Rafael Misoczki, Jean-Pierre Tillich, Nicolas Sendrier, and Paulo S. L. M. Barreto. "MDPC-McEliece: New McEliece variants from moderate density parity-check codes". In: *IEEE International Symposium on Information Theory* (2013), pp. 2069–2073 (cit. on pp. 39–43, 50, 53, 54).

[Nie07] Jesper Buus Nielsen. "Extending oblivious transfers efficiently - how to get robustness almost for free". 2007. IACR Cryptology ePrint archive: 2007/215 (cit. on p. 79).

[Nie12] Ruben Niederhagen. "Parallel Cryptanalysis". PhD thesis. Eindhoven University of Technology, 2012. URL: `http://polycephaly.org/thesis/index.shtml` (cit. on pp. 155, 162, 163).

[Nie86] Harald Niederreiter. "Knapsack-type cryptosystems and algebraic coding theory". In: *Problems of Control and Information Theory* 15 (1986), pp. 159–166 (cit. on pp. 19, 34, 39).

[NIS00] National Institute for Standards and Technology. "FIPS PUB 186-2: Digital signature standard". 2000. URL: `http://csrc.nist.gov/publications/fips/archive/fips186-2/fips186-2.pdf` (cit. on pp. 108, 111, 112, 121).

[NIS13] National Institute for Standards and Technology. "FIPS PUB 186-4: Digital signature standard (DSS)". 2013. URL: `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf` (cit. on pp. 111, 131, 133, 139).

[NNO$^+$12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. "A new approach to practical active-secure two-party computation". In: *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings.* 2012, pp. 681–700. DOI: `10.1007/978-3-642-32009-5_40` (cit. on pp. 77, 79).

[NP01] Moni Naor and Benny Pinkas. "Efficient oblivious transfer protocols". In: *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA.* 2001, pp. 448–457. URL: `http://dl.acm.org/citation.cfm?id=365411.365502` (cit. on p. 79).

[NSA05] National Security Agency. "Suite B cryptography / cryptographic interoperability". 2005. URL: `https://web.archive.org/web/20150724150910/https://www.nsa.gov/ia/programs/suiteb_cryptography/` (cit. on pp. 108, 111, 137).

[OLA$^+$13a] Thomaz Oliveira, Juilo López, Diego F. Aranha, and Francisco Rodríguez-Henríquez. "Two is the fastest prime". 2013 (cit. on p. 23).

[OLA$^+$13b] Thomaz Oliveira, Julio López, Diego F. Aranha, and Francisco Rodríguez-Henríquez. "Lambda coordinates for binary elliptic curves". In: *Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings.* Ed. by Guido Bertoni and Jean-Sébastien Coron. Vol. 8086. Lecture Notes in Computer Science. Springer, 2013, pp. 311–330. ISBN: 978-3-642-40348-4. DOI: `10.1007/978-3-642-40349-1_18` (cit. on p. 95).

[OS09] Raphael Overbeck and Nicolas Sendrier. "Code-based cryptography". In: *Post-quantum cryptography*. Ed. by Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. Springer, 2009, pp. 95–145. ISBN: 978-3-540-88701-0 (cit. on pp. 35, 36).

[OSC10a] State Commercial Cryptography Administration (OSCCA), China. "Public key cryptographic algorithm SM2 based on elliptic curves". Dec. 2010. URL: `http://www.oscca.gov.cn/UpFile/2010122214822692.pdf` (cit. on pp. 111, 119).

[OSC10b] State Commercial Cryptography Administration (OSCCA), China. "Recommanded curve parameters for public key cryptographic algorithm SM2 based on elliptic curves". Dec. 2010. URL: `http://www.oscca.gov.cn/UpFile/2010122214836668.pdf` (cit. on pp. 111, 119).

[Ove06] Raphael Overbeck. "Statistical Decoding Revisited". In: *Information Security and Privacy, 11th Australasian Conference, ACISP 2006, Melbourne, Australia, July 3-5, 2006, Proceedings*. Ed. by Lynn Margaret Batten. Vol. 4058. Lecture Notes in Computer Science. Springer, 2006, pp. 283–294. ISBN: 3-540-35458-1 (cit. on p. 43).

[Paa97] Christof Paar. "Optimized arithmetic for Reed–Solomon encoders". 1997. URL: `http://www.emsec.rub.de/media/crypto/veroeffentlichungen/2011/01/19/cnst.ps` (cit. on p. 61).

[Pat75] Nicholas J. Patterson. "The algebraic decoding of Goppa codes". In: *IEEE Transactions on Information Theory* 21 (1975), pp. 203–207 (cit. on p. 35).

[Per12] Edoardo Persichetti. "Improving the efficiency of code-based cryptography". PhD thesis. 2012. URL: `http://persichetti.webs.com/publications` (cit. on pp. 35, 39, 54).

[Per13] Edoardo Persichetti. "Secure and Anonymous Hybrid Encryption from Coding Theory". In: *Post-quantum cryptography*. Ed. by Philippe Gaborit. Vol. 7932. Lecture Notes in Computer Science. Springer and Springer, 2013, pp. 174–187. URL: `http://persichetti.webs.com/publications` (cit. on pp. 39, 54).

[Pet10] Christiane Peters. "Information-set decoding for linear codes over $\mathbf{F}_q$". In: *Post-quantum cryptography, third international workshop, PQCrypto, Darmstadt, Germany, May 25–28, 2010*. Ed. by Nicolas Sendrier. Vol. 6061. Lecture Notes in Computer Science. Springer, 2010, pp. 81–94. IACR Cryptology ePrint archive: 2009/589 (cit. on p. 22).

[PL07] Steffen Peter and Peter Langendörfer. "An efficient polynomial multiplier in GF(2^m) and its application to ECC designs". In: *Design, automation & test in Europe conference & exhibition, 2007. DATE '07*. IEEE, 2007. URL: `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?isnumber=4211749&arnumber=4211979&count=305&index=229` (cit. on p. 58).

[PQ12] Christophe Petit and Jean-Jacques Quisquater. "On polynomial systems arising from a weil descent". In: *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings.* Ed. by Xiaoyun Wang and Kazue Sako. Vol. 7658. Lecture Notes in Computer Science. Springer, 2012, pp. 451–466. ISBN: 978-3-642-34960-7. DOI: `10.1007/978-3-642-34961-4_28` (cit. on p. 95).

[PS12] Emmanuel Prouff and Patrick Schaumont, eds. "Cryptographic hardware and embedded systems—CHES 2012—14th international workshop, Leuven, Belgium, September 9–12, 2012, proceedings". Vol. 7428. Lecture Notes in Computer Science. Springer, 2012. ISBN: 978-3-642-33026-1.

[PVW08] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. "A framework for efficient and composable oblivious transfer". In: *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings.* 2008, pp. 554–571. DOI: `10.1007/978-3-540-85174-5_31` (cit. on p. 79).

[Rab81] Michael O. Rabin. "How to exchange secrets with oblivious transfer". In: *Technical Report TR-81, Aiken Computation Lab, Harvard University* (1981). IACR Cryptology ePrint archive: 2005/187 (cit. on p. 79).

[RK03] Francisco Rodríguez-Henríquez and Çetin Kaya Koç. "On fully parallel Karatsuba multipliers for GF(2^m)". In: *Proceedings of the international conference on computer science and technology.* Ed. by S. Sahni. Acta Press, 2003, pp. 405–410 (cit. on p. 58).

[RS62] John Barkley Rosser and Lowell Schoenfeld. "Approximate formulas for some functions of prime numbers". In: *Illinois Journal of Mathematics* 6 (1962), pp. 64–94. ISSN: 0019–2082. URL: `https://projecteuclid.org/euclid.ijm/1255631807` (cit. on p. 114).

[SC12] Reihaneh Safavi-Naini and Ran Canetti, eds. "Advances in cryptology—CRYPTO 2012—32nd annual cryptology conference, Santa Barbara, CA, USA, August 19–23, 2012, proceedings". Vol. 7417. Lecture Notes in Computer Science. Springer, 2012. ISBN: 978-3-642-32008-8.

[Sch15] Thomas Schneider. "Personal communication". 2015 (cit. on p. 80).

[Sch77] Arnold Schönhage. "Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2". In: *Acta Informatica* 7.4 (1977), pp. 395–398 (cit. on p. 161).

[Sco99] Michael Scott. "Re: NIST announces set of Elliptic Curves". 1999. URL: `https://groups.google.com/forum/message/raw?msg=sci.crypt/mFMukSsORmI/FpbHDQ6hM_MJ` (cit. on p. 125).

[Sem15] Igor A. Semaev. "New algorithm for the discrete logarithm problem on elliptic curves". In: *CoRR* abs/1504.01175 (2015). URL: `http://arxiv.org/abs/1504.01175` (cit. on p. 95).

[Sen10] Nicolas Sendrier, ed. "Post-quantum cryptography, third international workshop, PQCrypto, Darmstadt, Germany, May 25–28, 2010". Vol. 6061. Lecture Notes in Computer Science. Springer, 2010.

[SG14] Pascal Sasdrich and Tim Güneysu. "Efficient elliptic-curve cryptography using Curve25519 on reconfigurable devices". In: *ARC 2014*. Ed. by Diana Goehringer, Marco Domenico Santambrogio, João M. P. Cardoso, and Koen Bertels. Vol. 8405. Lecture Notes in Computer Science. Springer, 2014, pp. 25–36. ISBN: 978-3-319-05959-4. DOI: 10.1007/978-3-319-05960-0_3. URL: https://www.hgi.rub.de/media/sh/veroeffentlichungen/2014/03/25/paper_arc14_curve25519.pdf (cit. on p. 142).

[She59] Donald L. Shell. "A high-speed sorting procedure". In: *Communications of the ACM* 2 (1959), pp. 30–32 (cit. on p. 31).

[Sho01] Victor Shoup. "A proposal for an ISO standard for public key encryption (version 2.1)". 2001 (cit. on p. 35).

[Sho97] Peter W. Shor. "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer". In: *SIAM J. Comput.* 26.5 (1997), pp. 1484–1509. DOI: 10.1137/S0097539795293172. URL: http://dx.doi.org/10.1137/S0097539795293172 (cit. on p. 2).

[Sil09] Joseph H. Silverman. "The arithmetic of elliptic curves". Graduate Texts in Mathematics 106. Springer-Verlag, 2009 (cit. on p. 112).

[Ste+15] W. A. Stein et al. "Sage Mathematics Software (version 6.8)". The Sage Development Team. 2015. URL: http://www.sagemath.org (cit. on pp. 120, 123).

[Str10] Falko Strenzke. "A timing attack against the secret permutation in the McEliece PKC". In: *Post-quantum cryptography, third international workshop, PQCrypto, Darmstadt, Germany, May 25–28, 2010*. Ed. by Nicolas Sendrier. Vol. 6061. Lecture Notes in Computer Science. Springer, 2010, pp. 95–107 (cit. on p. 19).

[Str11] Falko Strenzke. "Timing attacks against the syndrome inversion in code-based cryptosystems". 2011 (cit. on p. 19).

[Str12] Falko Strenzke. "Fast and secure root finding for code-based cryptosystems". In: *Cryptology and network security—11th international conference, CANS 2012, Darmstadt, Germany, December 12–14, 2012, proceedings*. Ed. by Josef Pieprzyk, Ahmad-Reza Sadeghi, and Mark Manulis. Vol. 7712. Lecture Notes in Computer Science. Springer, 2012, pp. 232–246. ISBN: 978-3-642-35403-8. IACR Cryptology ePrint archive: 2011/672.pdf (cit. on pp. 19, 25).

[Tea10] Circuit Minimization Team. "Multiplication circuit for GF(256) with irreducible polynomial $X^8 + X^4 + X^3 + X + 1$". 2010. URL: http://cs-www.cs.yale.edu/homes/peralta/CircuitStuff/slp_84310.txt (cit. on p. 60).

[Thé03] Nicolas Thériault. “Index calculus attack for hyperelliptic curves of small genus”. In: *Advances in Cryptology - ASIACRYPT 2003, 9th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, November 30 - December 4, 2003, Proceedings*. Ed. by Chi-Sung Laih. Vol. 2894. Lecture Notes in Computer Science. Springer, 2003, pp. 75–92. ISBN: 3-540-20592-6. DOI: `10.1007/978-3-540-40061-5_5` (cit. on p. 96).

[Tho02] Emmanuel Thomé. “Subquadratic computation of vector generating polynomials and improvement of the block Wiedemann algorithm”. In: *Journal of Symbolic Computation* 33.5 (2002), pp. 757–775 (cit. on pp. 160, 162).

[WC81] Mark N. Wegman and J. Lawrence Carter. “New hash functions and their use in authentication and set equality”. In: *Journal of Computer and System Sciences* 22 (1981), pp. 265–279. ISSN: 0022-0000 (cit. on p. 65).

[Wie83] Stephen Wiesner. “Conjugate coding”. In: *SIGACT News* 15.1 (Jan. 1983), pp. 78–88. ISSN: 0163-5700. DOI: `10.1145/1008908.1008920` (cit. on p. 79).

[Wie86] Douglas H. Wiedemann. “Solving sparse linear equations over finite fields”. In: *IEEE Transactions on Information Theory* 32.1 (1986), pp. 54–62 (cit. on p. 154).

[Wik15a] Wikipedia. “NewDES”. (accessed 27 September 2015). 2015. URL: `https://en.wikipedia.org/wiki/NewDES` (cit. on p. 134).

[Wik15b] Wikipedia. “Nothing up my sleeve number”. (accessed 27 September 2015). 2015. URL: `https://en.wikipedia.org/wiki/Nothing_up_my_sleeve_number` (cit. on pp. 109, 135).

[Wik16a] Wikipedia. “Barrel shifter — Wikipedia, The Free Encyclopedia”. Note: accessed 2 February 2016. 2016. URL: `https://en.wikipedia.org/wiki/BarrelShifter` (cit. on p. 49).

[Wik16b] Wikipedia. “RdRand — Wikipedia, The Free Encyclopedia”. Note: accessed 2 February 2016. 2016. URL: `https://en.wikipedia.org/wiki/Rdrand` (cit. on p. 45).

[WP06] André Weimerskirch and Christof Paar. “Generalizations of the Karatsuba algorithm for efficient implementations”. 2006. IACR Cryptology ePrint archive: 2006/224 (cit. on p. 58).

[WS12] Xiaoyun Wang and Kazue Sako, eds. “Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings”. Vol. 7658. Lecture Notes in Computer Science. Springer, 2012. ISBN: 978-3-642-34960-7. DOI: `10.1007/978-3-642-34961-4`.

[WZ88] Yao Wang and Xuelong Zhu. “A fast algorithm for Fourier transform over finite fields and its VLSI implementation”. In: *IEEE Journal on Selected Areas in Communications* 6 (1988), pp. 572–577 (cit. on p. 11).

[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, 2005, pp. 32–35 (cit. on p. 154).

[YCB$^+$07] Bo-Yin Yang, Chia-Hsin Chen, Daniel Julius Bernstein, and Jiun-Ming Chen. “Analysis of QUAD”. In: *Fast Software Encryption*. Ed. by Alex Biryukov. Vol. 4593. Lecture Notes in Computer Science. Springer, 2007, pp. 290–308 (cit. on p. 153).

[YCC04] Bo-Yin Yang, Jiun-Ming Chen, and Nicolas Tadeusz Courtois. “On asymptotic security estimates in XL and Gröbner bases-related algebraic cryptanalysis”. In: *Information and Communications Security – ICICS 2004*. Ed. by Javier Lopez, Sihan Qing, and Eiji Okamoto. Vol. 3269. Lecture Notes in Computer Science. Springer, 2004, pp. 281–286 (cit. on p. 154).

[YGH16] Yuval Yarom, Daniel Genkin, and Nadia Heninger. “Cachebleed: A timing attack on openssl constant time RSA”. In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 224. URL: `http://eprint.iacr.org/2016/224` (cit. on p. 8).