Time-Area Efficient Hardware Architectures for Cryptography and Cryptanalysis

Dissertation

---

zur
Erlangung des Grades eines
Doktor-Ingenieurs
der
Fakultät für Elektrotechnik und Informationstechnik
an der Ruhr-Universität Bochum

---

von Martin Novotný
Bochum, Februar 2009

Author's contact information:
novotnym@fit.cvut.cz

Thesis Advisor:	**Prof. Dr.-Ing. Christof Paar** Ruhr-University Bochum, Germany
Secondary Referee:	**Ing. Jan Schmidt, Ph.D.** CTU in Prague, Czech Republic
Thesis submission:	February 2, 2009
Thesis defense:	April 30, 2009

Abstract

In the first part of the thesis we focus on scalable arithmetic units operating over the binary finite field $GF(2^m)$ with a normal basis representation of the field elements. The scalability is crucial in applications of different kind – small, low power embedded devices as opposed to high-throughput backbone applications.

Although scaling by digit-serialization is a well-known method, its application to normal-basis multipliers brings problems with irregularities. Little has been done for the case when the digit width does not divide the degree of the field, although this situation is unavoidable in cryptographic applications.

In this thesis we present four architectures of the digit-serial normal basis multiplier that we developed. We demonstrate digit-serialization on the pipelined multiplier by Agnew et al., however, these methods are also applicable to other multiplier structures, e.g. the multiplier by Kwon et al. All architectures can be implemented for any digit width. Our evaluation shows that their advantages are complementary with respect to the digit width.

Based on the scalable multiplier design, we extended our work to build an entire scalable arithmetic unit. Only a shifter has to be added to support inversion. It is scaled by the number of shifts implemented in hardware. This is a special case of another little-studied problem: scaling a sub-unit in the presence of another one, dominating the design in area and time. We present an optimization method applicable to such cases.

In the second part of the thesis we focus on cryptanalysis of GSM communication, which is encrypted with A5/1 cipher. We present two attacks against A5/1 cipher. Both attacks are supported by an existing low-cost special-purpose hardware device COPACOBANA. They represent the first real-world implementations of attacks against A5/1 reported in open literature.

The first attack is a guess-and-determine attack revealing the internal state of A5/1 in about 6 hours on average (and about 12 hours at the worst-case). To mount the attack only 64 consecutive bits of a known keystream are required and we do not need any precomputed data. We also propose an optimized version of the attack. Both plain and optimized version of the attack have been fully implemented and tested on our target platform.

The second attack is a time-memory-data trade-off attack revealing the internal state of A5/1 with certain probability in a matter of minutes. COPACOBANA is used in both the precomputation phase and the online phase of the attack. When designing the precomputation engine, we have utilized the features of underlying FPGA architecture to gain the maximum performance. Here proposed design approach can be reused when designing similar attacks against other stream ciphers.

Keywords:

Public Key Cryptography, Elliptic Curve Cryptography, Arithmetic Unit, Binary Finite Fields $GF(2^m)$, Normal Basis, Multiplication, Inversion, Cryptanalysis, Brute-Force Attack, TMDTO Attack, A5/1, COPACOBANA, FPGA

Kurzfassung

Die vorliegende Dissertation gliedert sich thematisch in zwei Teile. Der erste Teil beschäftigt sich mit skalierbaren Arithmetikeinheiten über endlichen Körpern der Form $GF(2^m)$ in Normalbasis-Repräsentation. Die Skalierbarkeit dieser Architekturen ist dabei essentiell, um den Anforderungen unterschiedlicher Anwendungen – zum Beispiel kleine, eingebettete Systeme mit geringer Leistungsaufnahme im Gegensatz zu Backbone-Systemen mit hohem Datendurchsatz – gerecht zu werden.

Skalierung durch Serialisierung ist eine wohlbekannte Methode, jedoch bringt ihre Anwendung auf Normalbasis-Multiplizierer Probleme durch bestimmte Irregularitäten mit sich. In der einschlägigen Literatur wurde der Fall, dass die Wortbreite kein Teiler des Erweiterungsgrades ist, so gut wie nicht behandelt, obwohl dieser Fall für kryptographische Anwendungen praktisch unvermeidbar ist.

In dieser Arbeit werden vier neuartige „digit-serial“ Normalbasis-Multiplizierer vorgestellt. Dabei handelt es sich um serialisierte Varianten des Multiplizierers von Agnew et al. Es ist erwähnenswert, dass die dazu entwickelten Serialisierungsmethoden auch auf anderere Architekturen, wie zum Beispiel den Multiplizierer von Kwon et al., angewandt werden können. Alle vorgestellten Architekturen können für beliebige Wortbreiten implementiert werden. Die durchgeführte Evaluierung zeigt, dass die Vorteile der Architekturen sich hinsichtlich der Wortbreite komplementär verhalten.

Als weiterer Forschungsbeitrag wird eine vollständig skalierbare Arithmetikeinheit entwickelt, die auf den vorgestellten Multiplizierern basiert. Diese Einheit wurde darauf optimiert möglichst wenig Chipfläche einzunehmen und ist in der Tat nur unwesentlich größer als der enthaltene Multiplizierer. In diesem Kontext stellt sich das bisher wenig untersuchte Problem der Dimensionierung eines Bausteins in der Gegenwart eines zweiten, der das Gesamtsystem

bezüglich Fläche und Zeit dominiert. Eine allgemeine Optimierungsstrategie für derartige Fälle wird vorgeschlagen.

Der zweite Teil dieser Arbeit beschäftigt sich mit der hardware-basierten Kryptanalyse der im GSM-Mobilfunknetz eingesetzten Stromchiffre A5/1. Basierend auf dem kürzlich vorgestellten, kosteneffizienten FPGA-Cluster COPACOBANA werden zwei Angriffe gegen A5/1 realisiert. Im Gegensatz zu bisherigen Arbeiten handelt es sich hierbei um zwei äußerst praktikable und vollständig implementierte Attacken.

Der erste Angriff fällt in die Klasse der sogenannten „guess-and-determine“ Attacken und ist in der Lage den geheimen internen Zustand der Chriffre in durchschnittlich 6 Stunden (12 Stunden im „worst-case“) zu bestimmen. Um diesen Angriff durchzuführen werden nur 64 (aufeinanderfolgende) Bits des Schlüsselstroms und keinerlei Vorberechnungen benötigt. Die vorgeschlagene Attacke, sowie eine optimierte Variante wurden komplett auf COPACOBANA implementiert und getestet.

Beim zweiten Angriff handelt es sich um eine „time-memory-data tradeoff“ Attacke. Mittels der entwickelten hardware-basierten Realisierung dieses Angriffs kann, mit einer gewissen aber signifikanten Wahrscheinlichkeit, der Initialzustand von A5/1 innerhalb von Minuten wiederhergestellt werden. Die Spezialhardware COPACOBANA wird sowohl für die benötigten (umfangreichen) Vorberechnungen als auch für den eigentlichen Angriff eingesetzt. Idealerweise könnte das hier vorgeschlagene Design als Referenz für gleichartige Angriffe auf Stromchiffren dienen.

Schlüsselworte:

Public-Key-Kryptographie, Elliptische-Kurven-Kryptographie, Hardwarearithmetik, Endliche Körper, Normalbasen, Multiplikation, Inversion, Kryptanalyse, Brute-Force Attacken, TMDTO Attacken, A5/1, COPACOBANA, FPGA

Acknowledgements

First of all, I would like to thank to both my supervisors, Christof Paar and Jan Schmidt.

I would like to thank Jan Schmidt for initiating my research, for sometimes pushing me to do the right things, and for overall support. Without his ideas, help and collaboration, the first part of this thesis would be infeasible.

Christof Paar gave me the opportunity to join his research group for 18 months. The experience I got in the fruitful atmosphere of the group is unique. Christof's constant encouragement and effort as the thesis supervisor contributed substantially to the quality and completeness of the thesis. I am also grateful for his generous help with the funding of my stay.

Finishing this thesis would be infeasible without the help of Irmgard Kühn. She was generous with her help and support during my whole research stay. Horst Edelmann helped me with all technical problems and gave me good tips for trips around the Ruhr Area.

Tim Güneysu and Jan Pelzl initiated my stay in Bochum. With Tim, we worked on COPACOBANA-related problems. Andy Rupp, Timo Gendrullis and I collaborated on cryptanalysis of A5/1. We also discussed some problems with Andrey Bogdanov. Our work led to the results which are summarized in the second part of this thesis.

I had the privilege to share a lot of fun with Timo Kasper. I still reminisce the voyages which we made with Francesco Regazzoni to explore the Ruhr Area and Germany. Thanks to Amir Moradi and his wife Shakila I could extend my knowledge about culture and life of people in different parts of the world. I would also like to extend my thanks to Axel Poschmann, Thomas Eisenbarth, Marko Wolf, Kerstin Lemke-Rust and many others.

I am also thankful to my landlord Mr. Falke and his family. Their house in Bochum became my second home for the whole period of my stay.

I am grateful to the head of our department, Hana Kubátová, for her unconditional help and support. Jiří Douša and Alois Pluháček were my great teachers who became my colleagues. I am also glad to have colleagues who create a cosy environment, namely Petr Fišer, Radek Dobiáš and Pavel Kubalík.

My thanks also go to many other people whom I forgot to mention, but who helped me in my life and work or who simply make this world better.

However, my greatest thanks go to my family. It is my parents who provided for my education and who have been supporting me throughout the my whole life. My nephews Ondřej and Marek, sons of my brother and my wonderful sister-in-law, are a constant source of happiness. My partner Mirek encouraged me to get the experience in Bochum. The 18 month I spent there, far away from home, were hard period for both of us, but Mirek has shown extraordinary patience, love and support.

Thank you!

To my partner and my family.

Contents

List of Tables

List of Algorithms

List of Figures

Chapter 1

Introduction

Cryptography increasingly finds its application area in everyday life. Bank transfers, identification cards, admittance systems, communication, the Internet, personal data, databases, they all need to be protected against unauthorized access.

Cryptanalysis, as a complementary discipline to cryptography, is in public view connected with breaking the ciphers for espionage purposes, money thefts, private data surveys etc. However, it is not necessarily that. If done seriously and if all results are published, then the cryptanalysis significantly contributes to cryptography by showing the weaknesses of cryptographic primitives or protocols used in cryptography. Such weaknesses can be then improved, e.g. by replacing the cipher with the stronger one.

Both cryptography and cryptanalysis demand for efficient hardware modules. For example, the cryptographic modules are necessary in RFID tags, used in transportation, in supermarkets, etc. The tag is powered from the electromagnetic field provided by the reader. The communication between the tag and the reader should be accomplished in a reasonable time. Therefore, the cryptographic hardware inside the tag must be cheap, fast, energy-efficient, and providing sufficient cryptographic strength at the same time.

Efficient hardware modules are necessary also in cryptanalysis. For example, when mounting an attack against certain cipher, we are typically given the limited budget and/or the limited hardware resources. Efficient implementation of the hardware modules allows for faster attack and, consequently, better cost-performance ratio.

In this thesis we contribute to both cryptography and cryptanalysis. From each field we have chosen one specific topic. In the first part of the thesis we focus on hardware architectures operating over elements of binary finite fields with normal basis representation. Such architectures are applicable e.g. in Elliptic Curve Cryptography, which increasingly finds its application area e.g. in bank cards, as a replacement of the RSA cipher. We propose the new structure of the normal basis arithmetic unit, which is both small and scalable. The scalability option allows the designer to meet the design constraints optimally.

In the second part of the thesis we focus on cryptanalysis of A5/1 cipher used in GSM communication. We describe hardware architectures of our two attacks on A5/1 cipher. The attacks have been implemented for a special-purpose hardware COPACOBANA. The attacks are designed to utilize both the properties of the cipher and the properties of the programmable devices used in COPACOBANA. Presented design approaches can be reused when designing attacks against similar (stream) ciphers.

Part I

Design Methods for Scalable Arithmetic Units over Binary Fields with Normal Basis

Chapter 2

Introduction to Part I

History of cryptography is old. In antiquity, middle ages, or in 20th century, there always were messages that had to be encrypted. However, with growing development of information technology in past few decades, the importance of cryptographic systems has increased. Bank transfers, identification cards, admittance systems, communication, the Internet, personal data, databases, they all need to be protected against unauthorized access.

While symmetric cryptosystems use the same key for both encryption and decryption, the asymmetric cryptosystems, based on the idea of Diffie and Hellman [DH76], use two keys. One key — public key — is used for encryption, while the other key — private key — is used for decryption. A nice idea of decrypting messages only with the knowledge of secret private key is replaced with the necessity of higher computational complexity of asymmetric cryptosystems in comparison to the symmetric ones. Therefore, many cryptographic systems, namely those used for transfering large amounts of data, combine advantages of both classes of ciphers. Before the data transfer starts, the key of a symmetric cipher is exchanged by both sides. For this key exchange, the asymmetric cryptosystem is used. Then the data is transferred using the symmetric cipher. Asymmetric cryptosystems are also used in cases where the initial dialog between both sides is impossible (e.g. for the encryption of e-mail messages), or for the authentication — digital signature schemes [FIP00], [ANS97], [ANS98a], [ANS98b] or smart cards [Mas97] can be mentioned here as typical examples.

The RSA cipher [RSA78] is probably the most frequently used asymmetric cipher in currently used applications. Its algorithm is based on a factorization problem. Other algorithms are based either on a discrete logarithm problem (DLP) [DH76] or an elliptic curve discrete logarithm problem (ECDLP) [Mil86], [Kob87].

The elliptic curve cryptosystems (ECC) need significantly shorter keys to achieve the same cryptographic strength as the classical RSA; e.g. the 160-bit ECC has the same cryptographic strength as the 1024-bit RSA [Cer97]. This fact is very important in applications such as smart cards, where the size of hardware or energy consumption is crucial. For this advantage, the EC cryptosystems are commercially more and more popular.

Elliptic curves used for cryptographic purposes have point coordinates which are elements of finite fields, $GF(q)$. Generally, finite fields are defined by a characteristic p and a degree m, marked $GF(p^m)$, where the characteristic p is a prime number and the degree m is an integer. In elliptic curve cryptography particularly prime fields $GF(p)$ and binary fields $GF(2^m)$ are used for a simpler implementation of arithmetic operations. Brief taxonomy may be found e.g. in [Paa99].

In our work we focused on binary fields, $GF(2^m)$. Field elements in binary fields are represented either in a polynomial basis or in a normal basis. The choice of the basis has a strong impact on hardware, each of them offering different advantages. Arithmetic units operating over the normal basis require smaller amount of hardware resources [A.1] in comparison to the units operating over the polynomial basis. On the other hand, arithmetic units operating over the polynomial basis are more common, more flexible and they better integrate with integer arithmetics in comparison to the units operating over normal basis. The main goal of this part of the thesis was a development of hardware architectures operating over the normal basis that would be as flexible as those operating over the polynomial basis.

Some applications require the cryptographic system to be as fast as possible, in other applications area and/or power consumption are strongly limited, in yet another applications the data throughput should be "the right one", while the minimum area and energy consumption are desirable. To meet different design constraints the designer has to choose different area/throughput trade-offs. The flexibility of the system in this sense is called *scalability*. The scalability is the crucial characteristics of current cryptographic systems.

The designer expects e.g. no more than a twofold increase in area for a twofold increase in throughput. This is the ideal case. To measure real

systems in this aspect we use *quality factor* as the ratio of throughput to area. Although this is not the only possibility, it is the most common measure.

In this part of the thesis we describe a normal basis cryptographic arithmetic unit that we developed. This arithmetic unit is able to perform two crucial operations, multiplication and inversion in normal basis. This unit has been designed to be as small as possible; in fact, it is slightly bigger than the multiplier itself. The arithmetic unit contains two principal subunits, a multiplier and a shifter used in an inversion algorithm. Both subunits may be scaled as much as the designer needs. This extended scalability allows the designer to tune the cryptographic system to fit the design constraints optimally.

This part of the thesis is structured as follows: In Chapter 3, we summarize basic mathematical backgrounds necessary for elliptic curve cryptography. In Chapter 4, we bring an overview on essential algorithms and arithmetic architectures for binary finite fields. In Chapter 5, we describe the multiplication/inversion unit that we developed. As mentioned above, the unit consists of two subunits, the multiplier and the shifter. As the scalability options of a standard multiplier are limited, we developed four architectures of multipliers that can be scaled as much as necessary. These architectures are described in Chapter 6. In the last Chapter 7, we discuss the shifter and its scalability options.

Chapter 3

Mathematical Background

In this chapter we summarize mathematical foundations of an elliptic curve cryptography. We remind the definition of an elliptic curve, element operations over elliptic curves and elliptic curves with point coordinates as the elements of finite fields, namely binary fields. The information given here was acquired from annex A of IEEE1363 standard [IEE00] and other sources, e.g. [MBG+93].

3.1 Elliptic Curves

Elliptic curve E over real numbers is a set of points satisfying the Weierstrass equation

$$y^2 = x^3 + ax + b, \tag{3.1}$$

along with an additional element called the *point at infinity* (denoted $\bigcirc$).

A basic operation defined on an elliptic curve is a point addition. Its geometrical construction is outlined in Figure 3.1.

Definition 3.1 (Point addition — geometrical approach)**.** *Let P_1 and P_2 be two points of elliptic curve E, $P_1, P_2 \in E$, with coordinates $P_1 = [x_1, y_1]$ and $P_2 = [x_2, y_2]$. Let l_1 be a secant of E that intersects E at points P_1 and P_2. Then l_1 intersects E at a third point $-Q = [x_Q, -y_Q]$. Point $Q = [x_Q, y_Q]$ is a result of point addition, $Q = P_1 + P_2$.*

Point $-Q = [x_Q, -y_Q]$ is the inverse of point $Q = [x_Q, y_Q]$.

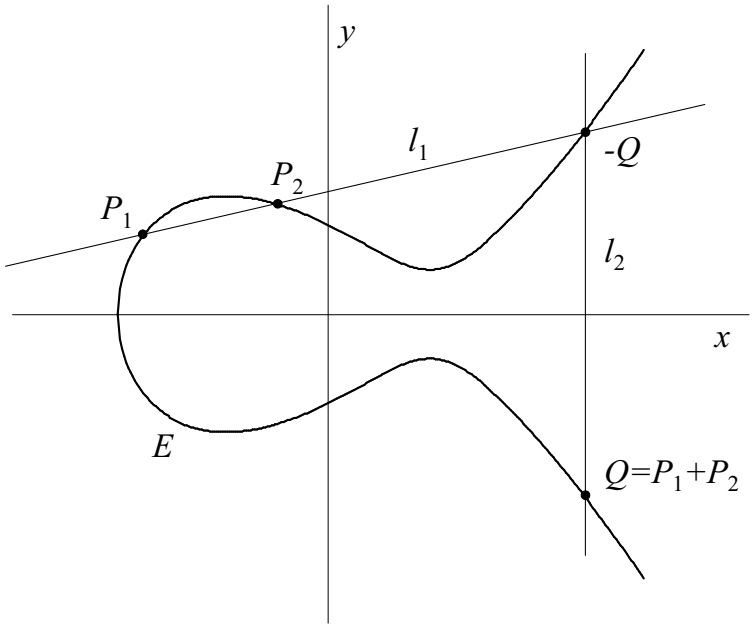

Figure 3.1: Point addition on an elliptic curve

Definition 3.2 (Point doubling — geometrical approach)**.** *Let $P_1 = P_2 = P$ be a point of an elliptic curve E, $P \in E$. Let l_1 be a tangent of E that intersects E at a point P. Then l_1 intersects E at a point $-Q = [x_Q, -y_Q]$. Point $Q = [x_Q, y_Q]$ is a result of a point doubling, $Q = 2P$.*

The point at infinity $\bigcirc$ plays the role of a neutral element:

$$P + \bigcirc = P,$$
$$P + (-P) = \bigcirc.$$

Definition 3.3 (Point addition — algebraic approach)**.** *Let P_1 and P_2 be two points of an elliptic curve E, $P_1, P_2 \in E$, with coordinates $P_1 = [x_1, y_1]$ and $P_2 = [x_2, y_2]$. Then the result of addition of points P_1 and P_2 is a point $Q = P_1 + P_2$ with coordinates x_Q, y_Q:*

$$x_Q = \lambda^2 - x_1 - x_2$$
$$y_Q = \lambda(x_1 - x_Q) - y_1,$$

where

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1}.$$

Definition 3.4 (Point doubling - algebraic approach)**.** *Let $P_1 = P_2 = P$ be a point of an elliptic curve E, $P \in E$, with coordinates $P = [x_1, y_1]$. Then the*

result of the doubling of a point P is a point $Q = 2P = P + P$ with coordinates x_Q, y_Q:

$$x_Q = \lambda^2 - 2x_1$$
$$y_Q = \lambda(x_1 - x_Q) - y_1,$$

where

$$\lambda = \frac{3x_1^2 + a}{2y_1}.$$

3.2 Elliptic Curves over Binary Finite Fields, $GF(2^m)$

For cryptographic purposes, the coordinates x and y of the points on an elliptic curve are expressed as elements of a finite field $GF(q)$. Generally, the finite field $GF(q)$ has $q = p^m$ elements, marked $GF(p^m)$. Characteristic p is a prime number, the degree m is a positive integer number. In cryptography, particularly prime fields $GF(p)$ and binary fields $GF(2^m)$ are used for simpler implementation of arithmetic operations.

We focused on elliptic curves over binary fields $GF(2^m)$, where field elements can be expressed as m-bit vectors. For the binary fields $GF(2^m)$, the Weierstrass equation is

$$y^2 + xy = x^3 + ax^2 + b, \tag{3.2}$$

where a and b are elements of $GF(2^m)$ with $b \neq 0$.

Also the point addition and point doubling are redefined for the elliptic curves over binary fields.

Definition 3.5 (Point addition on the elliptic curve over the binary field)**.** *Let P_1 and P_2 be two points of a elliptic curve E, $P_1, P_2 \in E$, with coordinates $P_1 = [x_1, y_1]$ and $P_2 = [x_2, y_2]$; $x_1, x_2, y_1, y_2 \in GF(2^m)$. Then the result of the addition of points P_1 and P_2 is a point $Q = P_1 + P_2$ with coordinates $x_Q, y_Q \in GF(2^m)$:*

$$x_Q = a + \lambda^2 + \lambda + x_1 + x_2 \tag{3.3}$$
$$y_Q = \lambda(x_2 + x_Q) + x_Q + y_2, \tag{3.4}$$

where

$$\lambda = \frac{y_2 + y_1}{x_2 + x_1}. \tag{3.5}$$

Definition 3.6 (Point doubling on the elliptic curve over the binary field)**.** *Let $P_1 = P_2 = P$ be a point of the elliptic curve E, $P \in E$, with coordinates $P = [x_1, y_1]$; $x_1, y_1 \in GF(2^m)$. Then the result of the doubling of a point P is a point $Q = 2P = P + P$ with coordinates $x_Q, y_Q \in GF(2^m)$:*

$$x_Q = a + \lambda^2 + \lambda \tag{3.6}$$

$$y_Q = \lambda(x_1 + x_Q) + x_Q + y_1, \tag{3.7}$$

where

$$\lambda = x_1 + \frac{y_1}{x_1}. \tag{3.8}$$

With the knowledge of the point addition we can define derivative operation — scalar point multiplication.

Definition 3.7 (Scalar multiplication of the point)**.** *Let k be a positive integer ($k \in \mathbb{N}$) and P be a point on an elliptic curve E, $P \in E$. Then a scalar multiple $Q = kP$ is a result of adding k copies of P, $Q = kP = P + P + ... + P$.*

The definition of a scalar point multiplication can be extended to k being zero or k being a negative integer: $0P = \bigcirc$, $(-k)P = k(-P)$.

Scalar point multiplication is the main operation used in elliptic curve cryptography, in other words, it is used in *cryptographic primitives.* The EC based cryptography utilizes the fact that for given $k \in \mathbb{N}$ and $P \in E$ it is relatively simple to compute $Q = kP$ (it takes $O(\log k)$ point additions or doublings), while the reverse operation — computation of k for known P and Q — is difficult (it takes $k - 1$ point additions). For the evaluation of the scalar point multiple, the double-and-add method (Horner scheme), the addition-subtraction method outlined in [IEE00] or other methods [LD99], [Mon87] can be used.

Example 3.1. For $k = 41 = 11001_2$, with the double-and-add method the scalar point multiple Q is evaluated after 5 point doublings and 2 point additions:

$$\begin{aligned} Q &= k \times P = 41 \times P = 101001_2 \times P \\ &= ((((((1P) \times 2 + 0P) \times 2 + 1P) \times 2 + 0P) \times 2 + 0P) \times 2 + 1P). \end{aligned}$$

Vice versa, to evaluate k from known points P and Q, we must perform 40 point additions $P+P+P+...+P$ until the result matches the point Q. After that we know the secret value $k = 41$.

The value k is called an *elliptic curve discrete logarithm* (more precise definition of an elliptic curve discrete logarithm is given e.g. in [IEE00]). The problem with evaluation of k for known P and Q is called an *elliptic curve discrete logarithm problem* (ECDLP).

3.3 Operations on Binary Field, $GF(2^m)$

From Equations 3.3 through 3.8 it is evident that the following operations on the elements of $GF(2^m)$ must be implemented:

- addition
- multiplication
- division/inversion
- squaring

If an algorithm for division is not known, then division is performed as multiplication by an inverse element of a divisor (denominator). In that case, an algorithm for inversion is involved. Although squaring is in general derived from multiplication, it is advantageous to consider it as a separate operation, since squaring may be performed faster than multiplication.

Let us remark that in the previous text we operated with so called *affine coordinates* of the points on an elliptic curve. If so called *projective coordinates* are used, than for point addition or point doubling, no division or inversion is necessary. On the other hand, an increased number of multiplications is inevitable. Division, however, is still necessary for conversion from projective to affine coordinates.

Addition of two elements of $GF(2^m)$ is always realized as a bit-wise addition modulo 2 (XOR operation). The realization of other operations depends on a basis representation of the field elements. There are two common families of basis representations for the binary fields: *polynomial basis* representations and *normal basis* representations.

A *polynomial basis* is a set of the form $B = \{t^{m-1}, \ldots, t^2, t^1, t^0\}$. The representation of $GF(2^m)$ via the polynomial basis is carried out by interpreting the bit string $(a_{m-1} \ldots a_2 a_1 a_0)$ as an element $a_{m-1}t^{m-1} + \cdots + a_2t^2 + a_1t + a_0$.

A *normal basis* is a set of the form $B = \{\beta^{2^0}, \beta^{2^1}, \beta^{2^2}, \ldots, \beta^{2^{m-1}}\}$. The representation of $GF(2^m)$ via the normal basis is carried out by interpreting the bit string $(a_0 a_1 a_2 \ldots a_{m-1})$ as the element $a_0\beta + a_1\beta^2 + a_2\beta^4 + \cdots + a_{m-1}\beta^{2^{m-1}}$. For more information about normal bases, see [Gao93] and [MBG+93].

As arithmetic units working over a normal basis representation are smaller and faster than those ones working over a polynomial basis [A.1], we focused entirely on a normal basis in our work.

3.4 Operations on $GF(2^m)$ with a Normal Basis Representation

In our work we entirely focused on a normal basis representation. Here we present algorithms concerning operations in a normal basis mentioned above.

Let

$$B = \{\beta^{2^0}, \beta^{2^1}, \beta^{2^2}, \ldots, \beta^{2^{m-1}}\}$$

be a normal basis in $GF(2^m)$. Let a, b be elements of $GF(2^m)$ with a normal basis B, then

$$a = a_0\beta^{2^0} + a_1\beta^{2^1} + a_2\beta^{2^2} + \cdots + a_{m-1}\beta^{2^{m-1}}$$
$$b = b_0\beta^{2^0} + b_1\beta^{2^1} + b_2\beta^{2^2} + \cdots + b_{m-1}\beta^{2^{m-1}},$$

where $a_i, b_i \in GF(2)$.

3.4.1 Addition

Addition of elements a and b is performed as addition of polynomials $a(\beta)$ and $b(\beta)$. As elements of $GF(2^m)$ are usually represented as m-bit vectors, the addition is equivalent to a bit-wise XOR operation on the vectors a and b.

3.4.2 Multiplication

Multiplication of two elements in $GF(2^m)$ with a normal basis B can be defined by a multiplication matrix $\mathbf{M}$. The multiplication matrix is a square matrix with elements $\lambda_{j,l} \in GF(2)$. The algorithm of finding the multiplication matrix $\mathbf{M}$ for a given normal basis can be found in [IEE00].

The coefficients of a product $c = a \times b$ are

$$c_i = \sum_{j=0}^{m-1} \sum_{l=0}^{m-1} a_{j+l} b_{l+i} \lambda_{jl},\ 0 \leq i \leq m-1, \tag{3.9}$$

where additions and multiplications are performed in $GF(2)$. Consequently, additions are performed as XOR operations and multiplications as AND operations. The indices of a and b are added modulo m.

Equation 3.9 can be rewritten into following forms:

$$c_i = \sum_{j=0}^{m-1} a_{j+l} \sum_{l=0}^{m-1} b_{l+i} \lambda_{jl},\ 0 \leq i \leq m-1, \tag{3.10}$$

or

$$c_i = \sum_{l=0}^{m-1} b_{l+i} \sum_{j=0}^{m-1} a_{j+l} \lambda_{jl},\ 0 \leq i \leq m-1.$$

Let C_N denotes the number of non-zero elements λ_{jl} in $\mathbf{M}$. As obvious, it corresponds to the number of product terms in Equation 3.9 for each c_i. Thus, C_N determines the complexity of multiplication — the higher the number C_N is, the more complex the multiplication is, i.e. multiplication consumes more time, more area or both. Mullin et al. [MOVW89] proved that the complexity $C_N \geq 2m-1$. Bases reaching $C_N = 2m-1$ are called optimal normal bases. Optimal normal bases belong to the subset of normal bases called Gaussian normal bases. In Gaussian normal bases, the complexity of the base is expressed by the type of a base T. Optimal normal bases are of Type I and Type II. As an illustrative example, let us show a set of equations for $m = 6$ and an optimal normal basis of Type II:

Example 3.2. For $GF(2^6)$ and its optimal normal basis the multiplication matrix $\mathbf{M}$ is

$$\mathbf{M} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} \tag{3.11}$$

Algorithm 3.1 Normal basis multiplication

Input: The multiplication matrix $\mathbf{M}$ for the field $GF(2^m)$;
field elements $a = (a_0 a_1 \dots a_{m-1})$ and $b = (b_0 b_1 \dots b_{m-1})$.
Output: The product $c = (c_0 c_1 \dots c_{m-1})$ of a and b.

1: $x \leftarrow a$
2: $y \leftarrow b$
3: **for** $k = 0$ to $m - 1$ **do**
4: (compute via matrix multiplication)
$c_k \leftarrow x\mathbf{M}y^{tr}$
(where y^{tr} denotes the matrix transpose of the vector y)
5: $x \leftarrow LeftShift(x)$, $y \leftarrow LeftShift(y)$,
(where $LeftShift$ denotes the circular left shift operation)
6: **end for**
7: **return** $c = (c_0 c_1 \dots c_{m-1})$

Let $a = (a_0 a_1 a_2 a_3 a_4 a_5)$ and $b = (b_0 b_1 b_2 b_3 b_4 b_5)$ be two elements of $GF(2^6)$. After application of a multiplication matrix 3.11 into 3.10 we obtain a following set of equations for bits $(c_0 c_1 c_2 c_3 c_4 c_5)$of result $c = a \times b$:

$$
\begin{aligned}
c_0 &= a_0 b_1 + a_1(b_0 + b_4) + a_2(b_3 + b_4) + a_3(b_2 + b_5) + a_4(b_1 + b_2) + a_5(b_3 + b_5) \\
c_1 &= a_1 b_2 + a_2(b_1 + b_5) + a_3(b_4 + b_5) + a_4(b_3 + b_0) + a_5(b_2 + b_3) + a_0(b_4 + b_0) \\
c_2 &= a_2 b_3 + a_3(b_2 + b_0) + a_4(b_5 + b_0) + a_5(b_4 + b_1) + a_0(b_3 + b_4) + a_1(b_5 + b_1) \\
c_3 &= a_3 b_4 + a_4(b_3 + b_1) + a_5(b_0 + b_1) + a_0(b_5 + b_2) + a_1(b_4 + b_5) + a_2(b_0 + b_2) \\
c_4 &= a_4 b_5 + a_5(b_4 + b_2) + a_0(b_1 + b_2) + a_1(b_0 + b_3) + a_2(b_5 + b_0) + a_3(b_1 + b_3) \\
c_5 &= a_5 b_0 + a_0(b_5 + b_3) + a_1(b_2 + b_3) + a_2(b_1 + b_4) + a_3(b_0 + b_1) + a_4(b_2 + b_4)
\end{aligned}
\tag{3.12}
$$

The set of equations is regular. The equation for bit c_{i+k} can be derived from the equation for c_i by a k-bit circular left rotation of arguments a and b. Algorithm 3.1 implements the multiplication of two elements of a field $GF(2^m)$ represented in a normal basis. This algorithm has been taken from [IEE00].

3.4.3 Squaring

Squaring in a normal basis is implemented as a circular shift of an argument "one bit to the right"; if $a = (a_0 a_1 \dots a_{m-2} a_{m-1})$, then $a^2 =$

Algorithm 3.2 Itoh-Teechai-Tsujii inversion in $GF(2^m)$

Input: A field $GF(2^m)$ and a nonzero field element β
Output: The reciprocal β^{-1}

1: Let $m-1 = b_r b_{r-1} \dots b_1 b_0$ be the binary representation of $m-1$, where the most significant bit b_r of $m-1$ is 1.
2: $\eta \leftarrow \beta$, $k \leftarrow 1$
3: **for** $i = r$ downto 1 **do**
4: $\quad \mu \leftarrow \eta$
5: $\quad$ **for** $j = k$ downto 1 **do**
6: $\quad\quad \mu \leftarrow \mu^2$
7: $\quad$ **end for**
8: $\quad \eta \leftarrow \mu\eta$, $k \leftarrow 2k$
9: $\quad$ **if** $b_{i-1} = 1$ **then**
10: $\quad\quad \eta \leftarrow \eta^2\beta$, $k \leftarrow k+1$
11: $\quad$ **end if**
12: **end for**
13: **return** η^2

$(a_{m-1}a_0a_1 \dots a_{m-2})$. As obvious, this operation is in a normal basis very simple — it can be performed in one or zero clock cycles.

3.4.4 Division/Inversion

For division or inversion in a polynomial basis, the extended Euclidean algorithm is used. Unfortunately, this algorithm is not applicable in a normal basis. In a normal basis, division is implemented as multiplication by an inverse element of a divisor.

The fastest known inversion algorithm that can be used in $GF(2^m)$ with a normal basis is the algorithm developed by Itoh, Teechai, and Tsujii [ITT86] outlined in Algorithm 3.2. Note that the algorithm can be generalized for any $GF(p^m)$. The algorithm uses repeated multiplication (steps 8 and 10) and squaring (steps 6, 10 and 13). During the execution of an algorithm, $r = \lfloor \log(m-1) \rfloor$ multiplications are performed in step 8, and $w(m-1)-1$ multiplications are performed in step 10, where $w(\circ)$ denotes the Hamming weight. Total number of multiplications necessary for one inversion is

$$I_M = \lfloor \log(m-1) \rfloor + w(m-1) - 1. \tag{3.13}$$

The number of iterative squarings performed in step 6 is

$$I_{IS} = (m-1) - w(m-1). \tag{3.14}$$

The number of squarings performed in step 10 is $w(m-1)-1$, and there is one last squaring in step 13 [ITT86]. A total number of squarings necessary for one inversion is then

$$I_S = m - 1. \tag{3.15}$$

Chapter 4

Previous work

Multiplication is the crucial operation to be implemented over $GF(2^m)$ with a normal basis. Other operations are in a normal basis either simple or based on multiplication. There has been an array of normal basis multipliers developed. An overview of several architectures of normal basis multipliers can be found e.g. in [ANR99]. Alternative architectures with a slightly reduced gate count or a critical path were reported in [GS00, RMH03, KGKH04]. Some multipliers are optimized for special cases of normal bases, mainly to the optimal normal bases of both Type I [KS98] and Type II [Kwo03, SK01]. Here we deal with Massey-Omura multiplier [MO86] that can be used for any type of a normal basis, and with its pipelined version by Agnew et al. [AMOV91].

4.1 Massively Parallel Multiplier

The set of equations defining the bits of result (see example set 3.12) can be implemented as a combinational logic that computes all bits of a result in parallel. Such a multiplier is huge (the amount of hardware is proportional to m^2, in the best case) and the critical path is long (it is proportional to $\log m$) This multiplier is also called *bit-parallel*.

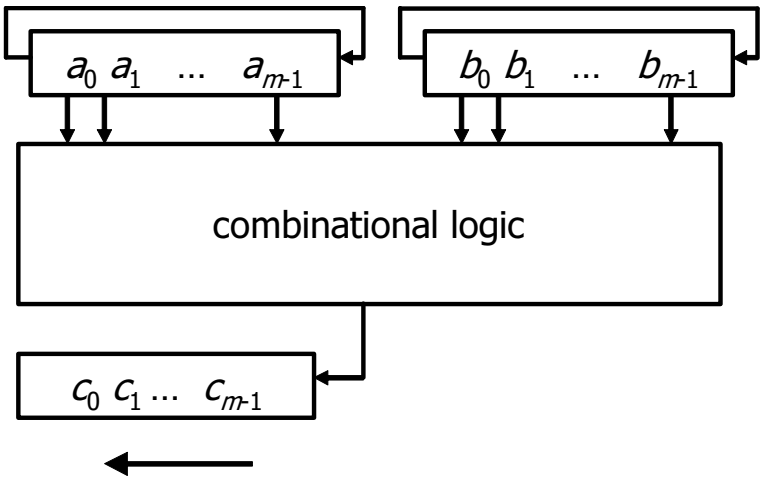

Figure 4.1: Massey-Omura multiplier

4.2 Massey-Omura Multiplier

Massey and Omura [MO86] proposed a multiplier that employs the regularity of equations for all bits of a result. If we construct an equation for one bit of a result (e.g. c_0), equations for other bits can be derived by rotating bits of arguments a and b, as shown in Algorithm 3.1 (see also an example set of Equations 3.12). In this multiplier, one bit of the result is computed in one clock cycle and the registers holding arguments a and b are rotated one bit to the left between cycles. The Massey-Omura multiplier is m times smaller than the massively parallel one because it contains logic for the computation of one bit only. The computation of the result takes m clock cycles. The length of the critical path remains the same as for the massively parallel multiplier. This multiplier is also called *bit-serial*. The block structure of the Massey-Omura multiplier is shown in Figure 4.1.

4.3 Pipelined Massey-Omura Multiplier

Agnew, et al. [AMOV91] modified the Massey-Omura multiplier by pipelining and parallelization. From Equation 3.10 it follows, that the equation for each bit of result can be divided into m terms $T_{i,j}$:

$$c_i = T_{i,0} + T_{i,1} + \cdots + T_{i,m-2} + T_{i,m-1} = \sum_{j=0}^{m-1} T_{i,j}, \tag{4.1}$$

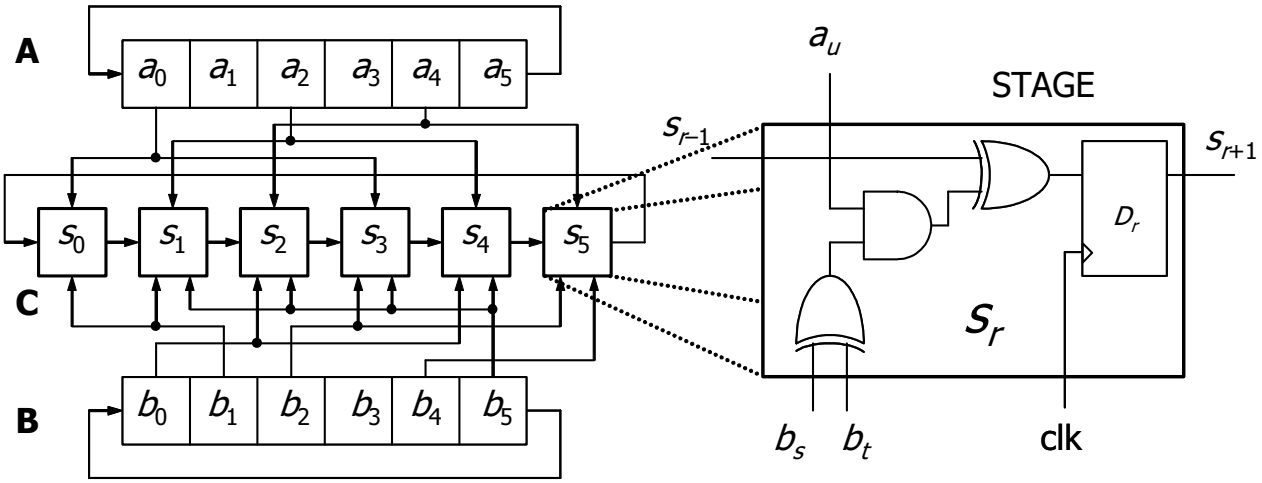

Figure 4.2: Modification of the Massey-Omura multiplier by Agnew et al. Structure shown here is for $GF(2^6)$ and its optimal normal basis

where

$$T_{i,j} = a_{j+i} \sum_{l=0}^{m-1} b_{l+i} \lambda_{jl} \tag{4.2}$$

(the values of the subscript indices are reduced modulo m).

In the multiplier by Agnew et al., the computation of the result is again performed in m clock cycles. In the k-th clock cycle, terms $T_{i,i+k}$ ($\forall i, 0 \leq i \leq m-1$) are evaluated and added to the intermediate results of the corresponding bits c_i. Registers A and B that hold arguments a and b are rotated one bit right in every clock cycle. As pipelining is used, also the register C, in which the result is successively evaluated, is rotated one bit right in every clock cycle. The result in the register C is available after m clock cycles. For illustration, the block structure of a multiplier for the set of Equations 3.12 is shown in Figure 4.2. The initial content of registers A and B is shown in this figure.

The combinational logic, which lies in front of each C register bit, implements one term. Denote the logic together with the register bit a *stage*. The amount of hardware in the multiplier by Agnew et al. is the same as for the Massey-Omura multiplier, but the combinational logic is distributed over the stages $s_0 s_1 \ldots s_{m-1}$ of the register C. Thus the critical path is short and constant (it does not depend on m) and the maximum achievable frequency is higher.

Rule 4.1 (pipelined Massey-Omura multiplier)**.** *Let $q = m$ be the number of clock cycles of one multiplication. Then, in the k-th clock cycle $(0 \leq k \leq q-1)$, the stage s_r $(0 \leq r \leq m-1)$ evaluates the term*

$$S_{r,k} = T_{r-k,r} \tag{4.3}$$

(the values of the subscript indices are reduced modulo m).

After substituting 4.2 into 4.3 we get

$$S_{r,k} = T_{r-k,r} = a_{r+r-k} \sum_{l=0}^{m-1} b_{l+r-k}\lambda_{rl},$$

$$S_{r,k} = a_{2r-k} \sum_{l=0}^{m-1} b_{(l+r)-k}\lambda_{rl} \tag{4.4}$$

(the values of the subscript indices are reduced modulo m).

The term $S_{r,k}$ is added to the partial result of the bit c_{r-k}, which is, due to the rotation of the register C, present in the stage s_r during the k-th clock cycle. The result $c_0c_1c_2 \ldots c_{m-1}$ is available in stages $s_{m-1}s_0s_1 \ldots s_{m-2}$ after m clock cycles.

Let us briefly explain the functionality of the multiplier. From Equations 4.3 and 4.4 follows that in the first clock cycle ($k = 0$), stage s_r evaluates the term (see also Figure4.3)

$$S_{r,0} = T_{r,r}$$

or

$$S_{r,0} = a_{2r-0} \sum_{l=0}^{m-1} b_{(l+r)-0}\lambda_{rl} \tag{4.5}$$

(the values of the subscript indices are reduced modulo m).

From the comparison of Equations 4.5 and 4.4 is obvious that to evaluate an appropriate term $S_{r,k}$ in the k-th clock cycle, only arguments a and b are needed to rotate k bits to the right. Arguments a and b are held in registers A and B.

From Equation 4.1 follows that on its run around the register C, the bit c_i must "collect" all terms $T_{i,j}$ with an equal first index i and with all distinct second indices $j, 0 \leq j \leq m-1$. The equality of the first index i is satisfied by the rotation of the register C — the term $T_{r-k,r}$ is added to the partial result

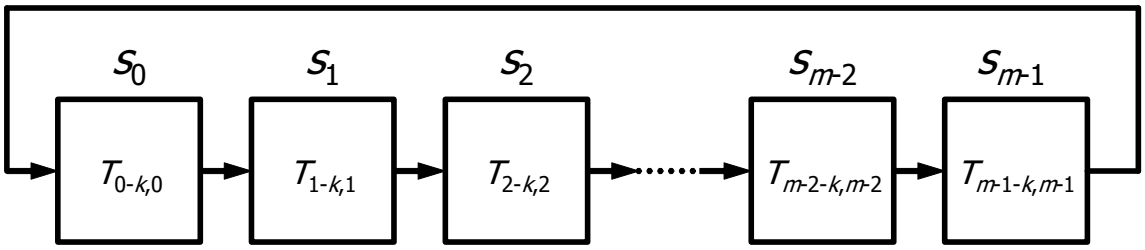

Figure 4.3: Terms evaluated in the stages of the register C in the bit-serial multiplier in the k-th clock cycle

of the bit c_{r-k} in the stage s_r during the k-th clock cycle. The second index j is fixed with an appropriate stage s_j (see 4.3). As every bit c_i of the result passes all stages s_j, it "collects" all terms $T_{i,j}$.

Let us note that the description given in Rule 4.1 corresponds to the one given at [AMOV91]. In this description, in the first clock cycle ($k = 0$), the stage s_r evaluates the term $T_{r,r}$. However, it is possible to make permutation of terms $T_{i,j}$. Generally, we have to only satisfy that the terms $T_{i,j}$ concurrently evaluated in stages $s_0 s_1 \dots s_{m-1}$ during one clock cycle have distinct values of indices i as well as distinct values of indices j.

Example 4.1. It is shown in Figure 4.4 how the product c from Example 3.2 is successively evaluated in a pipelined bit-serial multiplier. In the first clock cycle ($k = 0$) boxed terms are evaluated, in the second clock cycle ($k = 1$) overlined terms are evaluated etc. Multiplication takes $q = m = 6$ clock cycles.

In the following we will sketch the proof of correctness of the pipelined Massey-Omura multiplier. Outline of the proof will be used for proofs of other multipliers. To prove the correctness of any of the multipliers we have to show that the Equation 4.1 is satified for any c_i.

Proof of correctness of the bit-serial multiplier. Let $i = r - k$. From Rule 4.1 it follows that in the k-th clock cycle the partial result of the bit c_i is present in the stage s_{i+k}. The stage s_{i+k} evaluates the term $S_{i+k,k} = T_{i,i+k}$ which is added to the partial result of c_i present in the stage. As evaluation takes $q = m$ clock cycles, then

$$c_i = \sum_{k=0}^{m-1} S_{i+k,k} = \sum_{k=0}^{m-1} T_{i,i+k}$$

$$c_0 = \boxed{a_0 b_1} + \overline{a_1(b_0 + b_4)} + a_2(b_3 + b_4) + a_3(b_2 + b_5) + a_4(b_1 + b_2) + a_5(b_3 + b_5)$$

$$c_1 = a_1 b_2 + \boxed{a_2(b_1 + b_5)} + \overline{a_3(b_4 + b_5)} + a_4(b_3 + b_0) + a_5(b_2 + b_3) + a_0(b_4 + b_0)$$

$$c_2 = a_2 b_3 + a_3(b_2 + b_0) + \boxed{a_4(b_5 + b_0)} + \overline{a_5(b_4 + b_1)} + a_0(b_3 + b_4) + a_1(b_5 + b_1)$$

$$c_3 = a_3 b_4 + a_4(b_3 + b_1) + a_5(b_0 + b_1) + \boxed{a_0(b_5 + b_2)} + \overline{a_1(b_4 + b_5)} + a_2(b_0 + b_2)$$

$$c_4 = a_4 b_5 + a_5(b_4 + b_2) + a_0(b_1 + b_2) + a_1(b_0 + b_3) + \boxed{a_2(b_5 + b_0)} + \overline{a_3(b_1 + b_3)}$$

$$c_5 = \overline{a_5 b_0} + a_0(b_5 + b_3) + a_1(b_2 + b_3) + a_2(b_1 + b_4) + a_3(b_0 + b_1) + \boxed{a_4(b_2 + b_4)}$$

Figure 4.4: Evaluation of terms in pipelined bit-serial multiplier by Agnew et al. (for $GF(2^6)$ and its optimal normal basis)

As the indices are reduced modulo m, the Equation 4.1 is satisfied. □

4.4 Other Multipliers

Reyhani-Masoleh and Hasan [RMH03] proposed two other architectures of the multiplier. By utilizing the symmetric property of the multiplication, their multipliers have reduced area complexity in comparison to the multiplier by Agnew et al. On the other hand, the critical path of these multipliers is slightly longer, or at least comparable to that of the multiplier by Agnew et al.

Kwon et al. [KGKH04] proposed another structure for a pipelined bit-serial multiplier. Their multiplier is applicable only for odd values of m. The multiplier has the area complexity comparable to the multiplier by Reyhani-Masoleh and Hasan, while it preserves the critical path delay of the multiplier by Agnew et al.

The structure of the multiplier is similar to the structure of the multiplier by Agnew et al. Simply saying, by smart reordering the terms $T_{i,j}$, some logical expressions are repeated in several distinct stages. Consequently, the combinational logic implementing those expressions can be reused.

Figures 4.5 shows, how the set of equations is evaluated in multiplier by Agnew et al. Figure 4.6 shows, how the same set of equations is evaluated in multiplier by Kwon et al. Boxed terms are evaluated in the first clock cycle. As evident from the example in Figure 4.6, an expression $(b_2 + b_3)$ appears in

$$c_0 = \boxed{a_0 b_1} + a_1(b_0 + b_3) + a_2(b_3 + b_4) + a_3(b_1 + b_2) + a_4(b_2 + b_4)$$
$$c_1 = a_1 b_2 + \boxed{a_2(b_1 + b_4)} + a_3(b_4 + b_0) + a_4(b_2 + b_3) + a_0(b_3 + b_0)$$
$$c_2 = a_2 b_3 + a_3(b_2 + b_0) + \boxed{a_4(b_0 + b_1)} + a_0(b_3 + b_4) + a_1(b_4 + b_1)$$
$$c_3 = a_3 b_4 + a_4(b_3 + b_1) + a_0(b_1 + b_2) + \boxed{a_1(b_4 + b_0)} + a_2(b_0 + b_2)$$
$$c_4 = a_4 b_0 + a_0(b_4 + b_2) + a_1(b_2 + b_3) + a_2(b_0 + b_1) + \boxed{a_3(b_1 + b_3)}$$

Figure 4.5: Evaluation of terms in pipelined bit-serial multiplier by Agnew et al. (for $GF(2^5)$ and its Type II optimal normal basis)

two terms. Therefore, the XOR gate producing this expression is *shared* by corresponding two stages, which reduces the gate count in multiplier by Kwon et al. The same stands for an expression $(b_0 + b_2)$.

4.5 Digit-Serial Multiplier

When implementing a cryptosystem in a constrained environment such as at smart cards, the designer needs to consider trade-offs between the area and speed. Bit-serial implementations of the normal basis multiplier outlined above require less area, but they are slow, since they need m clock cycles to generate the product of the two field elements. On the other hand, massively-parallel (bit-parallel) versions are fast but require more area. In between the two ends of the architectural spectrum (i.e., fully bit-serial and fully bit-parallel), digit-serial multipliers exist. Such multipliers give a designer the flexibility to make trade-offs between the speed and the area.

The digit-serial Massey-Omura multiplier in parallel evaluates D bits (called a *digit*) of the result during one clock cycle, in contrast to a bit-serial version in which just one bit is evaluated during one clock cycle. Hence, the computation of one product is faster and takes only $q = \lceil \frac{m}{D} \rceil$ clock cycles. For $D = 1$ we get a bit-serial multiplier discussed in paragraph 4.2, while for $D = m$, we get a bit-parallel multiplier discussed in paragraph 4.1.

$$
\begin{aligned}
c_0 &= \boxed{a_0 b_1} + a_3(b_1 + b_2) + a_1(b_0 + b_3) + a_4(b_2 + b_4) + a_2(b_3 + b_4) \\
c_1 &= a_1 b_2 + \boxed{a_4(b_2 + b_3)} + a_2(b_1 + b_4) + a_0(b_3 + b_0) + a_3(b_4 + b_0) \\
c_2 &= a_2 b_3 + a_0(b_3 + b_4) + \boxed{a_3(b_2 + b_0)} + a_1(b_4 + b_1) + a_4(b_0 + b_1) \\
c_3 &= a_3 b_4 + a_1(b_4 + b_0) + a_4(b_3 + b_1) + \boxed{a_2(b_0 + b_2)} + a_0(b_1 + b_2) \\
c_4 &= a_4 b_0 + a_2(b_0 + b_1) + a_0(b_4 + b_2) + a_3(b_1 + b_3) + \boxed{a_1(b_2 + b_3)}
\end{aligned}
$$

Figure 4.6: Evaluation of terms in pipelined bit-serial multiplier by Kwon et al. (for $GF(2^5)$ and its Type II optimal normal basis)

The transformation of the digit-serial Massey-Omura multiplier into the pipelined multiplier by Agnew et al. leads to the evaluation of D terms in every stage of the multiplier during one clock cycle. The construction of the pipelined digit-serial multiplier is possible whenever a digit width D divides the number of bits m. The computation needs then only $q = \frac{m}{D}$ clock cycles. Every bit of the result is successively evaluated in q consecutive stages, e.g. the bit c_0 passes stages $s_0 \dots s_{q-1}$, while the bit c_{m-1} passes stages $s_{m-1}, s_0 \dots s_{q-2}$. The evaluation of the terms in the register C is shown in Figure 4.7.

Rule 4.2 (a standard digit-serial multiplier)**.** *Let $D|m$. Let $q = \frac{m}{D}$ be the number of the clock cycles of one multiplication. Then, in the k-th clock cycle $(0 \leq k \leq q-1)$, the stage s_r $(0 \leq r \leq m-1)$ evaluates the sum of terms*

$$S_{r,k} = \sum_{j=u_r}^{v_r} T_{r-k,j}, \tag{4.6}$$

where

$$
\begin{aligned}
u_r &= rD, \\
v_r &= u_r + D - 1
\end{aligned}
$$

(the values of the subscript indices are reduced modulo m).

The sum of terms $S_{r,k}$ is added to the partial result of the bit c_{r-k}, which is, due to the rotation of the register C, present in the stage s_r dur-

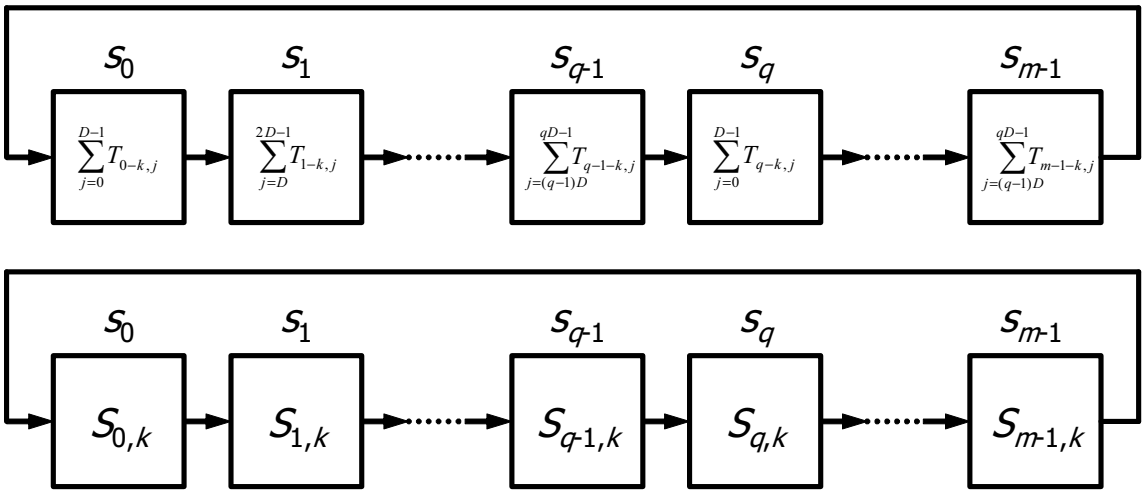

Figure 4.7: Evaluation of terms in the register C in standard digit-serial multiplier, a) full notation, b) abbreviated notation

ing the k-th clock cycle. The result $c_0c_1c_2\ldots c_{m-1}$ *is available in stages* $s_{q-1}s_qs_{q+1}\ldots s_{m-1}s_0\ldots s_{q-2}$ *after q clock cycles.*

To prove the correctness of the digit-serial multiplier we again have to show that the Equation 4.1 is satisfied for any c_i.

Proof of correctness of the digit-serial multiplier. Let $i = r-k$. From Rule 4.2 it follows that in the k-th clock cycle the partial result of the bit c_i is present in the stage s_{i+k}. The stage s_{i+k} evaluates the sum of terms

$$S_{i+k,k} = \sum_{j=u_{i,k}}^{v_{i,k}} T_{i,j},$$

where

$$u_{i,k} = (i+k)D,$$
$$v_{i,k} = u_{i,k} + D - 1$$

The sum of terms $S_{i+k,k}$ is added to the partial result of c_i present in the stage s_{i+k}. As evaluation takes $q = \frac{m}{D}$ clock cycles, then

$$c_i = \sum_{k=0}^{q-1} S_{i+k,k} = \sum_{k=0}^{q-1} \sum_{j=u_{i,k}}^{v_{i,k}} T_{i,j} \tag{4.7}$$

As the indices j form a sequence, concretely $u_{i,k} = v_{i,k-1} + 1$, the equation 4.7 may be rewritten

$$c_i = \sum_{k=0}^{q-1} \sum_{j=u_{i,k}}^{v_{i,k}} T_{i,j} = \sum_{j=u_{i,0}}^{v_{i,q-1}} T_{i,j} = \sum_{j=iD}^{iD+m-1} T_{i,j}$$

As the indices are reduced modulo m, the Equation 4.1 is satisfied. □

Let the block of q consecutive stages be denoted as a *pipeline block*. From Equation 4.1 follows that any pipeline block implements exactly m terms $T_{i,j}$. The whole multiplier implements exactly $D \times m$ terms $T_{i,j}$, hence, we may split the multiplier into D pipeline blocks. From 4.1 also follows that the terms $T_{i,j}$ evaluated in the stages s_i and s_{i+q} have the same second indices j.

In the digit-serial multiplier the amount of combinational logic in the stages of the register C is roughly D-times larger than in the bit-serial one. The amount of other logic and registers remains the same.

Example 4.2. Figure 4.8 shows how the product c from Example 3.2 is successively evaluated in the pipelined digit-serial multiplier. In this case, the digit width $D = 2$, i.e. two terms $T_{i,j}$ are evaluated in each stage in one clock cycle. In the first clock cycle ($k = 0$) boxed terms are evaluated, in the second clock cycle ($k = 1$) overlined terms are evaluated and in the last clock cycle ($k = 2$) unmarked terms are evaluated. Multiplication takes $q = \frac{m}{D} = 3$ clock cycles.

From 4.6 follows that the values of the first indices i of the terms $T_{i,j}$ evaluated in the stage s_r are changing between the cycles, while the values of the second indices j remain constant — they are fixed with an appropriate stage. Figure 4.9 sketches the values of the indices j of the terms $T_{i,j}$ being evaluated in the stages of a pipelined digit-serial multiplier for $D = 2$, as shown in Figure 4.8. Figure 4.10 sketches the values of the indices j for $D = 3$; in this case, multiplication takes $q = \frac{m}{D} = 2$ clock cycles.

The disadvantage of a standard digit-serial multiplier consists in the necessity of dividing the number of bits m by the digit width D. This property may limit scalability options of cryptographic system in which a normal basis multiplier is used, since the set of divisors for chosen m may be small. Moreover, NIST [FIP00] recommends m to be prime for its cryptographic strength. In this case, the construction of a standard digit-serial multiplier described here is impossible.

$$c_0 = \boxed{a_0b_1 + a_1(b_0 + b_4)} + \overline{a_2(b_3 + b_4) + a_3(b_2 + b_5)} + a_4(b_1 + b_2) + a_5(b_3 + b_5)$$

$$c_1 = a_1b_2 + a_2(b_1 + b_5) + \boxed{a_3(b_4 + b_5) + a_4(b_3 + b_0)} + \overline{a_5(b_2 + b_3) + a_0(b_4 + b_0)}$$

$$c_2 = \overline{a_2b_3 + a_3(b_2 + b_0)} + a_4(b_5 + b_0) + a_5(b_4 + b_1) + \boxed{a_0(b_3 + b_4) + a_1(b_5 + b_1)}$$

$$c_3 = \boxed{a_3b_4 + a_4(b_3 + b_1)} + \overline{a_5(b_0 + b_1) + a_0(b_5 + b_2)} + a_1(b_4 + b_5) + a_2(b_0 + b_2)$$

$$c_4 = a_4b_5 + a_5(b_4 + b_2) + \boxed{a_0(b_1 + b_2) + a_1(b_0 + b_3)} + \overline{a_2(b_5 + b_0) + a_3(b_1 + b_3)}$$

$$c_5 = \overline{a_5b_0 + a_0(b_5 + b_3)} + a_1(b_2 + b_3) + a_2(b_1 + b_4) + \boxed{a_3(b_0 + b_1) + a_4(b_2 + b_4)}$$

Figure 4.8: Evaluation of the terms in a pipelined digit-serial multiplier (for $GF(2^6)$ and its optimal normal basis; digit width $D = 2$)

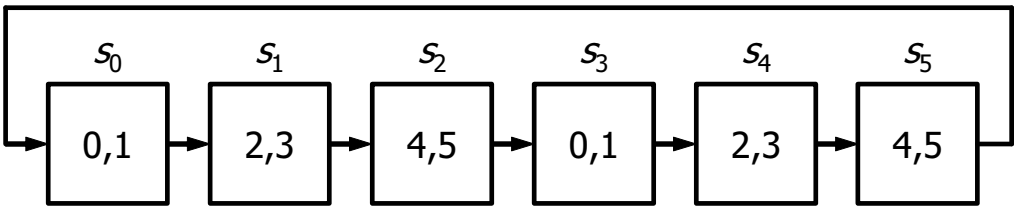

Figure 4.9: Evaluation of the terms in the stages of a digit-serial multiplier for $m = 6$ and $D = 2$. The values of the second indices j of the terms $T_{i,j}$ are introduced. Multiplication takes $q = 3$ clock cycles.

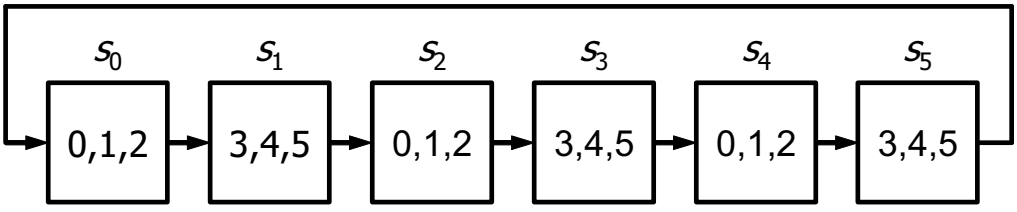

Figure 4.10: Evaluation of the terms in the stages of a digit-serial multiplier for $m = 6$ and $D = 3$. The values of the second indices j of the terms $T_{i,j}$ are introduced. Multiplication takes $q = 2$ clock cycles.

One of our goals was to overcome this disadvantage to allow the designers to scale their designs as much as they need. In Chapter 6, we will introduce four architectures of digit-serial multipliers that can be constructed for any digit width independently of the number of bits m. We have found out that one structure of a general digit-serial multiplier has been probably developed in [LL02]. Unfortunately, the authors neither describe their solution, nor present any reference.

Chapter 5

Multiplication/Inversion Unit

The computation of an inverse element (inversion) by the ITT algorithm [ITT86] may be implemented in a high-level language [BSGEG04], controlled by a microprogram [LMWL00, LL02], or implemented as a hardware macro in a hardware description language [BSGEG04].

As discussed in Subsection 3.4.4, Equations 3.13 through 3.15, the ITT algorithm needs $O(\log m)$ multiplications and $O(m)$ squarings, and thus it takes $O(m \log m)$ clock cycles when a bit-serial multiplier is used. This is a great disadvantage in comparison to the polynomial basis representation, where an inverse element can be computed by an extended Euclidean algorithm and the computation takes $O(m)$ cycles only. This negative effect can be reduced by a digit-serialization of the multiplier [A.1] discussed in Section 4.5. The multiplication takes $\frac{m}{D}$ clock cycles in a digit-serial multiplier and the inversion takes then $O(\frac{m}{D} \log m)$ clock cycles.

In this chapter, we present a modified multiplier by Agnew et al. [AMOV91]. Modifications that we made keep the properties of a multiplier, enabling an efficient implementation of the ITT inversion algorithms. In comparison with other implementations of inversion [LMWL00, LL02, LL03, BSGEG04], no additional registers or data transfers outside the multiplier are necessary. This leads to the savings in the area of a cryptographic processor in which a proposed multiplication/inversion unit is used. Also time necessary for the execution of a cryptographic algorithm may be saved.

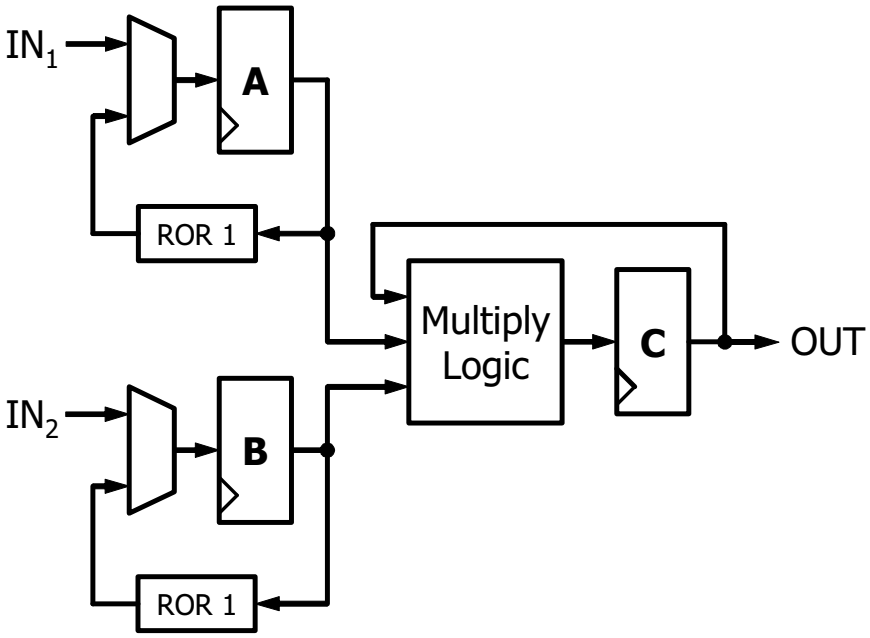

Figure 5.1: Block diagram of the pipelined multiplier by Agnew et al.

Multiplication/inversion unit, presented here, consists of two principal subunits, a multiplier and a shifter. Both these subunits may be scaled, i.e. accelerated in the cost of an increased area. This allows the designer to tune the multiplication/inversion unit for an optimum area, throughput, power, quality factor, or other characteristics.

The proposed design of the multiplication/inversion unit was published in [A.2].

5.1 Structure of the Unit

In Figure 5.1 we present a more precise block scheme of the bit-serial multiplier by Agnew et al., which also considers the input of operands. The multiplication is performed as follows: In the first step, both operands a and b are loaded from inputs IN_1 and IN_2 to the registers A and B, respectively. Then, in m clock cycles, both A and B registers are rotated one bit to the right in each clock cycle (this is represented by the blocks $ROR1$) and the result c in the register C is evaluated successively. After m clock cycles, the result $c = a \times b$ is available at the output OUT. All registers and data paths are m bit wide.

In Figure 5.2 we present our modification of the multiplier by Agnew et al. Modifications include extending the multiplexer preceding the register A

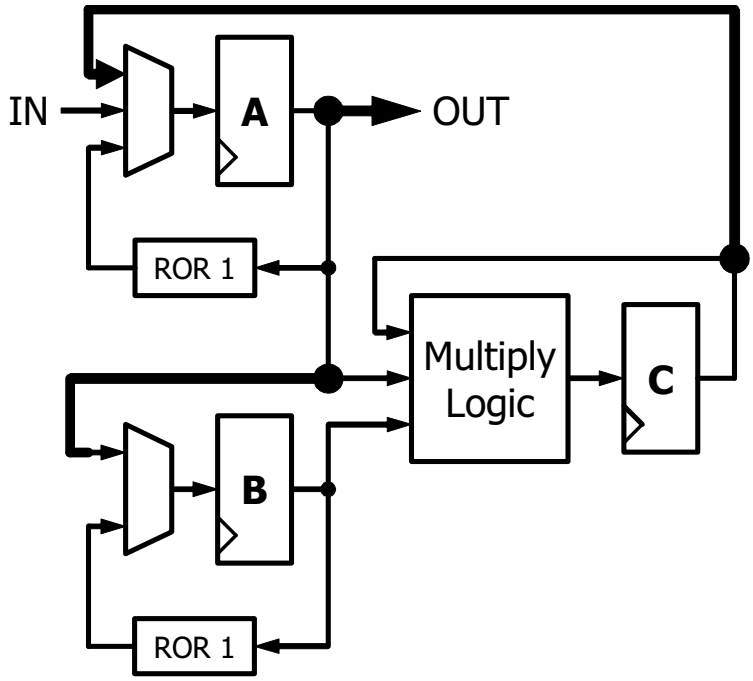

Figure 5.2: Multiplication/inversion unit

from 2:1 to 3:1, and redirecting some data paths (see bold lines). By smart handling of both data processing and data transfers inside the multiplier, we can implement both the multiplication and the ITT inversion in boundaries of this multiplier, and therefore we save additional registers and data transfers outside the multiplier.

The modified multiplier by Agnew et al. has a dedicated control unit (see Figure 5.3) based on a finite state machine and equipped with three counters denoted as $COUNT\_INV$, $COUNT\_MUL$ and $COUNT\_K$ as well as one shift register denoted as M. It implements the commands *load_op* (load operand), *multiply* and *invert*.

5.1.1 Multiplication

Multiplication is performed as follows: In the first two steps, both operands a and b are successively loaded from the input IN to the registers A and B: In the first step, the operand b is loaded from the input IN to the register A. In the second step, the operand b is moved from the register A to the register B and concurrently the operand a is loaded from the input IN to the register A. The multiplication is then performed in m clock cycles. Execution time is measured with a $COUNT\_MUL$ register a in control unit. When

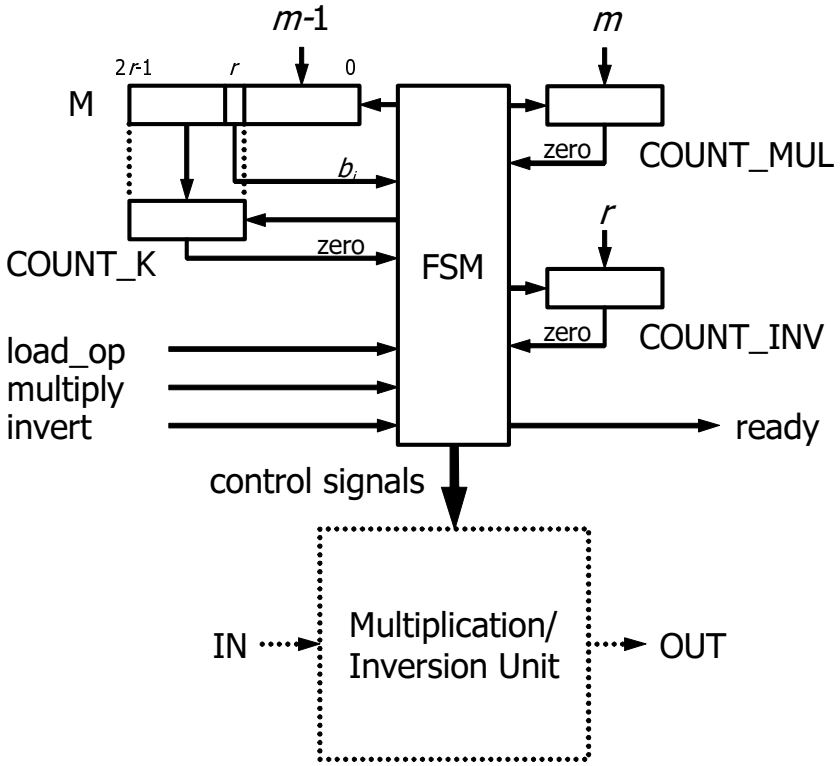

Figure 5.3: Control unit for a multiplication/inversion unit

multiplication is accomplished, the result $c = a \times b$ is loaded from the register C to the register A and is available at the output OUT.

5.1.2 Inversion

The inversion algorithm of Itoh, Teechai and Tsujii [ITT86] has been outlined in Algorithm 3.2. The ITT inversion algorithm is implemented in a multiplication/inversion unit according to Algorithm 5.1. Numbers of steps correspond to those ones in Algorithm 3.2. See also Figures 5.2 and 5.3.

By exploring Algorithm 3.2, the following observation can be made. Let $m-1 = b_r b_{r-1} \dots b_1 b_0$ be the binary representation of $m-1$, where the most significant bit b_r of $m-1$ is 1. Then the value of the variable k in Algorithm 3.2 corresponds to the value represented by bits $b_r \dots b_i$ in the i-th loop of the step 3. Note that the value of the index i is decremented between the loops starting with the value r. This fact is utilized in our algorithm. At the beginning, we store the value $m-1$ into the shift register M. Between the loops of step 3, the shift register is left-shifted in step 8. The value $k = b_r \dots b_i$ is present in bits $M_{2r-1} \dots M_r$ of the register M, from which it is loaded to the counter

Algorithm 5.1 An implementation of the ITT inversion in a multiplication/inversion unit

Input: IN ... value to be inverted;
m ... the degree of the finite field $GF(2^m)$;
r ... the order of the most significant bit in the binary representation of $m-1$, $r = \lfloor \log(m-1) \rfloor$
Output: OUT ... reciprocal IN^{-1}

```
 1: M_r ... M_0 ← m − 1
 2: A ← IN;
 3: for COUNT_INV = r downto 1 do
 4:    B ← A;
 5:    for COUNT_K = M_{2r−1} ... M_r downto 1 do
 6:       B ← B ror 1;
 7:    end for
 8:    C ← B × A
       A ← C
       M ← M shl 1
 9:    if M_r = 1 then
10:       B ← A
          B ← B ror 1; A ← IN
          C ← B × A
          A ← C
11:    end if
12: end for
13: return  A ← A ror 1;
```

$COUNT\_K$ in step 4. The rotation capability of the register B is then used for the computation of iterative squarings in steps 5 and 6.

As the register M is shifted, the bit b_i, tested in step 9, is present in the bit M_r of the register M in each loop of step 3. The counter $COUNT\_INV$ holds the current value of the variable i.

Note that the inverted value must be available at the input IN during the whole process of inversion, as it is repeatedly loaded in step 10. After completion of the inversion algorithm, the result is stored in the register A and is available at the output OUT.

5.1.3 Division

Division $c = \frac{a}{b}$ is implemented as multiplication by an inverse element of the denominator b. In multiplication/inversion unit described here, the first inversion of the denominator b is performed. After the completion of the inversion algorithm the inverse element b^{-1} is stored in the register A. Subsequently, the nominator a is loaded from the input IN to the register A and concurrently the inverse element b^{-1} is moved from the register A to the register B. Then multiplication of a and b^{-1} is performed. When accomplished, the result $c = \frac{a}{b}$ is loaded from the register C to the register A and is available at the output OUT.

5.2 Throughput Improvement of the Unit

As discussed in Subsection 3.4.4, the ITT algorithm needs $I_M = \lfloor \log(m-1) \rfloor + w(m-1) - 1$ multiplications (Equation 3.13) and $I_{IS} = (m-1) - w(m-1)$ iterative squarings in step 6 (Equation 3.14). If one squaring takes 1 clock cycle, then the total number of clock cycles spent for one inversion is

$$C_{INV} = I_M \times q + I_{IS} + const$$
$$C_{INV} = (\lfloor \log(m-1) \rfloor + w(m-1) - 1) \times q + (m-1) - w(m-1) + const,$$

where q is the number of clock cycles necessary for one multiplication and *const* represents the overhead caused by the data transfers and initialization. In case of a bit-serial multiplier, for which $q = m$, the total number of clock cycles spent for one inversion is

$$C_{INV} = (\lfloor \log(m-1) \rfloor + w(m-1) - 1) \times m + (m-1) - w(m-1) + const.$$

As we can see, the number of clock cycles necessary for the inversion is $O(m \log m)$.

The basic ECC operation, namely the computation of a scalar point multiple (Definition 3.7), is performed by repeated point addition and point doubling. If affine point coordinates are used, then, according to Equations 3.3 through 3.8, each of these operations needs 1 division and 1 multiplication, i.e. 1 inversion and 2 multiplications. It also needs 1 squaring and 6–9 additions,

but these operations are performed in one clock cycle. The number of clock cycles necessary for one point addition or doubling is then

$$C_{PADD} = (I_M + 2) \times q + I_{IS} + const.$$

For a bit-serial multiplier we get:

$$C_{PADD} = (\lfloor \log(m-1) \rfloor + w(m-1) - 1 + 2) \times m + (m-1) - w(m-1) + const.$$

We can reduce the number of clock cycles C_{PADD} in two ways:

- by accelerating the multiplications, i.e. by reducing the number q of clock cycles necessary for one multiplication, and
- by reducing the number I_{IS} of clock cycles necessary for the iterative squaring.

For the acceleration of the multiplication, we use a digit-serial multiplier. For the acceleration of the iterative squarings, we use a shifter unit.

5.2.1 Digit-Serialization of the Multiplier

The digit-serial multiplier has been discussed in paragraph 4.5. Its use in a multiplication/inversion unit may dramatically reduce the number of clock cycles.

> **Example 5.1.** Let $m = 180$. Then $m - 1 = 179 = 10110011_2$. The ITT inversion needs $\log 179 + w(179) - 1 = 7 + 5 - 1 = 11$ multiplications. Hence, the point addition needs $11 + 2 = 13$ multiplications. In the case of the bit-serial multiplier, $13 \times 180 = 2340$ clock cycles are necessary for all multiplications during one run of the point addition. In the case of the digit-serial multiplier for a digit $D = 10$, only $13 \times \frac{180}{10} = 234$ clock cycles are necessary.

The number of clock cycles necessary for one point addition or doubling is for the digit-serial multiplier/inverter

$$C_{PADD} = (\lfloor \log(m-1) \rfloor + w(m-1) - 1 + 2) \times \left\lceil \frac{m}{D} \right\rceil + (m-1) - w(m-1) + const.$$

Since more products are evaluated in one clock cycle, more combinational logic is necessary. The area of the block *COMB. LOGIC* in Figure 5.2 is proportional to D. The size of other blocks remains constant.

As the combinational logic is more complex, the length of the critical path τ grows proportionally to $\log D$, $\tau \propto \log D$. Since one multiplication needs $\frac{m}{D}$ clock cycles, the total time necessary for one multiplication is $O(\frac{m}{D} \log D)$ and the total time of one inversion (or point addition on an elliptic curve) is

$$T_{PADD} \approx const \times \left(\left(\frac{m}{D} \log m + m \right) \times \log D \right). \tag{5.1}$$

5.2.2 Modification of the Shifter

Another way to improve the throughput of the ITT inversion is to reduce the number of clock cycles necessary for the iterative squarings in step 5 of Algorithm 5.1.

> **Example 5.2.** Let $m = 180$. Then, the ITT algorithm is performed in 7 cycles (step 3 of the algorithm). Register B is in these 7 cycles right-rotated by 1, 2, 5, 11, 22, 44 and 89 bits (in steps 5). Each of the rotations is performed exactly once in an inversion operation. The total number of clock cycles necessary for execution of step 6 during one ITT inversion is $1 + 2 + 5 + 11 + 22 + 44 + 89 = 174$ clock cycles. Note that this number is comparable to 234 clock cycles necessary for all multiplications when $D = 10$, therefore reducing this number would be beneficial.

Step 5 of Algorithm 5.1 is entered exactly r-times ($r = \lfloor \log(m-1) \rfloor$) during the execution of the ITT algorithm. The register B is by each entry rotated by $k_i = b_r \dots b_i$ bits to the right. We used the rotation capability of the register B, represented by the block $ROR1$ to implement all these rotations. Formally written:

Let m be the degree of the finite field $GF(2^m)$. Let $(b_r b_{r-1} \dots b_1 b_0)$ be the binary representation of $m-1$ such that the most significant bit $b_r = 1$. The set of rotations required by the ITT algorithm is

$$\mathbb{K} = \{k_i, i = r \dots 1; k_i = (b_r \dots b_i)\}.$$

The binary representation of k_i is $b_r \dots b_i$. Each of the rotations k_i is performed exactly once in an inversion operation.

Instead of just one rotation block $ROR1$, we may use the set of r rotation blocks, each one implementing exactly one rotation from the set $\mathbb{K}$. All iterative squarings take then exactly r clock cycles. But, this solution leads to a

Table 5.1: The number of clock cycles required for squarings in ITT algorithm for various number of rotation blocks and for $m = 131$, 180 and 251.

	# rotation blocks				
m	1	2	3	4	5
131	128	23	14	10	9
180	174	24	15	11	9
251	244	34	16	12	10

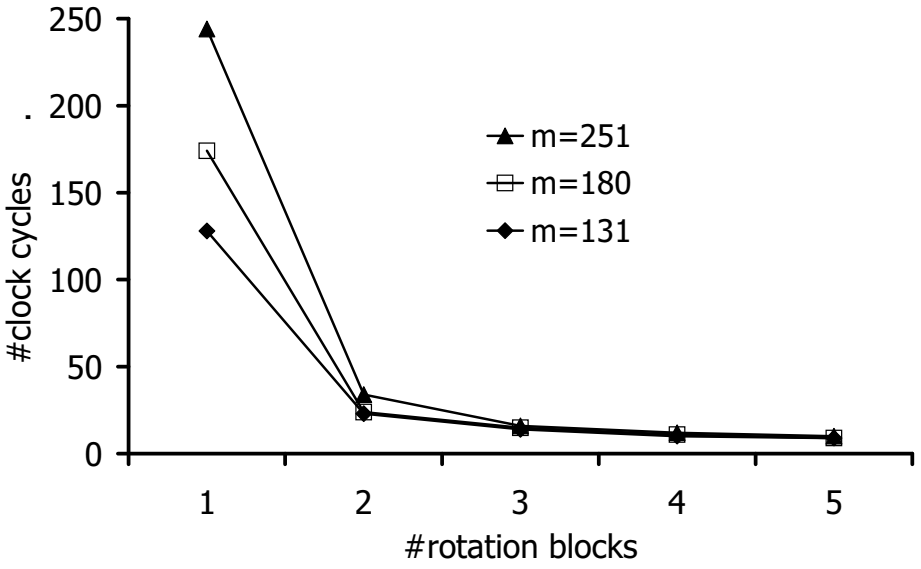

Figure 5.4: The number of clock cycles spent in squarings for various number of rotation blocks.

$(r+1)$-input multiplexer preceding the register B. Our measurements showed that such a multiplexer would be very large — its area would be comparable to the multiplier.

However, it is possible to trade-off between these two extreme approaches. We may implement a limited set $\mathbb{S}$ of rotations and we can compose all rotations from $\mathbb{K}$ using rotations from $\mathbb{S}$. Even with a relatively small set $\mathbb{S}$, clock savings would be significant, as demonstrated in Table 5.1 and Figure 5.4 for $m = 131$, 180 and 251. Blocks implementing "long distance" rotations are shown in Figure 5.5.

Example 5.3. [a shifter with 2 rotation blocks] Let $m = 180$. Then $\mathbb{K} = \{1, 2, 5, 11, 22, 44, 89\}$. If $\mathbb{S} = \{\mathbf{1}, \mathbf{10}\}$ (blocks rotating by **1** bit and

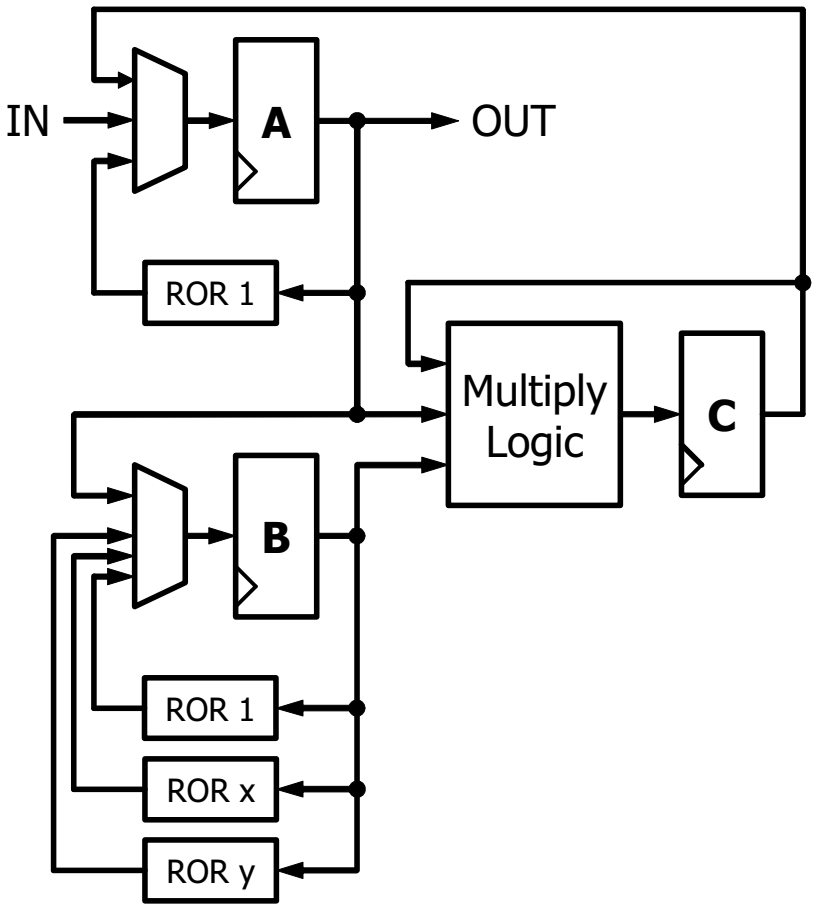

Figure 5.5: "Long distance" rotations $ROR\ x$ and $ROR\ y$ save clock cycles necessary for squarings in ITT algorithm.

10 bits to the right), then all 7 rotations of the set $\mathbb{K}$ will be implemented as follows:

$$\mathbb{K} = \{1\times\mathbf{1}; 2\times\mathbf{1}; 5\times\mathbf{1}; 1\times\mathbf{10}+1\times\mathbf{1}; 2\times\mathbf{10}+2\times\mathbf{1}; 4\times\mathbf{10}+4\times\mathbf{1}; 8\times\mathbf{10}+9\times\mathbf{1}\}.$$

Hence, they will need $I_{IS} = 1+2+5+(1+1)+(2+2)+(4+4)+(8+9) = 39$ clock cycles.

If $\mathbb{S} = \{\mathbf{1}, \mathbf{11}\}$ (blocks rotating by **1** bit and **11** bits to the right), then all 7 rotations of the set $\mathbb{K}$ will be implemented as follows:

$$\mathbb{K} = \{1 \times \mathbf{1}; 2 \times \mathbf{1}; 5 \times \mathbf{1}; 1 \times \mathbf{11}; 2 \times \mathbf{11}; 4 \times \mathbf{11}; 8 \times \mathbf{11} + 1 \times \mathbf{1}\}.$$

Hence, they will need $I_{IS} = 1 + 2 + 5 + 1 + 2 + 4 + (8 + 1) = 24$ clock cycles.

Example 5.4. [a shifter with 3 rotation blocks] Let $m = 180$. Then $\mathbb{K} = \{1, 2, 5, 11, 22, 44, 89\}$. If $\mathbb{S} = \{\mathbf{1}, \mathbf{5}, \mathbf{20}\}$, then all 7 rotations of the set $\mathbb{K}$ will be implemented as follows:

$$\mathbb{K} = \{1\times\mathbf{1}; 2\times\mathbf{1}; 1\times\mathbf{5}; 2\times\mathbf{5}+1\times\mathbf{1}; 1\times\mathbf{20}+2\times\mathbf{1}; 2\times\mathbf{20}+4\times\mathbf{1}; 4\times\mathbf{20}+1\times\mathbf{5}+4\times\mathbf{1}\}.$$

Hence, they will need $I_{IS} = 1+2+1+(2+1)+(1+2)+(2+4)+(4+1+4) = 25$ clock cycles.

If $\mathbb{S} = \{\mathbf{1}, \mathbf{5}, \mathbf{22}\}$, then all 7 rotations of the set $\mathbb{K}$ will be implemented as follows:

$$\mathbb{K} = \{1 \times \mathbf{1}; 2 \times \mathbf{1}; 1 \times \mathbf{5}; 2 \times \mathbf{5} + 1 \times \mathbf{1}; 1 \times \mathbf{22}; 2 \times \mathbf{22}; 4 \times \mathbf{22} + 1 \times \mathbf{1}\}.$$

Hence, they will need $I_{IS} = 1 + 2 + 1 + (2 + 1) + 1 + 2 + (4 + 1) = 15$ clock cycles.

Previous examples show that the number of clock cycles I_{IS} depends both on the choice of elements in $\mathbb{S}$ and their number $|\mathbb{S}|$. In Chapter 7 we discuss in detail how to choose the set $\mathbb{S}$. It shows that for the minimum number of clock cycles it holds that $\mathbb{S} \subseteq \mathbb{K}$.

When an optimal $\mathbb{S}$ is chosen, then the number of clock cycles spent for the iterative squarings may be approximated as $I_{IS} \approx n\sqrt[n]{m-1}$, where n is the number of rotation blocks, $n = |\mathbb{S}|$. If the multiplexer in front of the register B does not lie on a critical path, then the total time necessary for the point addition is

$$T_{PADD} \approx const \times \left(\left(\frac{m}{D}\log m + n\sqrt[n]{m-1}\right) \times \log D\right). \tag{5.2}$$

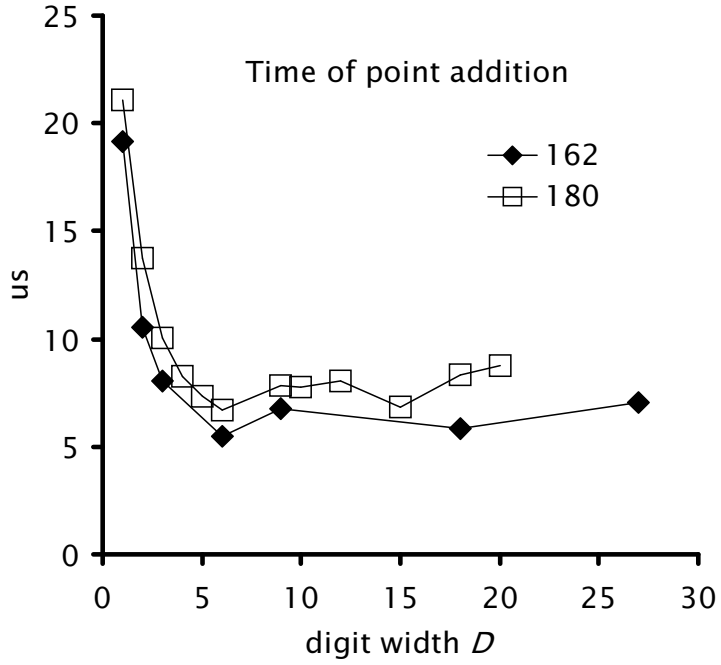

Figure 5.6: Time of point addition for different digit widths. No "long distance" rotation blocks are used.

5.3 Implementation Results

5.3.1 Effect of a Digit-Serialization of the Multiplier

The proposed multiplier/inverter has been implemented in the Xilinx Virtex300 FPGA using the Synopsys FPGA Express synthesis tool and the Foundation 3.3i implementation tool. Its functionality has been verified in the ModelSim simulator.

Table 5.2 shows implementation results for $m = 162$ and $m = 180$ and for a set of several digit widths D. No additional blocks performing "long distance" rotations have been used in these cases. As expected, the area of the multiplier/inverter grows with growing D and can be expressed as approx. $(2 + 0.5D) \times m$ slices or $(2 + 0.5D)$ slices per 1 bit.

The computation time does not depend on the digit width D in such a straightforward manner. The results in the last column of Table 5.2 and in Figure 5.6 correspond to Equation 5.1. Minimum time is obtained for $D = 6$. Other local minima are caused by the granularity of the FPGA. Whenever the capacity of a look-up table is exhausted, the length of the critical path increases.

Table 5.2: Implementation of the modified multiplier/inverter in the Xilinx Virtex300

m	D	Freq [MHz]	#Slices	#Slices per 1 bit	Point addition [clocks]	[μs]
162	1	104.504	406	2.51	2000	19.14
162	2	105.108	489	3.02	1109	10.55
162	3	101.266	569	3.51	812	8.02
162	6	93.906	814	5.02	515	5.48
162	9	61.244	1057	6.52	416	6.79
162	18	53.999	1836	11.33	317	5.87
162	27	40.193	2682	16.56	284	7.07
180	1	122.489	451	2.51	2582	21.08
180	2	102.533	544	3.02	1412	13.77
180	3	101.502	632	3.51	1022	10.07
180	4	100.492	724	4.02	827	8.23
180	5	97.069	815	4.53	710	7.31
180	6	94.153	903	5.02	632	6.71
180	9	64.008	1174	6.52	502	7.84
180	10	61.054	1265	7.03	476	7.80
180	12	54.174	1480	8.22	437	8.07
180	15	58.042	1798	9.99	398	6.86
180	18	44.607	2026	11.26	372	8.34
180	20	40.955	2248	12.49	359	8.77

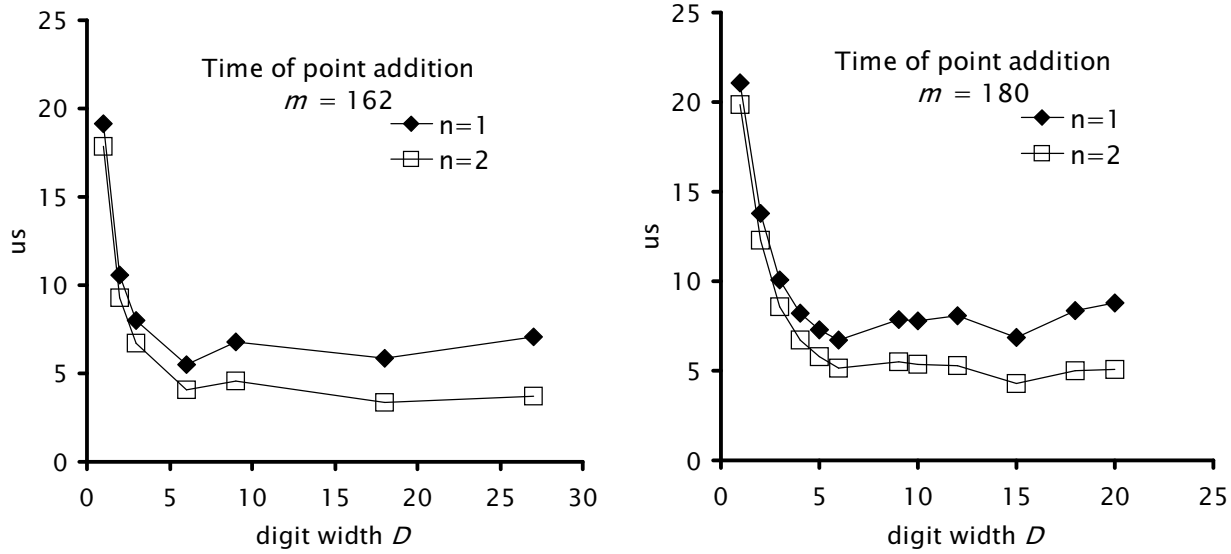

Figure 5.7: The effect of adding 1 rotation block, a) $m = 162$, b) $m = 180$.

5.3.2 Iterative Squarings Improvement in the Shifter

The blocks $ROR\ 1$, $ROR\ q$ and $ROR\ r$ in Figure 5.5 are permutations of wires. A shifter with $n = |\mathbb{S}|$ rotation blocks is implemented just by a $(n+1)$–input multiplexer. The effect of adding one rotation block (i.e. $n = 2$) is illustrated in Figures 5.7a and 5.7b. The expansion of multiplexers did not influence the clock period, as they did not lie on the critical path.

This new design is systematically faster; for $D = 6$, the speedup is over 20%, while the area increased by 10%. Note that the speedup is based on the minimization of the last term in Equation 5.2. In our case, this mechanism caused the second local minimum on the original curve to prevail, where the optimum digit width is $D = 18$ or 15 for $m = 162$ or 180, respectively. In both cases the actual speedup is 37%.

5.4 Summary and Final Remarks

In this chapter we presented a normal basis multiplication/inversion unit. This unit is based on the multiplier by Agnew et al. Hardware complexity has increased slightly in comparison to the multiplier by Agnew et al. — one input has been added to the multiplexer preceding the register A. We also redirected some data paths. By a smart control of the execution and data

transfers we are able to implement both multiplication and inversion, as well as division in a normal basis.

The unit consists of two principal parts, the multiplier and the shifter. Both these parts may be scaled. The multiplier may be scaled by a digit-serialization, the shifter may be scaled by the number of rotation blocks.

A standard digit-serial normal basis multiplier may be created only for such digit widths D that divide the number of bits m. Hence, in the examples we considered only such number of bits m to scale the multiplier well. But, this fact limits the designer options, particularly since m should be prime for its increased cryptographic strength. For m being prime the standard digit-serial multiplier cannot be created at all. Therefore, we developed several architectures of digit-serial multipliers that can be scaled by any digit width D, independently on the number of bits m. These architectures are presented in the following Chapter 6

In Chapter 7 we will discuss the way how to implement optimum shifter in the context of a digit-serialization of a multiplier. We will discuss the number of rotation blocks to use and what size of rotations to implement in these rotation blocks.

Chapter 6

Digit-Serial Multipliers of a General Digit Width

As stated before, m should be a prime number for cryptographic purposes. Therefore, a digit-serial multiplier for such a digit width D that does not divide m is necessary. The fact that D generally does not divide m brings irregularity in the structure of the multiplier. We developed four architectures of digit-serial normal basis multipliers that can be scaled by *any* digit width. These architectures were denoted as the *circular multiplier*, *linear multiplier*, *end-correction multiplier* and the *circular multiplier with distributed overlap*. In this chapter, we describe these architectures, we compare them in terms of area and throughput, and discuss their suitability for different digit widths. In the abbreviations, G means general digit width, C circular, L linear, and other characters distinguish between similar multipliers. Note that the usage of multipliers is not limited to the arithmetic unit described in the previous chapter. These multipliers can be used in any other system working with a normal basis representation of the field elements. In the following text, the notation introduced in Chapter 4 is used.

Part of the research described in this chapter was published in [A.6, A.7, A.8, A.9, A.10].

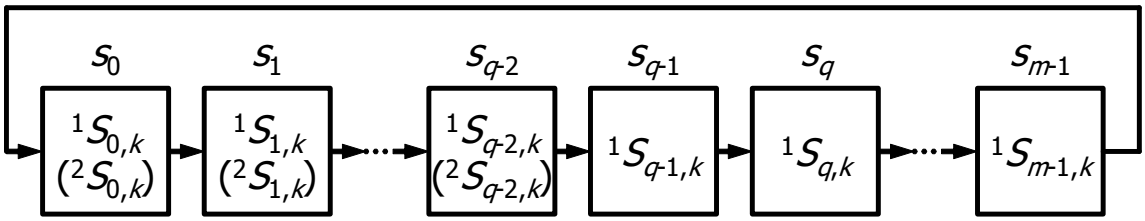

Figure 6.1: **(Circular Multiplier, GC)** Evaluation of the terms in the stages of the register C in a circular digit-serial multiplier.

6.1 Circular Multiplier (GC)

In Figure 6.1 we describe a structure of a circular multiplier, concretely how the terms are evaluated in the stages of the register C. To get a correct result, stages $s_0 \dots s_{q-2}$ must be able to evaluate two different groups of terms. When holding partial results of any of the bits $c_0 \dots c_{q-2}$, they must evaluate the first group (symbolized as ${}^1S$), but when holding partial results of any of bits $c_{m-q+1} \dots c_{m-1}$, they must evaluate the second group (${}^2S$). The stages are step-by-step switched from the group ${}^1S$ to the group ${}^2S$ during the computation in step with the partial results of the bits $c_{m-q+1} \dots c_{m-1}$ successively moving to the stages $s_0 \dots s_{q-2}$. In the k-th clock cycle, stages $s_k \dots s_{q-2}$ evaluate groups ${}^1S$, while stages $s_0 \dots s_{k-1}$ are switched to evaluate groups ${}^2S$. Stages $s_{q-1} \dots s_{m-1}$ do not switch and evaluate groups ${}^1S$ during the whole computation.

A shift register can be used to control a successive switching of the groups. The shift register is initially cleared; during the computation series of '1's is step-by-step shifted in it.

The evaluation of the result takes $q = \lceil \frac{m}{D} \rceil$ clock cycles.

Rule 6.1 (a circular digit-serial multiplier)**.** *Let $q = \lceil \frac{m}{D} \rceil$ be the number of clock cycles of one multiplication. Then, in the k-th clock cycle $(0 \le k \le q-1)$, the stage s_r $(0 \le r \le m-1)$ evaluates the sum of terms $S_{r,k}$: if $r \ge k$:*

$$S_{r,k} = {}^1S_{r,k} = \sum_{j=u_r}^{v_r} T_{r-k,j}, \tag{6.1}$$

where

$$u_r = rD \mod qD,$$
$$v_r = \min\{u_r + D - 1; m - 1\}$$

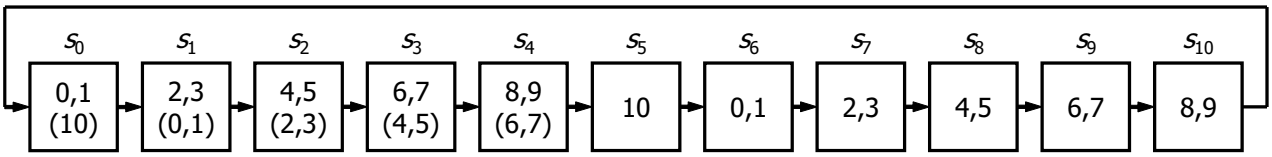

Figure 6.2: Evaluation of the terms in the stages of the circular digit-serial multiplier for $m = 11$ and $D = 2$. Values of the second indices j of the terms $T_{i,j}$ are introduced. Multiplication takes $q = 6$ clock cycles.

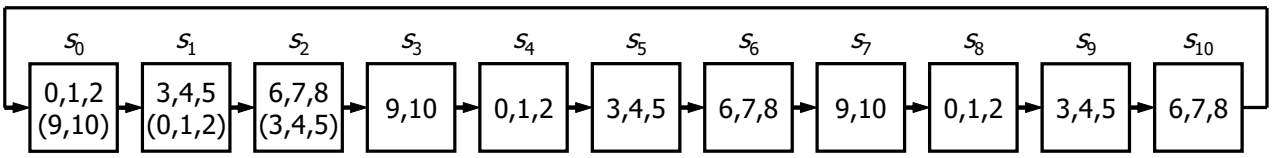

Figure 6.3: Evaluation of the terms in the stages of the circular digit-serial multiplier for $m = 11$ and $D = 3$. Values of the second indices j of the terms $T_{i,j}$ are introduced. Multiplication takes $q = 4$ clock cycles.

if $r < k$*:*

$$S_{r,k} = {}^2S_{r,k} = \sum_{j=u_r}^{v_r} T_{r-k,j}, \tag{6.2}$$

where

$$u_r = (r+m)D \mod qD,$$
$$v_r = \min\{u_r + D - 1; m - 1\}$$

(the values of the subscript indices are reduced modulo m*).*

The sum of terms $S_{r,k}$ *is added to the partial result of the bit* c_{r-k}*, which is, due to the rotation of the register* C*, present in the stage* s_r *during the* k*-th clock cycle. The result* $c_0c_1c_2\ldots c_{m-1}$ *is available in stages* $s_{q-1}s_qs_{q+1}\ldots s_{m-1}s_0\ldots s_{q-2}$ *after* q *clock cycles.*

Example 6.1. Let $m = 11$. It is shown in Figures 6.2 and 6.3 how the terms are evaluated in the stages of the register C in a circular digit-serial multiplier. Indices in parentheses belong to sets of terms ${}^2S$.

The circular multiplier drawback lies in the fact that $q - 1$ pipeline stages must evaluate two different sets of logic. In other words, the circular multiplier contains almost $D + 1$ pipeline blocks in contrast to D pipeline blocks at a

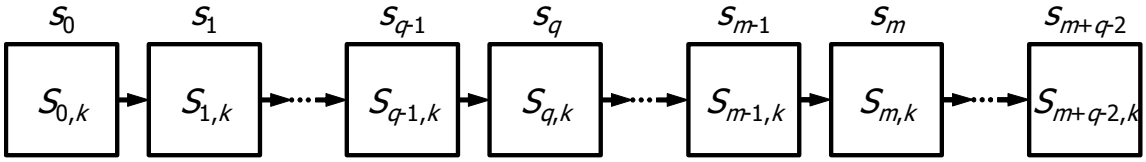

Figure 6.4: **(Linear Multiplier, GL)** Evaluation of the terms in the stages of the register C in a linear digit-serial multiplier. There is no loop in the linear multiplier.

standard multiplier (see Section 4.5 for the definition of a *pipeline block*). The pipeline blocks overlap in $q-1$ stages (hence we also call it the *circular multiplier with a concentrated overlap*). The multiplier must implement almost $(D+1) \times m$ terms $T_{i,j}$. Moreover, we need $q-1$ multiplexers that also consume a relatively large amount of area and may lie on the critical path. Also, the necessity of a successive switching of the groups may slightly complicate the control.

6.2 Linear Multiplier (GL)

Multiplexers, used for switching the groups in the circular multiplier, usually lie on the critical path and cause the clock period to increase. Another disadvantage of this multiplier is the necessity of switching the multiplexers successively (not at once). This led us to the development of another architecture of the multiplier.

Stages $s_0 \dots s_{q-2}$ are successively switched from the groups ${}^1S$ to groups ${}^2S$ in the circular multiplier. Instead of switching, we may evaluate groups ${}^2S$ in additional $q-1$ stages labeled $s_m \dots s_{m+q-2}$. The bit c_i of the result is successively evaluated in stages $s_i \dots s_{i+q-1}$, e.g. the last bit c_{m-1} is evaluated in stages $s_{m-1} \dots s_{m+q-2}$. Every set of q consecutive stages evaluates the whole set of m terms in Equation 4.1. The resulting linear structure of the multiplier is described in Figure 6.4. The evaluation of the result again takes $q = \lceil \frac{m}{D} \rceil$ clock cycles.

Before we propose the description of the linear multiplier, we will discuss the groups ${}^2S$ now evaluated in stages $s_m \dots s_{m+q-2}$ of the linear multiplier. These groups of terms were evaluated in stages $s_0 \dots s_{q-2}$ in the circular multiplier. This implies that $S_{r,k} = {}^2S_{r-m,k} (\forall r, m \leq r \leq m+q-2)$. Equations 6.1 and 6.2 imply that ${}^2S_{r-m,k} = {}^1S_{r,k}$. This fact leads to a simplified description of the linear multiplier.

Rule 6.2 (a linear digit-serial multiplier)**.** *Let* $q = \left\lceil \frac{m}{D} \right\rceil$ *be the number of clock cycles of one multiplication. Then, in the* k*-th clock cycle* $(0 \leq k \leq q-1)$ *the stage* s_r $(0 \leq r \leq m+q-2)$ *evaluates the sum of terms* $S_{r,k}$*:*

$$S_{r,k} = {}^{1}S_{r,k} = \sum_{j=u_r}^{v_r} T_{r-k,j}, \tag{6.3}$$

where

$$\begin{aligned} u_r &= rD \mod qD, \\ v_r &= \min\{u_r + D - 1; m-1\} \end{aligned} \tag{6.4}$$

(the values of the subscript indices of T *are reduced modulo* m*).*

The sum of terms $S_{r,k}$ *is added to the partial result of the bit* c_{r-k}*, which is, due to the* shifting *of the register* C*, present in the stage* s_r *during the* k*-th clock cycle. The result* $c_0c_1c_2\ldots c_{m-1}$ *is available in stages* $s_{q-1}s_qs_{q+1}\ldots s_{m-1}s_m\ldots s_{m+q-2}$ *after* q *clock cycles.*

We do not provide the proof of correctness of the circular multiplier for its relativity to the linear one, which was discussed above. I.e., if the linear multiplier is correct, then also the circular one must be correct. As before, to prove the correctness of the multiplier we have to show that the Equation 4.1 is satisfied for any c_i.

Proof of correctness of the linear multiplier. Let $i = r - k$. From Rule 6.2 it follows that in the k-th clock cycle the partial result of the bit c_i is present in the stage s_{i+k}. The stage s_{i+k} evaluates the sum of terms

$$S_{i+k,k} = \sum_{j=u_{i,k}}^{v_{i,k}} T_{i,j},$$

where

$$\begin{aligned} u_{i,k} &= (i+k)D \mod qD, \\ v_{i,k} &= \min\{u_{i,k} + D - 1; m-1\} \end{aligned}$$

The sum of terms $S_{i+k,k}$ is added to the partial result of c_i present in the stage s_{i+k}. As evaluation takes $q = \lceil \frac{m}{D} \rceil$ clock cycles, then

$$c_i = \sum_{k=0}^{q-1} S_{i+k,k} = \sum_{k=0}^{q-1} \sum_{j=u_{i,k}}^{v_{i,k}} T_{i,j} \tag{6.5}$$

As obvious from Equation 6.5, all terms $T_{i,j}$ have identical first index i.

We have to prove that every index j, $0 \leq j \leq m-1$, appears in Equation 6.5 exactly once. Every bit c_i is successively evaluated in q consecutive stages $s_i, \ldots, s_{i+q-1}$. From Equation 6.4 we get

$$u_{r+q} = (r+q)D \mod qD = rD \mod qD = u_r$$

and also

$$v_{r+q} = \min\{u_{r+q} + D - 1; m-1\} = \min\{u_r + D - 1; m-1\} = v_r$$

As $u_{r+q} = u_r$ and $v_{r+q} = v_r$, the stages s_r and s_{r+q} evaluate terms $T_{i,j}$ with identical sets of second indices j. Therefore, it is sufficient to inspect any set of q consecutive stages.

We choose stages $s_0, \ldots, s_{q-1}$ (the bit c_0 is evaluated here). For $0 \leq r \leq q-1$ we get $u_r = rD$. For $0 \leq r \leq q-2$ we get $v_r = (r+1)D - 1 = u_{r+1} - 1$, i.e., in stages $s_0 \ldots s_{q-1}$ the indices j create a sequence of consecutive integers. The lowest value of j is $u_0 = 0$, the highest value of j is $v_{q-1} = \min\{u_{q-1} + D - 1; m-1\} = \min\{qD - 1; m-1\} = m-1$. □

When comparing the circular and linear multipliers, we find out that the amount of combinational logic which evaluates terms is the same in both cases. Also the number of flip-flops in the registers A and B is the same. The register C needs $q-1$ more flip-flops in the linear multiplier; on the other hand, the shift register used to control multiplexers in the circular multiplier consists of $q-1$ flip-flops as well. The disadvantage of the circular multiplier lies in the necessity of application of the multiplexers — the multiplexers consume additional combinational logic and they may increase the critical path length. It leads us to state that the linear multiplier is more advantageous in comparison with the circular multiplier. Also the description of the multiplier is simpler, as we can see from Rule 6.2.

Note that definition of the linear multiplier given in Rule 6.2 is just one of possible definitions. The terms may also be distributed over the stages

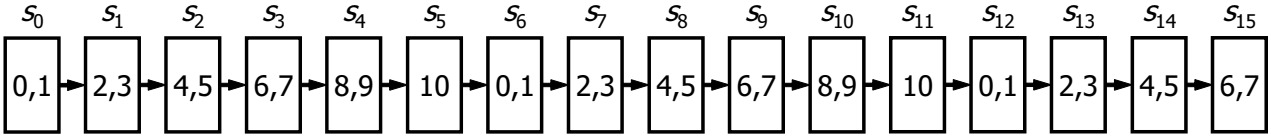

Figure 6.5: Evaluation of the terms in the stages of a linear digit-serial multiplier for $m = 11$ and $D = 2$. Values of the second indices j of the terms $T_{i,j}$ are introduced. Multiplication takes $q = 6$ clock cycles.

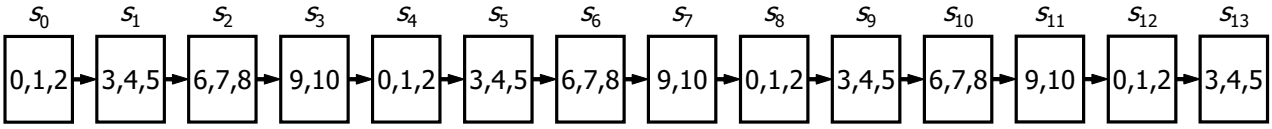

Figure 6.6: Evaluation of the terms in the stages of a linear digit-serial multiplier for $m = 11$ and $D = 3$. Values of the second indices j of the terms $T_{i,j}$ are introduced. Multiplication takes $q = 4$ clock cycles.

evenly, i.e. from any subset of q consecutive stages, exactly $(qD - m)$ of them would evaluate $(D - 1)$ terms, while the remaining $q - (qD - m)$ stages would evaluate D terms. Also, permutations of the terms are possible. *Generally*, when implementing the linear multiplier, two conditions must be satisfied:

1. In the k-th clock cycle, the stage s_r will evaluate terms $T_{i,j}$ with first index $i = r - k$. This is satisfied by the rotation of the registers A and B.
2. Let Q be any subset of q consecutive stages. Then
   (a) The number of terms evaluated in Q must be m.
   (b) For each j, $0 \leq j \leq m - 1$, there will be the term $T_{i,j}$ in Q.

Therefore the stages s_r and s_{r+q} evaluate terms with equal sets of the second indices j.

Example 6.2. Let $m = 11$. It is shown in Figures 6.5 and 6.6 how the terms are evaluated in the stages of the register C in a linear digit-serial multiplier.

6.3 End-Correction Multiplier (GCEC)

In the end-correction multiplier calculation of the result takes again $q = \lceil \frac{m}{D} \rceil$ clock cycles. Every stage evaluates a sum of exactly D terms in the first $q-1$ clock cycles. In the last clock cycle, $qD - m$ terms are switched off and the stage evaluates the remaining $m - (q-1)D$ terms only.

Rule 6.3 (an end-correction multiplier)**.** *Let $q = \lceil \frac{m}{D} \rceil$ be the number of clock cycles of one multiplication. Then, in the k-th clock cycle $(0 \leq k \leq q-2)$, the stage s_r $(0 \leq r \leq m-1)$ evaluates the sum of terms*

$$ {}^1S_{r,k} = \sum_{j=u_r}^{v_r} T_{r-k,j}, \tag{6.6} $$

where

$$ \begin{aligned} u_r &= rD, \\ v_r &= u_r + D - 1 \end{aligned} $$

In the last clock cycle $(k = q-1)$, the stage s_r $(0 \leq r \leq m-1)$ evaluates the following subset of ${}^1S_{r,k}$

$$ {}^2S_{r,k} = \sum_{j=u_r}^{v_r} T_{r-k,j}, \tag{6.7} $$

where

$$ \begin{aligned} u_r &= rD \mod m, \\ v_r &= u_r + (m - (q-1)D) - 1 \end{aligned} $$

(the values of the subscript indices are reduced modulo m).

The sum ${}^1S_{r,k}$ (or the sum ${}^2S_{r,k}$) is added to the partial result of the bit c_{r-k}, which is, due to the rotation of the register C, present in the stage s_r during the k-th clock cycle. The result $c_0c_1c_2 \ldots c_{m-1}$ is available in stages $s_{q-1}s_qs_{q+1} \ldots s_{m-1}s_0 \ldots s_{q-2}$ after q clock cycles.

Proof of correctness of the end-correction multiplier. Let $i = r - k$. From Rule 6.3 it follows that in the k-th clock cycle the partial result of the bit c_i is present in the stage s_{i+k}. The stage s_{i+k} evaluates the sum of terms

$$^1S_{i+k,k} = \sum_{j=u_{i,k}}^{v_{i,k}} T_{i,j},$$

where

$$\begin{aligned} u_{i,k} &= (i+k)D, \\ v_{i,k} &= u_{i,k} + D - 1 \end{aligned}$$

for $0 \leq k \leq q-2$. In the last clock cycle ($k = q-1$) the stage s_{i+k} evaluates the sum of terms

$$^2S_{i+q-1,q-1} = \sum_{j=u_{i,q-1}}^{v_{i,q-1}} T_{i,j},$$

where

$$\begin{aligned} u_{i,q-1} &= (i+q-1)D, \\ v_{i,q-1} &= u_{i,q-1} + (m-(q-1)D) - 1 = iD + m - 1 \end{aligned}$$

First, we have to show that $v_{i,q-1} \geq u_{i,q-1}$. We have $q = \left\lceil \frac{m}{D} \right\rceil < \frac{m}{D} + 1$. Then

$$\begin{aligned} v_{i,q-1} &= u_{i,q-1} + (m-(q-1)D) - 1 = u_{i,q-1} + m - qD + D - 1 \\ &= u_{i,q-1} + m - \left\lceil \frac{m}{D} \right\rceil D + D - 1 > u_{i,q-1} + m - (\frac{m}{D} + 1)D + D - 1 \\ &= u_{i,q-1} + m - m - D + D - 1 = u_{i,q-1} - 1 \end{aligned}$$

We have got $v_{i,q-1} > u_{i,q-1} - 1$. As both $v_{i,q-1}$ and $u_{i,q-1}$ are integers, then also $v_{i,q-1} \geq u_{i,q-1}$.

Second, we have to show that the Equation 4.1 is satisfied. The sum of terms $^1S_{i+k,k}$ or $^2S_{i+q-1,q-1}$ is added to the partial result of c_i present in the stage s_{i+k}. As evaluation takes $q = \left\lceil \frac{m}{D} \right\rceil$ clock cycles, then

$$c_i = \sum_{k=0}^{q-2} {}^1S_{i+k,k} + {}^2S_{i+q-1,q-1} = \sum_{k=0}^{q-2} \sum_{j=u_{i,k}}^{v_{i,k}} T_{i,j} + \sum_{j=u_{i,q-1}}^{v_{i,q-1}} T_{i,j} \tag{6.8}$$

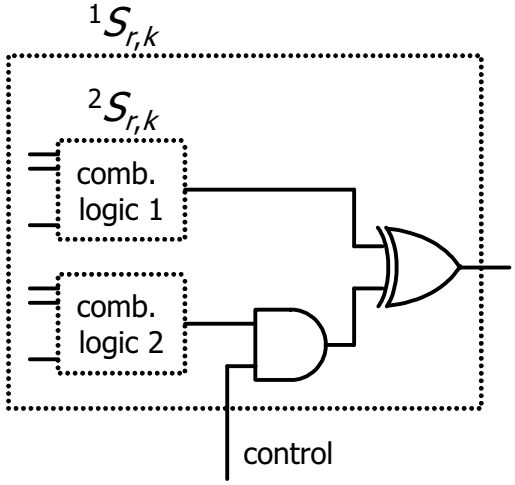

Figure 6.7: The sum $^2S_{r,k}$ contains the subset of the terms from the sum $^1S_{r,k}$, $^2S_{r,k} \subseteq {}^1S_{r,k}$. In the last clock cycle, some terms are switched off.

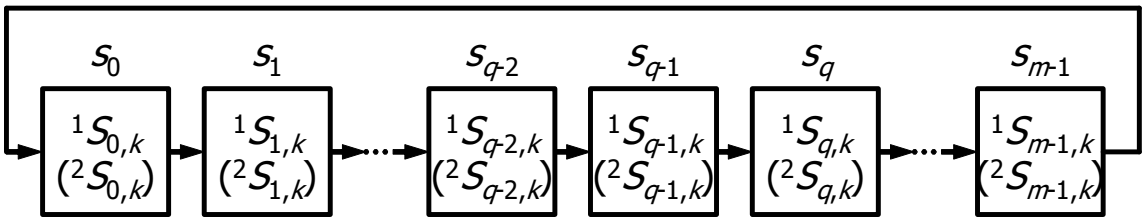

Figure 6.8: **(End-Correction Multiplier, GCEC)** Evaluation of the terms in the stages of the register C in the end-correction digit-serial multiplier.

As the indices j form a sequence, concretely $u_{i,k} = v_{i,k-1} + 1$, the equation 6.8 may be rewritten

$$c_i = \sum_{k=0}^{q-2} \sum_{j=u_{i,k}}^{v_{i,k}} T_{i,j} + \sum_{j=u_{i,q-1}}^{v_{i,q-1}} T_{i,j} = \sum_{j=u_{i,0}}^{v_{i,q-2}} T_{i,j} + \sum_{j=u_{i,q-1}}^{v_{i,q-1}} T_{i,j}$$
$$= \sum_{j=iD}^{(i+q-1)D-1} T_{i,j} + \sum_{j=(i+q-1)D}^{iD+m-1} T_{i,j} = \sum_{j=iD}^{iD+m-1} T_{i,j}$$

As the indices are reduced modulo m, the Equation 4.1 is satisfied. □

The block structure of combinational logic in the stage s_r is depicted in Figure 6.7. The evaluation of the terms in the register C is described in Figure 6.8.

Note that we have found out that a circular structure of the general digit-serial multiplier has been probably developed in [LL02]. Unfortunately, the authors neither describe their solution, nor offer any reference. According to a brief e-mail communication with the authors, their structure is similar to the end-correction multiplier, however, it is more complex. They calculate the product during $q + 1$ clock cycles using two sets of combinational logic. The first set, that evaluates D terms in each stage (similarly to our solution), is used during the first q clock cycles. In an additional clock cycle, the calculation is switched to the second set, that subtracts superfluous terms. This set evaluates ($m \mod D$) terms in each stage.

In spite of the lack of information, we believe that the following comparison of the end-correction multiplier and the multiplier [LL02] is not far from the truth.

The necessity of the second set of logic increases the area of combinational logic in [LL02] by the factor between 0% (for the cases where $m \mod D$ is close to 0) and 100% (for the cases where $m \mod D$ is close to D) in contrast to our solution of the end-correction multiplier. In average, the area of combinational logic in [LL02] is 50% larger than the area of combinational logic in the end-correction multiplier. In addition, the multiplexers used for alternating the two sets of logic take larger area than AND gates performing similar function in the end-correction multiplier. The number of flip-flops is the same in both multipliers.

Moreover, the additional clock cycle increases the calculation time. This fact is important mainly for larger Ds, e.g., if $m = 173$ and $D = 22$, then the number of clock cycles spent for one multiplication in the end-correction multiplier is $q = \lceil \frac{m}{D} \rceil = 8$, while the number of clock cycles spent in the multiplier [LL02] is $q + 1 = 9$. If both multipliers run at the same frequency, then in this case the time of multiplication is 12.5% larger in the multiplier [LL02].

In other words, the end-correction multiplier is systematically both smaller and faster than the multiplier briefly outlined in [LL02]. The area of combinational logic in the end-correction multiplier is about 0–50% smaller (33% in average) than the area of the multiplier in [LL02]. While the multiplier [LL02] needs $q + 1$ clock cycles for the calculation of one product, the end correction multiplier needs q clock cycles only.

Example 6.3. Let $m = 11$. It is shown in Figures 6.9 and 6.10 how the terms are evaluated in the stages of the register C in the end-correction

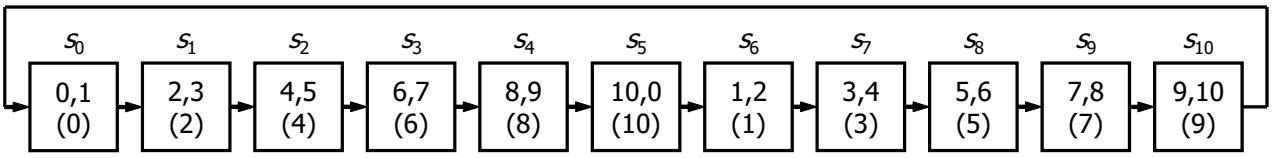

Figure 6.9: Evaluation of the terms in the stages of the end-correction digit-serial multiplier for $m = 11$ and $D = 2$. Values of the second indices j of terms $T_{i,j}$ are introduced. Multiplication takes $q = 6$ clock cycles. In the last clock cycle, $qD - m = 6 \cdot 2 - 11 = 1$ term is switched off in each stage; the stage then evaluates remaining $m - (q-1)D = 11 - 5 \cdot 2 = 1$ term.

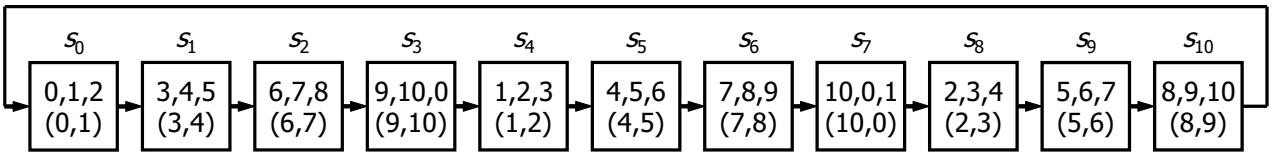

Figure 6.10: Evaluation of the terms in the stages of the end-correction digit-serial multiplier for $m = 11$ and $D = 3$. Values of the second indices j of terms $T_{i,j}$ are introduced. Multiplication takes $q = 4$ clock cycles. In the last clock cycle, $qD - m = 4 \cdot 3 - 11 = 1$ term is switched off in each stage; the stage then evaluates remaining $m - (q-1)D = 11 - 3 \cdot 3 = 2$ terms.

digit-serial multiplier. The indices in parentheses belong to the sets of terms ${}^2S$.

6.4 Circular Multiplier with Distributed Overlap (GCDIST and GCDO)

The design of this multiplier arises from the design of a circular multiplier (with a concentrated overlap). In contrast to GC, the main idea of this multiplier is to distribute the overlap, make it smaller and simplify the control. The number of terms evaluated in this multiplier is the same as in the standard one, i.e. $D \times m$ terms $T_{i,j}$. The area overhead insists in less than $\frac{m}{2}$ AND gates in comparison with the standard multiplier.

As stated in Section 4.5, in case of a standard multiplier we may split the multiplier into D pipeline blocks, each containing q consecutive stages. All D pipeline blocks together create the whole multiplier, as $D \times q = m$. This rule can be satisfied only if D divides m. For such D that do not divide m, the value $q = \lceil \frac{m}{D} \rceil$ and for these cases $D \times q > m$.

To divide the multiplier into D pipeline blocks as equally as possible, some of pipeline blocks must contain q pipeline stages, while some other must contain $q-1$ stages only. Exactly, $(m \mod D)$ pipeline blocks will contain q stages (we denote them as *long pipeline blocks*), while the remaining $(D-(m \mod D))$ pipeline blocks will contain $q-1$ stages (we denote them as *short pipeline blocks*).

The idea of this multiplier is to distribute calculation of all m terms $T_{i,j}$ among only $q-1$ stages of each pipeline block, while the calculation would again last q clock cycles. Therefore, if the pipeline block is q stages long (a long pipeline block), then one of the stages will be empty (nothing will be computed in this stage), while if the pipeline block is $q-1$ stages long (a short pipeline block), then the first $q-1$ stages of the following block will be switched-off in the last, q-th clock cycle (or, equivalently, the last $q-1$ stages of the previous pipeline block will be switched-off in the first clock cycle) to "compute" the empty stage. Switching-off some stages is less hardware complex than multiplexing between two different sets of logic — while the multiplexer is relatively complex, the AND gate used for switching-off is relatively simple. The evaluation of the result takes $q = \lceil \frac{m}{D} \rceil$ clock cycles and cannot be shortened — the empty stage appears at different clock cycles for different partial results c_i.

There are many possible definitions of the multiplier. One of them may assume that the first $(m \mod D)$ pipeline blocks are q stages long (long pipeline blocks), succeeded by $(D-(m \mod D))$ pipeline blocks being $q-1$ stages long (short pipeline blocks). For such assumption we will present the following description.

Rule 6.4 (a circular digit-serial multiplier with distributed overlap, GCDIST)**.** *Let* $q = \lceil \frac{m}{D} \rceil$ *be the number of clock cycles of one multiplication. Let* $D_x = \left\lceil \frac{m}{q-1} \right\rceil$ *be the maximum number of terms being evaluated in one stage. Let* $F = ((m \mod D) \times q)$ *be the number of stages of all long pipeline blocks. Then, in the* k*-th clock cycle* $(0 \leq k \leq q-1)$*, the stage* s_r $(0 \leq r \leq m-1)$ *evaluates the sum of terms* $S_{r,k}$*:*

(long pipeline blocks):

a) if $((r < F) \wedge (r \mod q = 0))$*:*

$$S_{r,k} = \emptyset$$

b) if $((r < F) \wedge (r \mod q \neq 0))$*:*

$$S_{r,k} = \sum_{j=u_r}^{v_r} T_{r-k,j},$$

where

$$u_r = (r - 1 \mod q) \times D_x,$$
$$v_r = \min\{u_r + D_x - 1; m - 1\}.$$

(short pipeline blocks):

c) if $((r \geq F) \wedge (k = q - 1))$*:*

$$S_{r,k} = \emptyset$$

d) if $((r \geq F) \wedge (k \neq q - 1))$*:*

$$S_{r,k} = \sum_{j=u_r}^{v_r} T_{r-k,j},$$

where

$$u_r = ((r - F) \mod (q - 1)) \times D_x,$$
$$v_r = \min\{u_r + D_x - 1; m - 1\}.$$

The sum of terms $S_{r,k}$ *is added to the partial result of the bit* c_{r-k}*, which is, due to the rotation of the register* C*, present in the stage* s_r *during the* k*-th clock cycle. The result* $c_0c_1c_2 \ldots c_{m-1}$ *is available in stages* $s_{q-1}s_qs_{q+1} \ldots s_{m-1}s_0 \ldots s_{q-2}$ *after* q *clock cycles.*

This multiplier evaluates the same amount of terms as the standard multiplier. The area overhead lies in the excessive number of AND gates necessary for switching-off certain stages in the last clock cycle. In the worst case, the number of necessary AND gates could be almost m.

But, one more improvement in the area can be done. As mentioned above, to implement the "empty stage" in short pipeline blocks, we can *either* switch-off the first $q-1$ stages of the following block in the last clock cycle *or* switch-off the last $q-1$ stages of the previous block in the first clock cycle. Fortunately, we can join these two approaches together. We do not need to switch-off all short pipeline blocks in the last (or first) clock cycle; instead, we can switch-off

only all *odd* short pipeline blocks in the first *and* the last clock cycles, while *even* blocks will not be switched at all. This approach a lot more minimizes hardware resources. The number of additional AND gates is consequently less than $\frac{m}{2}$.

Rule 6.5 (an optimized circular digit-serial multiplier with a distributed overlap, GCDO)**.** *Let* $q = \lceil \frac{m}{D} \rceil$ *be the number of clock cycles of one multiplication. Let* $D_x = \left\lceil \frac{m}{q-1} \right\rceil$ *be the maximum number of the terms being evaluated in one stage. Let* $F = ((m \mod D) \times q)$ *be the number of stages of all long pipeline blocks. Then, in the k-th clock cycle* $(0 \leq k \leq q-1)$*, the stage* s_r $(0 \leq r \leq m-1)$ *evaluates the sum of terms* $S_{r,k}$*:*

(long pipeline blocks):

a) if $((r < F) \wedge (r \mod q = 0))$*:*

$$S_{r,k} = \emptyset$$

b) if $((r < F) \wedge (r \mod q \neq 0))$*:*

$$S_{r,k} = \sum_{j=u_r}^{v_r} T_{r-k,j},$$

where

$$\begin{aligned} u_r &= (r-1 \mod q) \times D_x, \\ v_r &= \min\{u_r + D_x - 1; m-1\}. \end{aligned}$$

(short pipeline blocks):

c) if $(r \geq F) \wedge \left(\left(\left\lfloor \frac{r-F}{q-1} \right\rfloor \mod 2 = 0 \right) \wedge ((k=0) \vee (k=q-1)) \right)$*:*

$$S_{r,k} = \emptyset$$

d) if $(r \geq F) \wedge \neg \left(\left(\left\lfloor \frac{r-F}{q-1} \right\rfloor \mod 2 = 0 \right) \wedge ((k=0) \vee (k=q-1)) \right)$*:*

$$S_{r,k} = \sum_{j=u_r}^{v_r} T_{r-k,j},$$

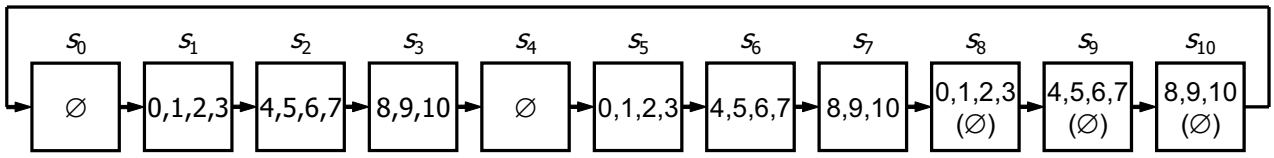

Figure 6.11: Evaluation of the terms in the stages of circular digit-serial multiplier with a distributed overlap (GCDIST as well as GCDO) for $m = 11$ and $D = 3$. The values of the second indices j of the terms $T_{i,j}$ are introduced. Multiplication logic is switched off in stages s_8 through s_{10} in the last clock cycle (denoted by $\emptyset$). Multiplication takes $q = 4$ clock cycles.

where

$$u_r = ((r - F) \mod (q - 1)) \times D_x,$$
$$v_r = \min\{u_r + D_x - 1; m - 1\}$$

The sum of terms $S_{r,k}$ is added to the partial result of the bit c_{r-k}, which is, due to the rotation of the register C, present in the stage s_r during the k-th clock cycle. The result $c_0c_1c_2 \dots c_{m-1}$ is available in stages $s_{q-1}s_qs_{q+1} \dots s_{m-1}s_0 \dots s_{q-2}$ after q clock cycles.

Example 6.4. Let $m = 11$ and $D = 3$. It is shown in Figure 6.11 how the terms are evaluated in the stages of the register C in GCDIST and GCDO multipliers. Coincidently, in this case, the multipliers contain only one short pipeline block and hence the structure of both multipliers is the same.

The difference between GCDIST and GCDO multipliers is illustrated in Figures 6.12 and 6.13. The multipliers are constructed for $m = 13$ and $D = 3$. The multipliers contain two short pipeline blocks. While both short pipeline blocks $s_5 \dots s_8$ and $s_9 \dots s_{12}$ are equipped with switching-off AND gates in the GCDIST multiplier, only one (the odd one) short pipeline block $s_5 \dots s_8$ is equipped with AND gates in the GCDO multiplier.

6.5 Area and Critical Path Length

Let C_N be the number of non-zero entries λ_{ij} in a multiplication matrix $\mathbf{M}$. In Tables 6.1 and 6.2, we summarize and compare proposed 4 architectures of

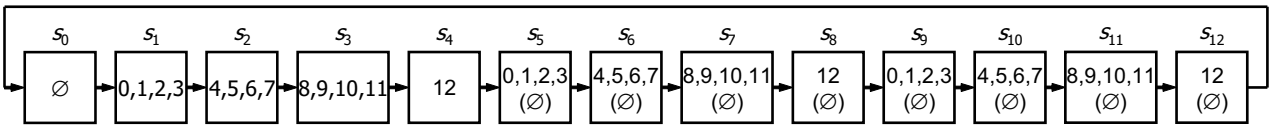

Figure 6.12: Evaluation of the terms in the stages of a circular digit-serial multiplier with a distributed overlap (GCDIST) for $m = 13$ and $D = 3$. The values of the second indices j of the terms $T_{i,j}$ are introduced. Multiplication logic is switched off in stages s_5 through s_{12} in the last clock cycle (denoted by $\emptyset$). Multiplication takes $q = 5$ clock cycles.

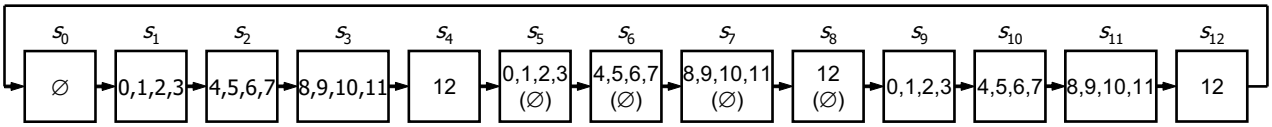

Figure 6.13: Evaluation of the terms in the stages of an optimized circular digit-serial multiplier with a distributed overlap (GCDO) for $m = 13$ and $D = 3$. The values of the second indices j of the terms $T_{i,j}$ are introduced. Multiplication logic is switched off in stages s_5 through s_8 in the first and last clock cycles (denoted by $\emptyset$). Multiplication takes $q = 5$ clock cycles.

the general digit-width normal basis multipliers with the standard digit-serial and the bit-serial multiplier. Below, we briefly comment on the tables.

6.5.1 Bit-Serial Multiplier

In the bit-serial multiplier each stage evaluates exactly one term $T_{i,j}$. According to Equation 4.2, each term is implemented by one AND gate and at most $\left\lceil \frac{C_N}{m} \right\rceil$ XOR gates which form a tree with depth $\leq \left\lceil \log \left\lceil \frac{C_N}{m} \right\rceil \right\rceil$.

6.5.2 Standard Digit-Serial Multiplier

When D divides m, every stage evaluates exactly D terms; consequently, the entire logic of the standard multiplier evaluates mD terms. Every term is implemented by one AND gate and at most $\left\lceil \frac{C_N}{m} \right\rceil$ XOR gates, which form a tree with depth $\leq \left\lceil \log_2 \left\lceil \frac{C_N}{m} \right\rceil \right\rceil$. The terms evaluated in one stage are summed up by another tree of XOR gates with depth $\lceil \log_2 D \rceil$.

6.5.3 Circular Digit-Serial Multiplier

The circular multiplier with a concentrated overlap contains additional logic that in each of the first $q-1$ stages evaluates an alternative sum of terms. Each sum of terms consists of D terms at maximum. First $q-1$ stages contain multiplexers that may lie on a critical path. Moreover, we need also additional control logic for successive switching the first $q-1$ stages. This control logic may be implemented either by a shift register or by a counter with a decoder.

6.5.4 Linear Digit-Serial Multiplier

The linear digit-serial multiplier contains additional $q-1$ stages. Every stage evaluates at most D terms, and hence the critical path of the linear multiplier is comparable with that of the standard multiplier.

6.5.5 End-Correction Digit-Serial Multiplier

In the end-correction digit-serial multiplier, exactly D terms are evaluated in every stage; some of them are switched-off in the last clock cycle. To switch them off, one AND gate is necessary in every stage, which may lie on the critical path.

6.5.6 Circular Multiplier with Distributed Overlap

In this multiplier, each stage may evaluate at most $2D$ terms. We must note, however, that this is a marginal case. Some stages contain one AND gate to switch-off all logic in a stage. This AND gate may lie on a critical path. The total number of additional AND gates is less than m in case of GCDIST or less than $\frac{m}{2}$ in case of GCDO.

6.6 Implementation Results

We implemented the above multipliers in Xilinx Virtex 4 VLX25 FPGA with Leonardo Spectrum 2005 and ISE 7.1i. We measured the area of the design by the number of slices used, and observed the minimum clock period. The time of multiplication is a product of the minimum clock period and the number of clock cycles. The quality factor is a reciprocal of the time area product.

Table 6.1: Hardware resources

	AND	XOR	MUX	flip-flops
bit-serial ($D = 1$)	m	$\leq C_N$	0	$3m$
standard digit-serial	mD	$\leq C_N D$	0	$3m$
GC	$< mD + m$	$< C_N D + C_N$	$q - 1$	$3m + q - 1$
GL	$< mD + m$	$< C_N D + C_N$	0	$3m + q - 1$
GCEC	$mD + m$	$\leq C_N D$	0	$3m$
GCDO	$< mD + \frac{m}{2}$	$\leq C_N D$	0	$3m$

Table 6.2: Critical path length

	AND	XOR	MUX
bit-serial ($D = 1$)	1	$\leq \lceil \log \lceil C_N/m \rceil \rceil$	0
standard digit-serial	1	$\leq \lceil \log \lceil C_N/m \rceil \rceil + \lceil \log D \rceil$	0
GC	1	$\leq \lceil \log \lceil C_N/m \rceil \rceil + \lceil \log D \rceil$	1
GL	1	$\leq \lceil \log \lceil C_N/m \rceil \rceil + \lceil \log D \rceil$	0
GCEC	$1 - 2$	$\leq \lceil \log \lceil C_N/m \rceil \rceil + \lceil \log D \rceil$	0
GCDO	$1 - 2$	$\leq \lceil \log \lceil C_N/m \rceil \rceil + \lceil \log D \rceil + 1$	0

To compare digit-serial multipliers of a general digit width with the standard multiplier, we have chosen the degree as a composite number $m = 180$. Since the structure of the circular (GC) and the end-correction multiplier (GCEC) is the same as the structure of a standard multiplier, whenever the digit width D divides the degree m (in these cases sums ${}^1S$ and ${}^2S$ are identical), we worked out another set of experiments for a prime number $m = 173$. In this set, we compared the architectures of general digit-serial multipliers only.

Implementation results for the first twenty values of D are summarized in Tables 6.3 and 6.4. As we expected, the linear multiplier (GL) is systematically better than the circular multiplier (GC). As the GC multiplier contains additional multiplexers that lie on a critical path, it has both greater area and longer critical path. For that reason, we omit results for GC in the following diagrams. We will note that we can compare only cases where the digit width D does not divide the degree m. If $D|m$, then the GC multiplier is identical with the standard one.

The dependencies of multiplication time on the digit width D for the composite degree $m = 180$ and for the prime degree $m = 173$ are available in Figure 6.14. As expected, multiplication time decreases with growing D. Note that results for the standard multiplier and Ds not dividing the degree m cannot exist.

The dependencies of the quality factor on the digit width D are available in Figure 6.15. The quality factor here is evaluated as the throughput/area ratio of the multiplier itself. It decreases with the growing D as the achievable clock frequency decreases for the larger Ds. However, when using the multiplier in a cryptographic system, the critical paths may lie in other units outside the multiplier. For such cases, the quality factor of the whole system may increase with growing D unless the critical path transfers to the multiplier. Also, the fixed area of other units (a controller, registers, etc.) used in a cryptographic system causes a shift of the optimum of the quality factor to higher values of D.

We observed that in case of the end-correction multiplier, our synthesis tools produced the structure shown in Figure 6.16 instead of the structure from Figure 6.7. Hence the critical path length was greater than that assumed in Table 6.2. We tried to synthesize the end-correction multiplier with other (older) versions of Leonardo Spectrum and ISE, but the results were almost identical.

Except of the length of the critical path in the end-correction multiplier, the implementation results correspond to the theoretical values introduced in

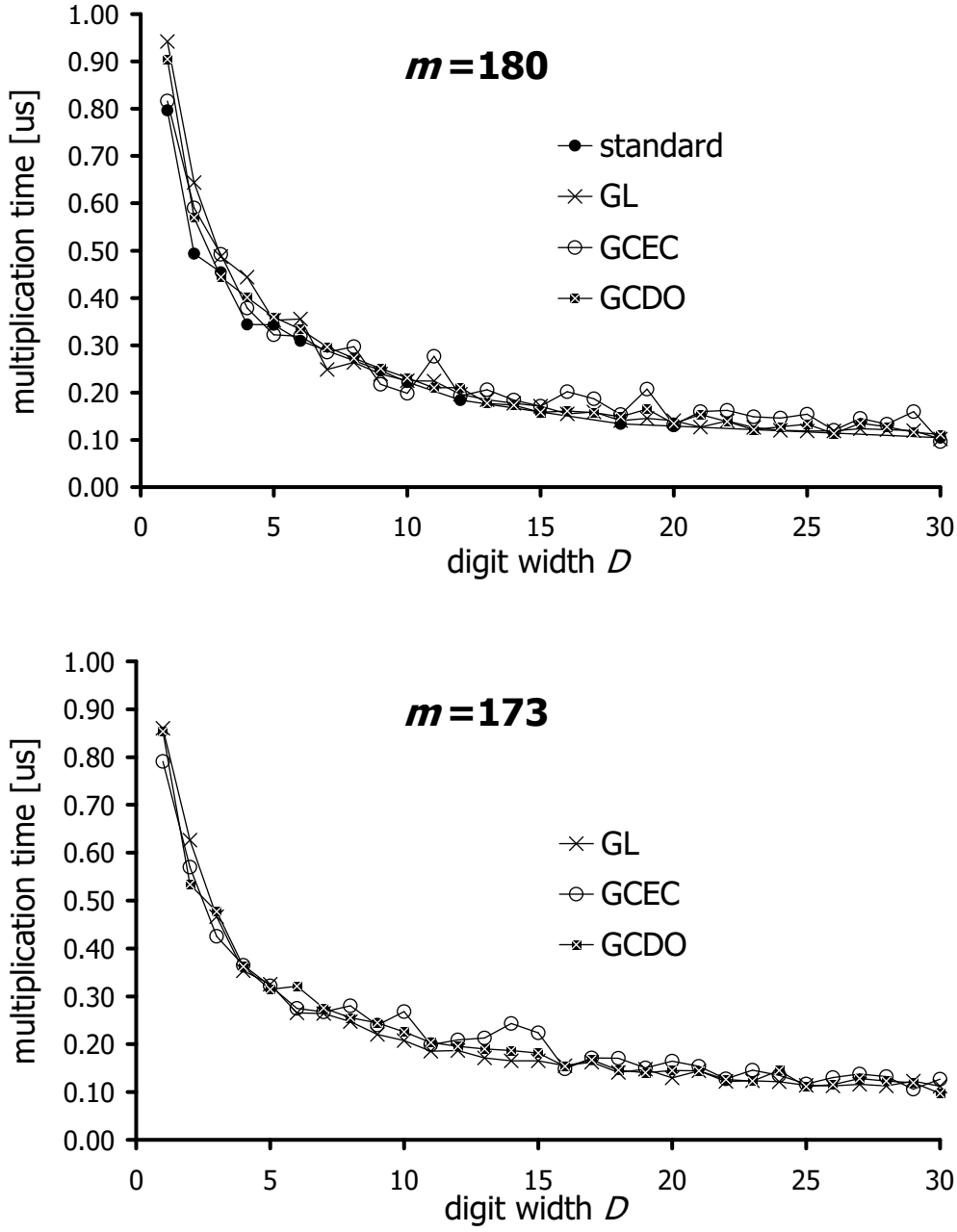

Figure 6.14: Time spent for the calculation of one product for variable digit widths

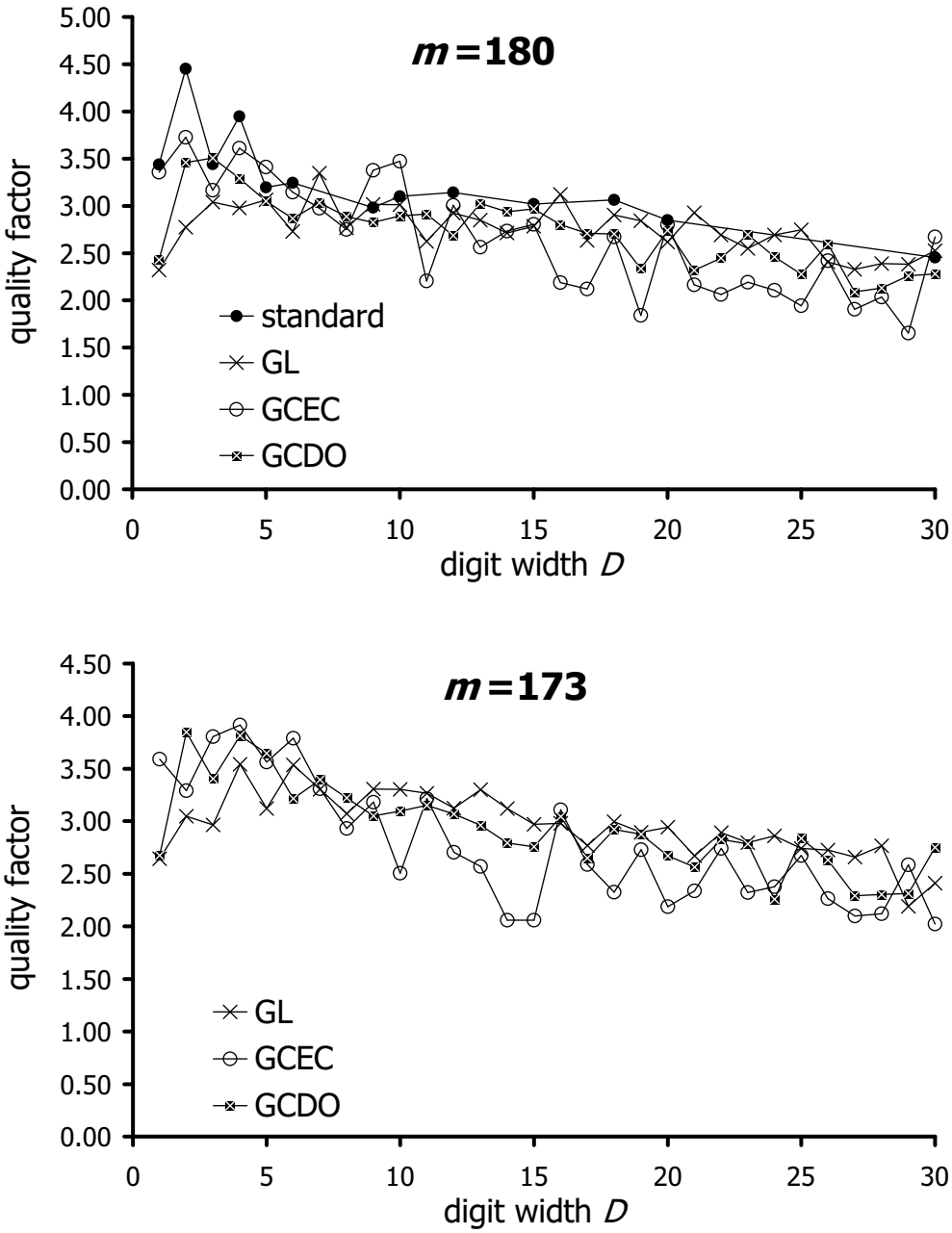

Figure 6.15: Quality factor as a function of the digit width

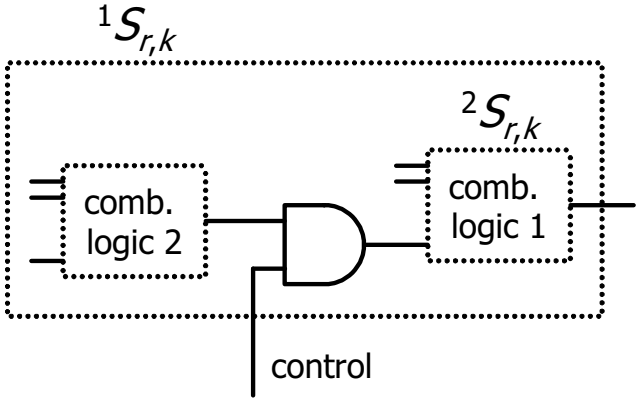

Figure 6.16: Combinational logic synthesized in the stage of the end-correction multiplier

Tables 6.1 and 6.2. They confirm that the general digit-serial multipliers are of the same quality as the standard one. In other words, we do not pay much for the added flexibility.

In the linear multiplier, the number of additional stages is large for small Ds. This causes the excess in area of the GL multiplier over the GCEC and GCDIST multiplier for smaller Ds. For larger Ds, the area of all multipliers is comparable, while the length of the critical path in the GCEC and GCDIST multiplier prevails. Generally, the GCEC and GCDIST multipliers have a better quality factor for smaller Ds ($D < 6$), while the GL multiplier is better for larger Ds ($D > 6$). In particular, for the results presented in Tables 6.3 and 6.4 and in Figures 6.14 and 6.15, the end-correction multiplier had 30% better quality factor than the linear one for $D = 3$, while the linear was 50% better for $D = 14$.

Table 6.3: Implementation results for $m = 180$

	Time of Multiplication [us]					Area [slices]					Quality factor				
D	std	GC	GL	GC EC	GC DO	std	GC	GL	GC EC	GC DO	std	GC	GL	GC EC	GC DO
1	0.80	0.82	0.94	0.82	0.90	365	365	457	365	455	3.44	3.35	2.32	3.35	2.43
2	0.49	0.49	0.64	0.59	0.57	455	455	560	455	508	4.45	4.48	2.78	3.73	3.46
3	0.45	0.44	0.49	0.49	0.44	640	643	675	642	643	3.44	3.53	3.04	3.17	3.51
4	0.34	0.38	0.44	0.38	0.40	736	735	756	731	757	3.95	3.57	2.98	3.61	3.29
5	0.34	0.32	0.35	0.32	0.36	912	910	926	911	912	3.19	3.44	3.06	3.41	3.05
6	0.31	0.29	0.36	0.32	0.33	997	1013	1031	997	1045	3.24	3.37	2.73	3.14	2.86
7		0.30	0.25	0.29	0.30		1223	1201	1177	1113		2.77	3.35	2.97	3.03
8		0.27	0.26	0.30	0.27		1370	1369	1224	1272		2.71	2.77	2.75	2.89
9	0.25	0.23	0.24	0.22	0.25	1361	1368	1384	1363	1411	2.98	3.17	3.02	3.38	2.82
10	0.22	0.21	0.23	0.20	0.23	1451	1458	1474	1449	1494	3.10	3.32	3.01	3.47	2.89
11		0.22	0.22	0.28	0.21		1699	1699	1638	1634		2.62	2.62	2.21	2.91
12	0.18	0.19	0.20	0.19	0.21	1725	1731	1741	1721	1768	3.14	3.10	2.92	3.01	2.68
13		0.18	0.18	0.21	0.18		1921	1901	1897	1863		2.84	2.85	2.56	3.02
14		0.17	0.18	0.18	0.17		2088	2078	1987	1967		2.75	2.71	2.73	2.94
15	0.16	0.16	0.17	0.17	0.16	2079	2081	2093	2081	2128	3.02	3.02	2.78	2.80	2.97
16		0.16	0.16	0.20	0.16		2054	2062	2261	2216		3.07	3.12	2.19	2.80
17		0.16	0.16	0.19	0.16		2379	2381	2527	2343		2.69	2.63	2.12	2.71
18	0.13	0.14	0.14	0.15	0.15	2442	2442	2449	2440	2485	3.06	3.01	2.90	2.67	2.70
19		0.15	0.15	0.21	0.17		2419	2424	2617	2589		2.75	2.84	1.84	2.33
20	0.13	0.15	0.14	0.13	0.13	2713	2712	2719	2714	2712	2.85	2.54	2.62	2.80	2.74

Table 6.4: Implementation results for $m = 173$

	Period [ns]				Time of Mult. [us]				Area [slices]				Quality factor			
D	GC	GL	GC EC	GC DO	GC	GL	GC EC	GC DO	GC	GL	GC EC	GC DO	GC	GL	GC EC	GC DO
1	4.86	4.97	4.57	4.93	0.84	0.86	0.79	0.85	352	440	352	438	3.38	2.64	3.59	2.67
2	7.41	7.20	6.55	6.13	0.64	0.63	0.57	0.53	619	524	533	487	2.51	3.05	3.29	3.85
3	10.86	8.04	7.33	8.23	0.63	0.47	0.43	0.48	725	723	618	615	2.19	2.96	3.81	3.41
4	13.15	8.03	8.30	8.23	0.58	0.35	0.37	0.36	815	799	699	724	2.12	3.54	3.92	3.81
5	9.83	9.29	9.19	8.98	0.34	0.33	0.32	0.31	989	986	872	872	2.94	3.12	3.56	3.65
6	12.19	9.15	9.48	11.07	0.35	0.27	0.27	0.32	1100	1066	960	969	2.57	3.53	3.79	3.21
7	12.05	10.56	10.69	11.03	0.30	0.26	0.27	0.28	1173	1147	1131	1068	2.83	3.30	3.31	3.40
8	12.61	11.24	12.74	11.61	0.28	0.25	0.28	0.26	1319	1316	1218	1215	2.73	3.07	2.93	3.22
9	14.02	11.02	12.01	12.23	0.28	0.22	0.24	0.24	1396	1372	1309	1340	2.56	3.31	3.18	3.05
10	14.15	11.53	14.90	12.53	0.25	0.21	0.27	0.23	1475	1458	1488	1432	2.66	3.30	2.50	3.10
11	14.24	11.58	12.44	12.73	0.23	0.19	0.20	0.20	1661	1652	1563	1557	2.64	3.27	3.22	3.15
12	14.41	12.47	13.92	13.02	0.22	0.19	0.21	0.20	1729	1711	1772	1669	2.68	3.12	2.70	3.07
13	14.02	12.23	15.22	13.56	0.20	0.17	0.21	0.19	1781	1769	1826	1778	2.86	3.30	2.57	2.96
14	13.16	12.70	18.70	14.39	0.17	0.17	0.24	0.19	1946	1941	1999	1914	3.00	3.12	2.06	2.79
15	13.36	13.76	18.64	15.09	0.16	0.17	0.22	0.18	2058	2038	2171	2003	3.03	2.97	2.06	2.76
16	14.86	14.14	13.48	13.99	0.16	0.16	0.15	0.15	2176	2157	2170	2135	2.81	2.98	3.11	3.04
17	16.71	14.81	15.59	15.19	0.18	0.16	0.17	0.17	2219	2217	2253	2260	2.45	2.77	2.59	2.65
18	14.42	14.10	17.09	14.61	0.14	0.14	0.17	0.15	2382	2368	2515	2342	2.91	3.00	2.33	2.92
19	14.99	14.79	15.08	14.05	0.15	0.15	0.15	0.14	2345	2336	2430	2474	2.84	2.89	2.73	2.88
20	17.51	14.44	18.30	16.15	0.16	0.13	0.16	0.15	2614	2614	2776	2576	2.43	2.94	2.19	2.67

Chapter 7

Scalable Shifter Synthesis

Our arithmetic unit (AU) design (Figure 7.1) is dominated by the multiplier in both area and time. Moreover, the multiplier has a natural scaling parameter, namely the digit width. We consider the design of the multiplier to be a primary task. This leaves us to consider the design of the shifter to be a dependent task, that is, the shifter will be optimized in a scope of the dominating block. Our aim is to develop a procedure where the digit-width controls the design of the entire AU.

Surprisingly, the problem of an optimum shifter design exhibits a rich structure and complex solution space. Although the solution presented here is heuristic, proving some problem-specific theorems played a role in the decomposition of the problem.

We tried to solve the task by behavioral synthesis software; however, we never persuaded the synthesizers to decompose the rotations in time and space as described below.

The problem of the synthesis of an optimum shifter in the frame of a normal basis arithmetic unit was published in [A.4].

7.1 Problem Formulation

Let m be the degree of the finite field we work in, $GF(2^m)$. Let

$$(b_r b_{r-1} \dots b_1 b_0)$$

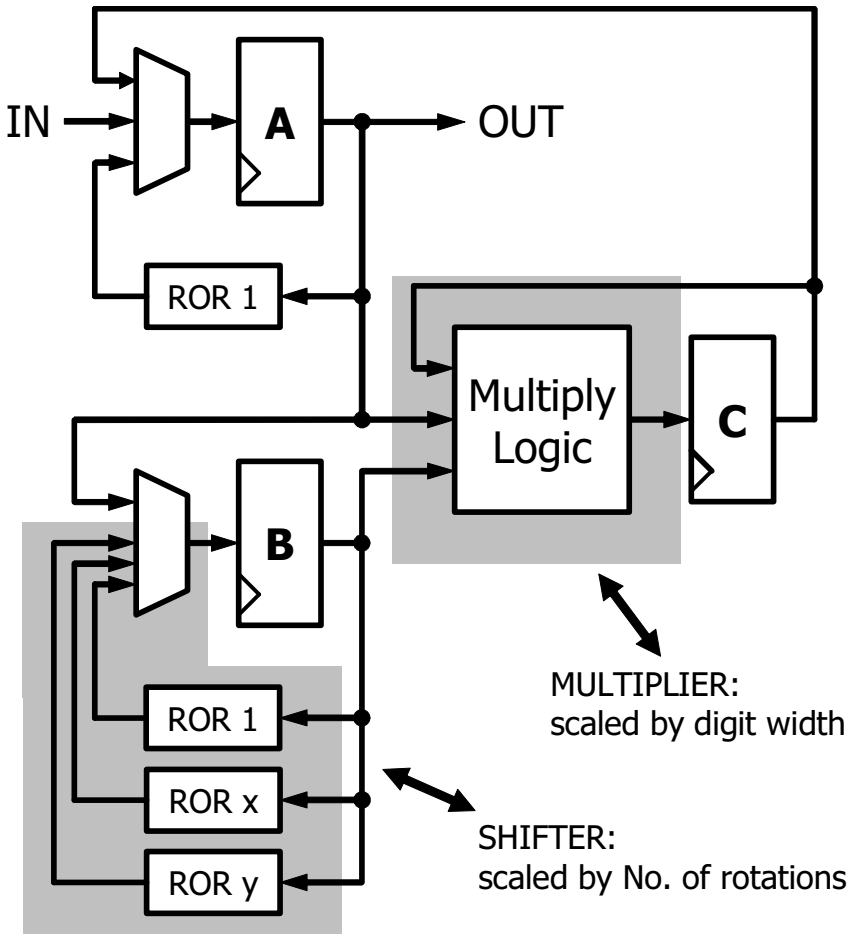

Figure 7.1: Arithmetic unit contains 2 scalable subunits, the multiplier and the shifter

be the binary representation of $m-1$ such that the most significant bit $b_r = 1$. The set of rotations required by the ITT algorithm is

$$\mathbb{K} = \{k_i, i = r \ldots 1; k_i = (b_r \ldots b_i)\} \tag{7.1}$$

The binary representation of k_i is $b_r \ldots b_i$. Each of the rotations is performed exactly once in an inversion operation.

The task is to synthesize a shifter which implements all rotations in the set $\mathbb{K}$ and which, when combined with the rest of the AU, gives the optimum throughput/area ratio.

When any unit is scaled, its area A and time t vary in opposite directions. The time t depends on the number of clock cycles T spent in a given calculation and on the critical path length τ in hardware.

The area A can be measured by any metric used for the rest of the unit. The number of primitive gates is the most common measure; a technology-dependent measure such as the number of transistors or programmable blocks is also suitable.

Similarly, the critical path length τ can be given either as the number of logic levels, or in the units of time as appropriate.

To compare differently scaled units, we use the quality measure

$$Q = At = AT\tau$$

which should be minimized.

Let A, T, and τ be the area, the total number of clock cycles spent in rotations, and the critical path length of the shifter. Let A_0, T_0, and τ_0 be the area, the total number of clock cycles spent in multiplications, and the critical path length of the rest of the AU. The measure of quality of the entire unit, and hence our optimization criterion, is

$$Q = (A + A_0)(T + T_0) \max\{\tau, \tau_0\} \tag{7.2}$$

This equation also shows the two dependencies between the shifter and the multiplier. Firstly, the ratio of the shifter area and time must be "the right one". For each A_0, T_0 and τ_0, the area and number of clock cycles of the shifter should be adjusted to achieve a minimum Q of the entire unit.

Secondly, the shifter may slow down the AU clock. In this case, not only the rotation time T_0 is longer, but also the multiplication time is longer —

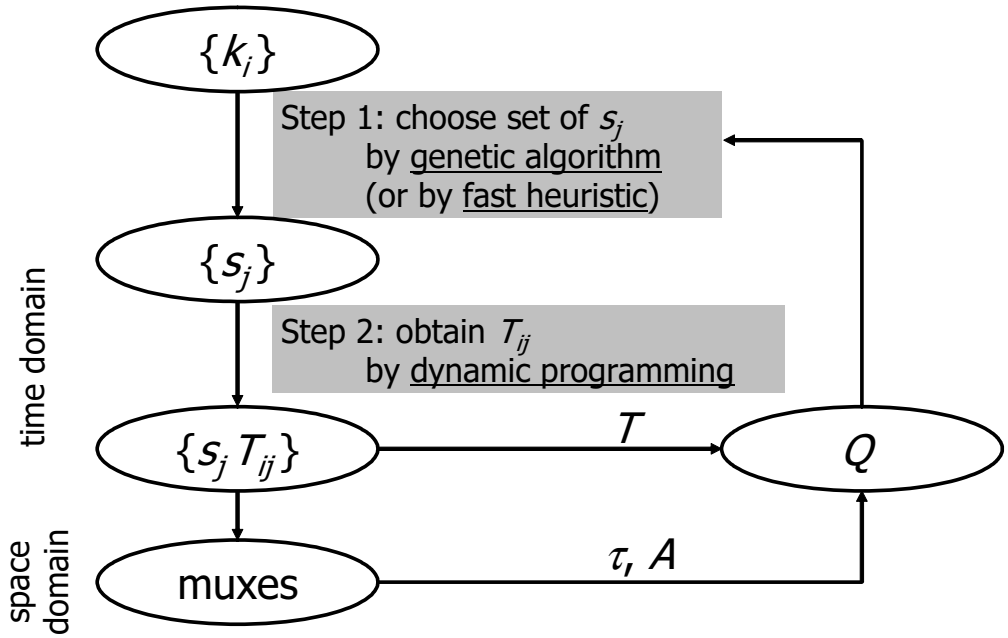

Figure 7.2: Approach overview.

$T_0\tau$ instead of $T_0\tau_0$. As the multiplier dominates, this penalty may become unacceptable.

7.2 Approach Overview

To scale the shifter, we first find a limited set $\mathbb{S} = \{s_j, 1 \leq j \leq n, n \leq r\}$ of n rotations to be implemented in hardware and their combinations which realize the given set $\mathbb{K}$ of rotations requested by the ITT algorithm. This is the *time domain subproblem*. Then, we construct the hardware — the *space domain subproblem*. We approximate the hardware as an n–input multiplexer.

An iterative process (a genetic algorithm in this case) chooses a set of rotations. Then, a dynamic programming procedure finds the number of clock cycles required for the whole computation in $O(m^2 \log m)$ time, which is acceptable for the given range of m. The rotations compose modulo m, which makes many more rotation sets feasible.

Finally, the area and critical path length of the shifter are estimated, giving the value of the optimization criterion of the iterative process. The block diagram of the process is depicted in Figure 7.2

7.2.1 Sub-optimum Rotation Set by a Genetic Algorithm

In the time-domain problem, we seek $n \leq r$ hardware-implemented rotations. Configuration is given by n values $s_i, 0 \leq s_i, m$. This is the phenotype of the genetic algorithm. The genotype (chromosome) is a fixed structure with a simple decoder. Constrained optimization is implemented by a penalty for each rotation k_i which the individual in question cannot implement.

The rest of the genetic algorithm is quite classical, with single-point crossover and linear scaling of the fitness values. The adaptive nature of the linear scaling causes the convergence to remain unchanged even in the presence of a large area and time of the multiplier, where the differences in evaluations are relatively small.

7.2.2 Sub-optimum Rotation Set by a Fast Heuristic

Based on the observation of the genetic algorithm result, we designed a much faster heuristic procedure for finding rotation set:

1. Find n_{max} such that the critical path length of an n_{max}-input multiplexer is not bigger than that of the multiplier.

2. For $i = 1 \ldots n_{max}$ do:

   (a) Implement rotation by 1.

   (b) If $i < \frac{r}{2}$, implement every $\lceil \frac{r}{i} \rceil$-th rotation.

   (c) If there are still less than i implemented rotations, pick the rest at random.

   (d) Obtain the number of clock cycles by dynamic programming.

   (e) Compute the quality measure and record.

3. Choose the best solution.

The random assignment in Step 2.c is justified by the fact that using more distinct rotations than $\frac{r}{2}$ does not bring much improvement. The complexity of this algorithm is $O(m^2 \log^2 m)$.

7.3 Results

The optimizer was implemented using the GAlib C++ library [Wal]. A number of experiments has been performed, with m in the range interesting for elliptic curve cryptography, that is, from 160 to 250. The following Observations were made for the procedure using a genetic algorithm:

1. The solution found by the algorithm was an optimum one (where an optimum found by other methods was available)

2. Any realized rotation in an optimum solution was identical to a given rotation (i.e. $\mathbb{S} \subseteq \mathbb{K}$ in optimum case of $\mathbb{S}$), although even slightly sub-optimum solutions did not have this property.

3. No optimum and only a few of sub-optimum solutions employed the modularity of the rotation composition.

4. When the modularity was not used, the search space became disconnected, and more time was needed to achieve equivalent results.

5. All optimum solutions were best implemented by a multiplexer.

6. With the population size of 100, the algorithm required cca. 3000 generations to converge at $m = 160$, rising to 4000 at $m = 250$.

Table 7.1 illustrates the influence of the multiplier size on the shifter. The results were obtained for $m = 163$, where the required set of rotations is $\mathbb{K} = \{1, 2, 5, 10, 20, 40, 81\}$. The area of an n–input multiplexer was $1.5n$ and the critical path of the unit was outside the shifter.

The procedure using the fast heuristics gave the results almost identical to the other for the "normal" cases, i.e. for $n \leq 3$. Beyond that, the results were worse, in average, by 15%.

7.3.1 Future Work

The biggest challenge for the future is definitely Observation 2. If it was true in general, the search space of the problem would be drastically reduced. The proof would have to state the existence of an optimum solution with certain characteristics, and we are not aware of many techniques for such a proof.

Table 7.1: Shifters Adjusted to Different Multipliers

Multiplier			Shifter		
digit width	A_0	T_0	A	T	optimum set of rotations $\mathbb{S} = \{s_j\}$
1	489	1956	244	159	$\{1\}$
6	2934	326	488	24	$\{1, 10\}$
–	0	0	976	10	$\{1, 5, 20, 81\}$

In the case that Observation 2 does not hold in general, an almost obvious experiment to perform is to *encourage* the genetic algorithm to produce solutions for which better implementations than multiplexers exist.

A weaker assertion that might be eventually proven is Observation 3, however, it does not benefit the solution immediately (cf. Observation 4).

7.3.1.1 Reformulating the Observation 2

To better understand the problem of Observation 2 we bring a popular example involving heaps of coins.

We are given a number m. We should create several heaps of coins. The last heap should contain the coins of total volume $k_r = \left\lfloor \frac{m-1}{2} \right\rfloor$, the previous one should contain the volume $k_{r-1} = \left\lfloor \frac{k_r}{2} \right\rfloor$, etc., that is, the i-th heap has a volume $k_{i-1} = \left\lfloor \frac{k_i}{2} \right\rfloor$, until $k_1 = 1$. The heaps should be created with coins of only n nominal values. The problem is to select the nominal values such that the total number of all coins in all heaps would be minimal. Observation 2 says that the selected nominal values should be equal to some volumes of heaps, i.e., there are exactly n heaps with exactly 1 coin. In other words, when creating the set $\mathbb{S}$ of nominal values, the search space can be limited only to the set $\mathbb{K}$, which significantly reduces the complexity of the search algorithm.

Example 7.1. For $m = 180$ we should create the heaps with volumes $\mathbb{K} = \{1, 2, 5, 11, 22, 44, 89\}$. If we can use coins of $n = 2$ nominal values, then the optimum selection would be $\mathbb{S} = \{1, 11\}$, as illustrated in Figure 7.3a, while for $n = 3$ the nominal values should be $\mathbb{S} = \{1, 5, 22\}$, as illustrated in Figure 7.3b. In both cases $\mathbb{S} \subset \mathbb{K}$.

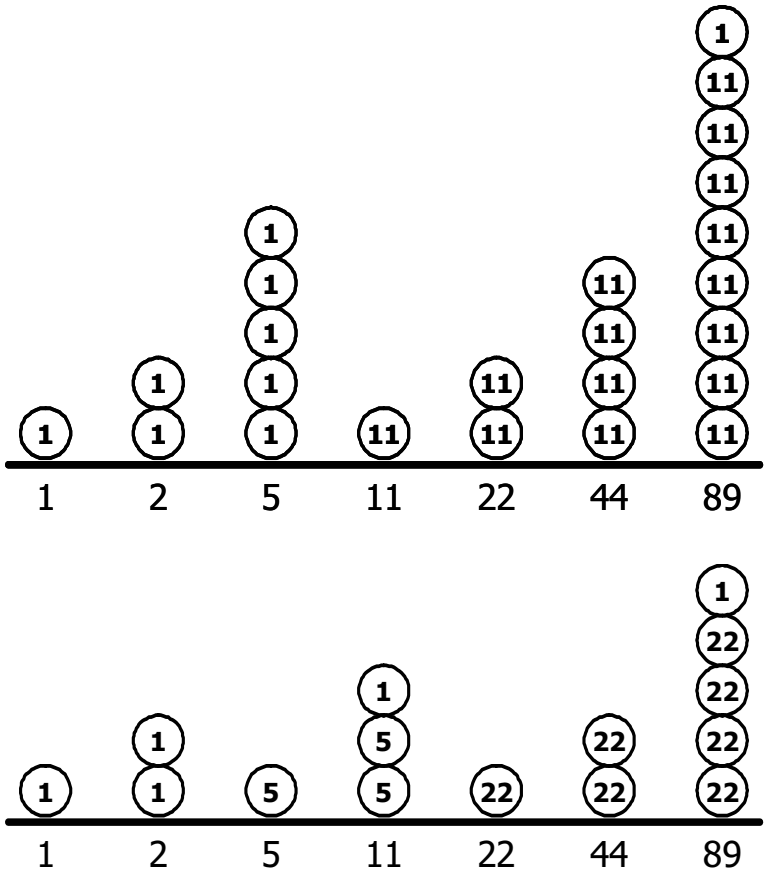

Figure 7.3: Heaps of coins for a) 2 nominal values and b) 3 nominal values of coins and for $m = 180$.

We also have made an extensive search for values of $m = 129 \ldots 256$ and for number of rotations $n = 2 \ldots 5$. For those values the Observation 2 was always satisfied. Nevertheless, it is still not sure if the Observation 2 holds for general values m and n. We presented this problem to the group of mathematicians at the Workshop on Factoring Large Numbers, held in Essen in April 2008, however, the problem still remains open.

7.4 Summary

In this chapter, we have presented a process which optimizes a block in a scope of another, dominating block with strong interdependences in the area and time of computation.

We decomposed the problem into time- and space-domain subproblems. A part of the time-domain problem is solved exactly and repeatedly inside the genetic algorithm. The solution of the space-domain subproblem was approximated and the approximation verified.

The results show that formally proving certain properties of the circuit can dramatically reduce the search space. Such proofs, although problem-dependent and hard to construct, could improve the performance of EDA software.

Chapter 8

Conclusions of Part I

We have developed an arithmetic unit working over elements of a finite field $GF(2^m)$ with a normal basis representation of the field elements. This arithmetic unit is able to perform two crucial operations, multiplication and inversion in a normal basis. The unit is applicable in smart cards and other public-key cryptography systems based on elliptic curves.

For its potential use in smart cards and embedded systems, we designed the unit to be as small as possible; actually, it is slightly bigger than the multiplier itself. The arithmetic unit contains two principal subunits, a multiplier and a shifter used in an inversion algorithm. As the application area requires the systems to be scalable, we developed methods that allow both subunits to scale as much as necessary. This extended scalability allows the designer to tune the cryptographic system to fit the design constraints optimally.

The multiplier is scaled by the digit width. As the standard normal basis multiplier may be scaled only for such digit widths D that divide the length of arguments m, we developed four architectures (i.e. methods) of the multiplier that can be scaled by *any* digit width. Some architectures give better results for smaller digit widths, some are better for the bigger ones. That again allows the designer to choose the architecture that matches design constrains better. Generally, general digit-serial multipliers are of the same quality as the standard ones. In other words, we do not pay much for the added flexibility.

The shifter is scaled by the number of rotations. It shows that only a small number of rotations should be implemented to obtain the best possible results.

We observed that the quality factor of isolated parts of the cryptographic system can be misleading in case when the system contains parts that are scaled differently or cannot be scaled at all (e.g. a control unit). In such cases, the parts must be manipulated differently to achieve the optimum quality factor of the entire system.

Part II

COPACOBANA-Assisted Attacks on GSM Communication

Chapter 9

Introduction to Part II

GSM, *Global System for Mobile communication*, originally *Groupe Spécial Mobile*, is the most widely used system for mobile phone communication. It is estimated that more than 3 billion subscribers all over the world use mobile phones based on this system.

GSM was developed in 1980s for the use in Europe. The GSM standard defines algorithms for authentication as well as for data encryption. The encryption algorithms are denoted as A5/1 and A5/2. The original cipher, A5/1, is used within Europe and in most other countries, while the A5/2 cipher has been developed later — due to the export restriction — for deploying GSM outside Europe.

Initially, the designs of both ciphers were kept secret, however the general design was leaked by Anderson in 1994 [And94]. The designs of both A5/1 and A5/2 were entirely reverse engineered by Briceno in 1999 [BGW99]. Immediately after the reverse engineering the intentionally weaker cipher A5/2 was cryptanalyzed by Goldberg, Wagner and Green [GWG99].

In this work we focus on the stronger GSM cipher A5/1. The cipher has been extensively analyzed [And94, Gol97, Bab95, KS01, PS00, Gol00, BSW01, BS00, BBK03, BBK06]. Previous work done in cryptanalysis of A5/1 we summarize in Section 10.2. To the best of our knowledge, many of the proposed attacks against A5/1 have never been fully implemented and/or they lack from practicability.

Here we present two known-plaintext attacks against A5/1 that we developed and fully implemented. They represent the first real-world implementa-

tions of attacks against A5/1 reported in open literature. To implement both attacks, we used a special-purpose hardware, COPACOBANA, however, the attacks are different.

The first attack belongs to the general group of the brute-force attacks, more precisely the guess-and-determine attacks. We utilized special properties of A5/1 to make the attack very efficient. The approach we used is applicable with some modifications also to some other stream ciphers. The attack reveals the internal state of A5/1 in about 6 hours on average (and about 12 hours at the worst-case). To mount the attack only 64 consecutive bits of a known keystream are required and we do not need any precomputed data. We also propose an optimized version of the attack. Both plain and optimized version of the attack have been fully implemented and tested on our target platform.

The second attack belongs to the general group of the time-memory trade-off attacks. With certain probability it is able to reveal the internal state of the cipher in a matter of minutes. COPACOBANA is used in both the pre-computation phase and the online phase of the attack. The engine generating the time-memory trade-off tables has been tailored and optimized to ideally utilize the properties of FPGA chips used in COPACOBANA. This approach provides very high performance at a relatively low cost. Here proposed design approach can be reused when designing similar attacks against other stream ciphers.

The text is structured as follows: In Chapter 10 we describe the A5/1 cipher, we bring an overview on proposed attacks on A5/1 and we describe the special-purpose computing platform COPACOBANA. In Chapter 11, we describe our guess-and-determine attack implemented in COPACOBANA. Chapters 12 and 13 are dedicated to the time-memory trade-off attack. In Chapter 12, we present an overview on time-memory trade-off and time-memory-data trade-off attacks. Then, in Chapter 13, we describe our implementation of the time-memory-data trade-off attack implemented for COPACOBANA. In Chapter 14, we discuss method for backtracking (reverse clocking) of the A5/1 cipher. Last Chapter 15 summarizes the results and concludes this work.

Chapter 10

Background

10.1 A5/1 Cipher

A5/1 is a synchronous stream cipher. The keystream y produced by A5/1 engine is bitwise added modulo 2 to the plaintext, producing the ciphertext. Communication between the mobile phone and the base transceiver station (BTS) is divided into frames, each being 114 bits long. For each frame, the new keystream is produced. All frames of one phone call share the same session key $K = (k_0, \ldots, k_{63}) \in GF(2)^{64}$, which is exchanged between the mobile phone and the BTS at the beginning, when the phone call is established. The 22-bit initialization vector $IV = (v_0, \ldots, v_{21}) \in GF(2)^{22}$ is unique for each frame of the phone call, however, it is equal to the 22-bit frame number FN which is publicly known.

A5/1 cipher consists of 3 linear feedback shift registers (LFSRs) $R1$, $R2$ and $R3$ being 19, 22 and 23 bits long, as depicted in Figure 10.1. The taps of each register correspond to the coefficients of a primitive polynomial of a given length, hence each LFSR produces a sequence of a maximum period. Most significant bits of all 3 registers are added modulo 2 to produce one bit of a keystream per clock cycle. The registers are clocked irregularly; bits $R1[8]$, $R2[10]$ and $R3[10]$ are used as clocking bits (CBs), where $CB_1 = R1[8]$, $CB_2 = R2[10]$, $CB_3 = R3[10]$. At each clock cycle, majority M of all three clocking bits is calculated, $M = maj(CB_1, CB_2, CB3) = CB_1CB_2 + CB_1CB_3 + CB_2CB_3$. The register Ri is clocked if its associated clocking bit CB_i is equal to the majority of all three clocking bits ($CB_i = M$),

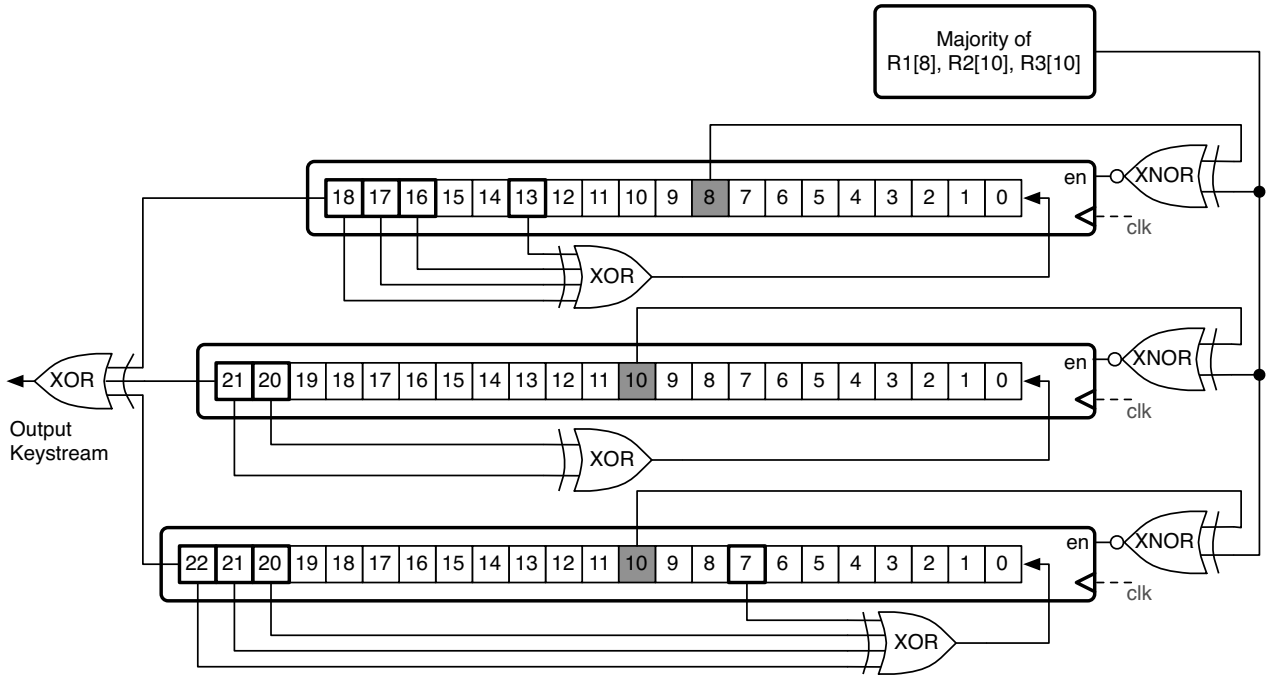

Figure 10.1: A5/1 cipher

otherwise it is stopped. According to Table 10.1, either 2 or 3 registers are clocked at each clock cycle. The probability of clocking a register is equal to $\frac{3}{4}$.

The algorithm of the generation of one keystream is as follows: At the beginning, all 3 registers $R1$, $R2$ and $R3$ are reset. Then the *initialization phase* follows. During the initialization phase the clocking rule is not applied, i.e. all three registers are clocked regularly. In 64 clock cycles, the registers are initialized with a 64 bit key K; in an i-th clock cycle, $0 \leq i \leq 64$, the i-th bit of key is added modulo 2 to the least significant bit of each register and the registers are clocked (shifted to the left by one bit). The key K is followed by a 22 bit wide initialization vector IV which is again bit-by-bit added modulo 2 to the least significant bits of all 3 registers and the registers are clocked.

After initialization phase, the *warm-up phase* follows. During this phase the registers are already clocked irregularly. The cipher is clocked for 100 clock cycles and the output is discarded. After that, during the *executional phase*, 228 bits of keystream are produced, 114 of them are used to encrypt uplink traffic and 114 are used to decrypt downlink traffic. The algorithm of A5/1 is depicted in Figure 10.2.

Table 10.1: Clockcontrol of A5/1

CB of $R1$: $R1[8]$	0	0	0	1	0	1	1	1
CB of $R2$: $R2[10]$	0	0	1	0	1	0	1	1
CB of $R3$: $R3[10]$	0	1	0	0	1	1	0	1
Majority	0	0	0	0	1	1	1	1
Clock $R1$?	√	√	√	–	–	√	√	√
Clock $R2$?	√	√	–	√	√	–	√	√
Clock $R3$?	√	–	√	√	√	√	–	√

1. **(Reset)** Set all 3 registers $R1$, $R2$ and $R2$ to 0
2. **(Initialization)** Load 64 bit key K followed by a 22 bit frame number FN to all 3 registers
   - Registers clocked regularly
   - K and FN are bit-by-bit XORed to the least significant bits of the registers $R1$, $R2$ and $R3$.
3. **(Warm-up)** Clock for 100 clock cycles and discard the output
   - Registers clocked irregularly
4. **(Execution)** Clock for 228 cycles, generate 114+114 bits of keystream (for each direction)
   - Registers clocked irregularly
5. Repeat for the next frame

Figure 10.2: Algorithm of A5/1

10.2 Previous Work

Cryptanalysis of block ciphers is, in most cases, focused on finding the key used for an encryption. The key is also used in case of stream ciphers, however, cryptanalysis of stream ciphers is mostly focused on finding the internal state of the cipher [Gol97, BS00], i.e. the content of the internal registers at certain time of execution. Once the internal state is revealed, the cipher may be run forward, decrypting the remainder of the ciphertext. In some cases the cipher may be also run backwards, which is e.g. the case of A5/1.

During last 15 years the security of A5/1 has been extensively analyzed. Pioneering work in this field was done by Anderson [And94], Golic [Gol97], and Babbage [Bab95].

10.2.1 Guess-and-Determine Attacks

Anderson's basic idea was to guess the complete content of the registers $R1$ and $R2$ and about half of the register $R3$. In this way the clocking of all three registers is determined and the second half of $R3$ can be derived given 64 bits of keystream. In the worst-case each of the 2^{52} determined state candidates (i.e., candidates for S^w) needs to be verified against the keystream which imposes a high workload when done in software.

The hardware-assisted attack by Keller and Seitz [KS01] is based on Anderson's idea. However, they proposed a way to exclude a significant fraction of possible candidates at a very early stage of the verification process. The authors claim that their approach reduces the attack complexity to $2^{41} \cdot (\frac{3}{2})^{11}$ with an expected computing time of 14 clock-cycles per guess. This results in a worst-case complexity of $2^{51.24}$ clock cycles. They implemented the attack on a Xilinx XC4062 FPGA. The FPGA is hosting seven instances of the guessing algorithm and operates at a frequency of 18.65 MHz leading to an attack time of about 236 days. Unfortunately, the approach given in [KS01] does not only immediately discard wrong candidates but a priori *restricts* the search for candidates to a certain subspace. This fact is not explicitly mentioned in the paper. Moreover, no complete analysis of the attack is given. Our analyses in Section 11.1 show that the success probability of their attack is only about 18% and the expected computing time for a guess is slightly higher than the stated one.

The key idea of Golic's attack [Gol97] is to guess the lower half of each register (these bits determine the register clocking in the first few clock-cycles)

and clock the cipher until the guessed bits "run-out". Each output bit immediately yields a linear equation in terms of the internal state bits belonging to the upper halves of three registers. Then we continue guessing the clocking sequence yielding again other linear equations that describe the output of the majority function. Whenever 64 linearly independent equations are obtained in this way the system is solved using Gaussian elimination. The complexity of this attack is $O(2^{40})$ steps. However, each step is fairly complex since it comprises to compute the solution of an 64×64 LSE (and the verification of the corresponding state candidate).

Pornin and Stern proposed a SW/HW tradeoff attack [PS00] that is based on Golic's approach but in contrast to Golic they are guessing the clocking sequence from the very first step, similarly to [Gol00]. These guesses create a tree with 4 branches in each node (each branch represents one clocking combination, cf. Table 10.1). While traversing a path down the tree, three equations are obtained at each node (similarly to the second phase of Golic's method), namely two equations describing the clocking and one equation describing the output. Hence, after n steps (in depth) one collected $3n$ equations. The tradeoff parameter n is chosen such that $3n < 64$. Thus, each path in the tree leads to an underdetermined LSE that is solved in software resulting in a parametric solution on the internal state. The basis of the corresponding linear subspace containing all solutions to such an LSE consists of $(64 - 3n + 1)$ 64-bit vectors. These vectors are sent to the hardware, where a brute force attack is performed, i.e., each of the 2^{64-3n} elements of the subspace is generated and loaded to the A5/1 instance. The instance is run after each load to verify the obtained output keystream against the given keystream. The authors estimated an average running time of 2.5 days when using an XP-1000 Alpha station for the software part and two Pamettes 4010E for the hardware part of the attack (where $n = 18$).

The authors consider to place twelve A5/1 instances into one Xilinx 4010E FPGA, occupying $12 \times 36 = 432$ CLBs out of 576 (75% of the FPGA). Unfortunately, any details (especially the area) of the unit generating 2^{64-3n} internal states are missing which makes it hard to verify the stated figures. However, these figures do not seem to be based on real measurements and we consider them as too optimistic; we expect that the generator unit occupies a relatively large area. For instance, when choosing $n = 18$ the transmitted basis consists of 11 vectors, i.e., $11 \times 64 = 704$ bits. Since the deployed Xilinx 4010E FPGA contains only 1152 flip-flops, more than 60% of them would be used just for holding the coefficients of the basis. So there seems not to be enough space

to place twelve A5/1 units (needing further $12 \times 64 = 768$ flip-flops) on the FPGA as stated in the paper.

10.2.2 Time-Memory-Data Trade-off Attacks

Finally, there is a whole class of time-memory-data tradeoff (TMDTO) attacks on A5/1 which share the common feature that a large amount of known keystream must be available and/or huge amounts of data must be precomputed and stored in order to achieve reasonable success rates and workloads for the online phase of these attacks. Simple forms of such attacks have been independently proposed by Babbage [Bab95] and Golic [Gol97].

Recently, Biryukov, Shamir, and Wagner presented an interesting (non-generic) variant of an TMDTO [BSW01] (see also [BS00]) utilizing a certain property of A5/1 (low sampling resistance). The precomputation phase of this attack exhibits a complexity of 2^{48} and memory requirements of only about 300 GB, where the online phase can be executed within minutes with a success probability of 60%. However, 2 seconds of known keystream (i.e., about 25000 bits) are required to mount the attack making it impractical.

Another important contribution in this field is due to Barkan, Biham, and Keller [BBK03] (see also [BBK06]). They exploit the fact that GSM employs error correction before encryption — which reveals the values of certain linear combinations of stream bits by observing the ciphertext — to mount a ciphertext-only TMDTO. However, in the precomputation phase of such an attack huge amounts of data need to be computed and stored; even more than for known-keystream TMDTOs. For instance, if we assume that 3 minutes of ciphertext (from the GSM SACCH channel) are available in the online phase, one needs to precompute about 50 TB of data to achieve a success probability of about 60% (cf. [BBK06]). There are 2800 contemporary PCs required to perform the precomputation within one year.

These are practical obstacles making actual implementations of such attacks very difficult. In fact, to the best of our knowledge no full implementation of TMDTO attack against A5/1 has been reported yet.

10.3 COPACOBANA — A Cost-Optimized Parallel Code Breaker

Cryptanalysis of modern cryptographic algorithms demands for a high computation power, which can be provided by

- Supercomputers, like IBM BlueGene, Cray or SGI.
- Distributed computing
- Application specific integrated circuits (ASICs)
- Field-programmable gate arrays (FPGAs)

Supercomputers tend to provide sophisticated options for high speed communication and large portions of distributed memory that are mostly not required for simple cryptanalytical number crunching. Unfortunately, the availability of these features increases the costs of theses systems significantly, resulting in a nonoptimal cost-performance ratio of an attack on a cipher.

Distributed computing with loosely coupled processors connected via the Internet is a popular approach, e.g., demonstrated by the SETI@home project [Uni05]. However, this has the disadvantage that the success strongly depends on the number of participating users. Hence, distributed computing usually results in an unpredictable runtime for an attack since the available computational power varies due to the dynamically changing number of contributors.

The high non-recurring engineering costs for ASICs have put most projects for building special-purpose hardware for cryptanalysis out of reach for commercial or research institutions. One of the most notable ASIC-based projects was the DES hardware cracker called *Deep Crack* built in 1998 by Electronic Frontier Foundation (EFF). Running the exhaustive key search it found the DES key within 56 hours [Ele98]. Their DES cracker consisted of $1,536$ custom designed ASIC chips at a cost of material of around US$ 250,000 and could search 88 billion keys per second.

With the improvements in FPGA technology, reconfigurable computing has emerged as a cost-effective alternative for certain supercomputer applications. Unlike standard CPUs the reconfigurable circuits offer much higher computing performance at comparable price. They do not reach the performance of dedicated ASICs, however, their reprogrammability allows for using them for different projects. Also the non-recurring costs are much lower if FPGAs are used rather than special-purpose ASIC is designed. Therefore, special-purpose

cryptanalytical machines have now become a possibility outside government agencies.

The Cost-Optimized Parallel Code Breaker (COPACOBANA) is a high-performance, low-cost cluster consisting of 120 *Xilinx Spartan3-XC3S1000* FPGAs. It has been jointly developed by the Christian Albrechts University of Kiel and the Ruhr University in Bochum in 2006 [KPP+06]. Currently, COPACOBANA appears to be the only reconfigurable parallel FPGA machine optimized for code breaking tasks reported in the open literature. Depending on the actual algorithm, its parallel hardware architecture can outperform conventional computers by several orders of magnitude. COPACOBANA has been designed under the assumption that the typical cryptanalytical task is characterized by following properties:

- computational-costly operations are parallelizable,
- parallel instances have only a very limited need to communicate with each other,
- the demand for data transfers between host and nodes is low due to the fact that computations usually dominate communication requirements and
- typical cryptographic algorithms and their corresponding hardware nodes demand very little local memory which can be provided by the on-chip RAM modules of the FPGA.

These characteristics were utilized in the design of COPACOBANA. To achieve minimum building costs, the architecture based on a common bus interconnecting all FPGAs with a central controller has been used. COPACOBANA also does not contain any external memory modules.

COPACOBANA consists of a controller card, up to 20 DIMM modules, and the backplane, as depicted in a diagram at Figure 10.3. Every custom-made DIMM module hosts six *Xilinx Spartan3-XC3S1000-4FT256* FPGAs and a DC-DC converter providing core voltage for the FPGAs. The backplane hosts 20 DIMM connectors for the DIMM modules and one special connector for the controller. The controller card connects COPACOBANA with a host computer via its interface. Two types of controllers have been developed; the first one is equipped with USB interface, while the second one uses 1 Gbit Ethernet interface. The host computer can either communicate with a single FPGA or broadcast the data to all FPGAs. Besides data transfers, the controller card also allows for the configuration of the FPGAs.

All modules are interconnected with a common 64 bit data bus. The address bus has 16 bits; 5 bits are used to address the DIMM module (combinations 00000 and 11111 are used to address none and all modules, respectively), 6 bits are used to address the FPGA in the DIMM module (one-hot encoding is used) and 5 bits are used to select addresses inside the FPGA. If the Ethernet controller is used, then, due to pin limitations, the address has just 12 bits, leaving only 1 bit to select addresses inside the FPGA.

A clock signal of 20 MHz is generated in the controller card and distributed via the backplane. If a circuit implemented on the FPGA is required to run at a frequency other than 20 MHz, then the frequency is multiplied/divided using a Digital Clock Manager (DCM) inside each FPGA. According to the datasheet [Xil07], the circuits implemented in *Spartan 3* FPGAs can reach an operational frequency of up to 300 MHz, however, our experience shows that maximum frequencies achievable in practical circuits do not exceed 160 MHz.

COPACOBANA is placed into a 3 units high, standard 19" rack module, as shown in photo in Figure 10.4. When configured with a design occupying 97 % of area of each FPGA, running at 100 MHz, and equipped with all 20 DIMM modules, the whole machine has a power consumption of about 600 W.

COPACOBANA has been primarily developed for implementation of brute force attacks. However, it may be used for many other cryptanalytical tasks. For example, en efficient implementation of Pollard-Rho for the discrete logarithm problem over elliptic curves has been proposed in [GPP07]. Besides cryptanalysis, COPACOBANA finds also other application areas, like the massively parallelized DNA motiff search [SWPS08], and others.

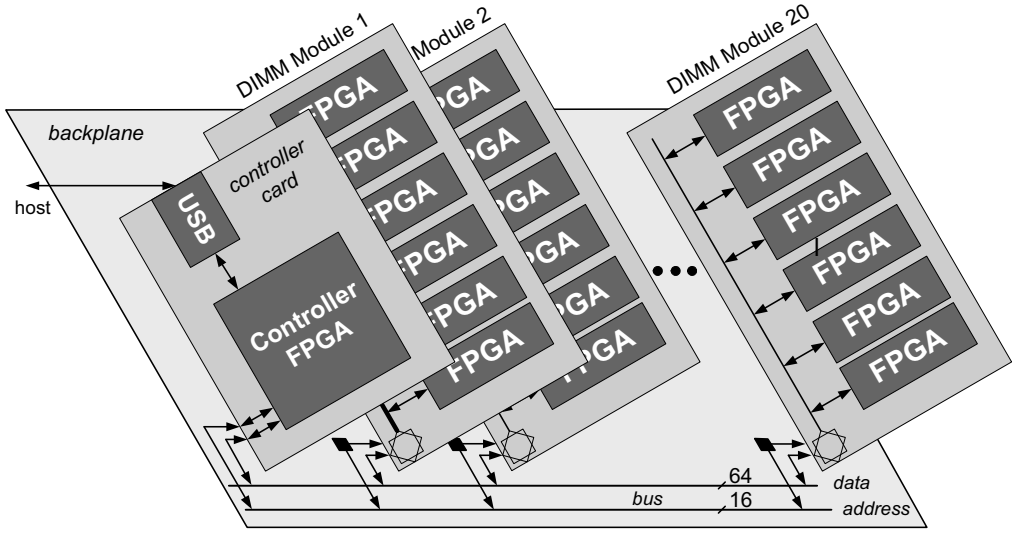

Figure 10.3: Architecture of COPACOBANA.

Figure 10.4: Photo of COPACOBANA. Courtesy of SciEngines GmbH

Chapter 11

Smart Brute-Force Attack on A5/1

As discused in Section 10.2, many of the proposed attacks against A5/1 lack from practicability and/or have never been fully implemented. In contrast to these attacks, we present a real-world attack revealing the internal state of A5/1 in about 6 hours on average (and about 12 hours in the worst-case) using an existing low-cost special-purpose hardware device COPACOBANA. To mount the attack only 64 consecutive bits of a known keystream are required and we do not need any precomputed data. Also the communication requirements with the host computer are relatively small.

On the theoretical side, we present a modification and analysis of the approach sketched in [KS01]. Furthermore, we propose an optimization of the attack implementation leading to an improvement of about 13% in computation time compared to a plain implementation. Both plain and optimized version of the attack have been fully implemented and tested on our target platform.

This has been a collaborative work with Timo Gendrullis and Andy Rupp. This work has been published in [A.11].

11.1 Analysis and Modification of Keller and Seitz's Approach

The approach is based on a simple guess-and-determine attack proposed by R. Anderson in 1994 where the shorter registers $R1$ and $R2$ are guessed and the longer register $R3$ is to be determined. But because Anderson neglected the asynchronous clocking of the registers at first, only the 12 most significant bits of $R3$ can be determined from the known keystream while the remaining bits have to be guessed as well.

Keller and Seitz's attack can be divided into two phases, into the *determination phase* in which a possible state candidate consisting of the three registers of A5/1 after its warm-up phase is generated and into a subsequent *postprocessing phase* in which the state candidate is checked for consistency.

11.1.1 Analysis

In the determination phase, Keller and Seitz try to reduce the complexity of the simple guess-and-determine attack further by early recognizing contradictions that can occur by guessing the clocking bit (CB) of $R3$ such that $R3$ will not be clocked. Therefore, they first completely guess the registers $R1$ and $R2$ and then derive register $R3$ in the following manner. Let $Ri^{(t)}[n]$ denote the n-th bit of register Ri at a time t, where $t = 0$ is immediately after the warm-up phase of A5/1 and increases by 1 every clock-cycle. Then, foremost compute the first most significant bit (MSB) of $R3$, which is $R3^{(0)}[22]$, immediately out of $R1^{(0)}[18]$ and $R2^{(0)}[21]$ and the first bit of the known keystream (KS). Then inspect the clocking bits of registers $R1$ and $R2$, which are $R1^{(0)}[8]$ and $R2^{(0)}[10]$, and guess the first clocking bit of $R3$, namely $R3^{(0)}[10]$. If $R1^{(0)}[8]$ and $R2^{(0)}[10]$ are not equal, $R3$ will be clocked in either way and so both possibilities for $R3^{(0)}[10]$ have to be checked. But if the CBs of $R1$ and $R2$ are identical then at least these two registers will be clocked. Assume now the CB of $R3$ is chosen to be different from the ones of $R1$ and $R2$, i.e., $R3^{(0)}[10] \neq R1^{(0)}[8]$, and as a consequence $R3$ will not be clocked. Now in one half of these cases the generated output bit of the MSBs of all three registers (which are $R1^{(1)}[18] = R1^{(0)}[17]$, $R2^{(1)}[21] = R2^{(0)}[20]$, $R3^{(1)}[22] = R3^{(0)}[22]$) does not match the given keystream bit and a contradiction occurs. As a consequence the CB of $R3$ has to be guessed in a way that $R3$ will be clocked together with $R1$ and $R2$, i.e., the CB of $R3$ is to be chosen equal to the CBs of $R1$ and $R2$, so that a new MSB can be computed.

By early recognizing this possible contradiction while guessing $R3^{(t)}[10]$, all arising states of this contradictory guess neither need to be computed further on nor checked afterwards. To further reduce the complexity of the attack they do not only discard these described wrong possibilities for the CB of $R3$ in case of a contradiction but they also limit the number of choices to the one of not-clocking $R3$ if this is possible without any contradiction. After having computed the first MSB of $R3$ the process of guessing a CB and computing another MSB of $R3$ is repeated until $R3$ is completely determined which is after having clocked $R3$ for 11 times.

This heuristic reduces the number of possibilities for $R3^{(t)}[10]$ in one half of all cases from two to one. The number of possible state candidates to be checked decreases thus from 2^{11} to $(2-\frac{1}{2})^{11} = (\frac{3}{2})^{11} \approx 2^{6.43} \approx 86$ for every fixed guess of registers $R1$ and $R2$ in general. This results in $2^{41} \cdot 2^{6.43} = 2^{47.43}$ possible state candidates. But because they discard some valid states as well as states leading to a contradiction they have only a low success probability. The number of all valid state candidates for one fixed guess of $R1$ and $R2$ is $(2-\frac{1}{4})^{11} = (\frac{7}{4})^{11} \approx 2^{8.88} \approx 471$. Thus, the number of state candidates inspected by Keller and Seitz in proportion to the number of valid state candidates results in a success probability of only $\frac{86}{471} \approx 0.18 = 18\%$.

Immediately after the determination phase, the A5/1 is performed with the generated state candidate in the postprocessing phase and the generated output bits are checked against the remaining bits of the 64 bit known keystream. Keller and Seitz just state that this consistency check in the postprocessing phase will proceed fast and that both, determining a state candidate and checking it against the known keystream, will take $14 \approx 2^{3.81}$ clock-cycles. This leads to a complexity of $2^{47.43} \cdot 2^{3.81} = 2^{51.24}$ clock-cycles. But with this expected amount of clock-cycles they underestimated the time complexity as will be shown in Section 11.1.2.

One instance of Keller and Seitz's guessing algorithm occupies 313 out of the 2304 configurable logic blocks (CLBs) of the XC4062 FPGA. It is hard to estimate how fast the original Keller-Seitz attack would be when implemented on COPACOBANA, since the architecture and the performance of the XC4062 [Xil99] and the Spartan-3 XC3S1000 [Xil07] FPGAs are different. For example, one XC4000 CLB only roughly corresponds to one Spartan-3 slice, because it contains two 4-input look-up tables (LUT), one 3-input LUT and two flip-flops (FF), while a Spartan-3 slice contains only two 4-input LUTs and two FFs. Because the available number of slices on a Spartan-3 XC3S1000 FPGA is 7680 and if we assume that one instance of the guessing algorithm would occupy 313 slices, a maximum number of 24 instances could be implemented

on one FPGA. This leaves just 168 slices for other circuits for controlling the instances. According to the datasheets the "internal performance of XC4000 family chips can exceed 150 MHz" while the "maximum toggle frequency of Spartan-3 chips is 630 MHz". That represents a performance ratio of less than 4.2. Out of these figures we estimate that the attack would not be faster than $\frac{24}{7} \times 4.2 \times 120 = 1728$ times when run on COPACOBANA. This yields to a minimum of 3.27 hours to perform the search of Keller and Seitz. But if we recall again that (i) the attack searches only through 18% of the valid states, the search through all valid states would take at least 18.19 hours, (ii) the number of guessing instances implemented in one FPGA would be less than 24 since at least an additional control logic has to be implemented, and (iii) Keller and Seitz underestimate the time complexity as will be shown in Section 11.1.2, the computation time is expected to increase significantly.

11.1.2 A Slight Modification

Our algorithm is similar to the one proposed by Keller and Seitz except that we only discard wrong possibilities for $R3^{(t)}[10]$ that would immediately lead into a contradiction. But if no contradiction appears we still check both possibilities for $R3^{(t)}[10]$, which means clocking and not-clocking $R3$. Because of this, we take every possible state candidate into account and therefore will find unlike Keller and Seitz the correct state candidate in any case. This reduces only in $\frac{1}{4}$ of all cases the number of choices from two to one and, hence, the expected number of possibilities for $R3$ that need to be checked is approximately 471 for every fixed guess of registers $R1$ and $R2$ (cf. Section 11.1.1).

A flowchart of the decisions during the determination phase and the post-processing phase shows Figure 11.1. A more detailed overview of a highlighted block, i.e. how $R3^{(t)}[10]$ is guessed and how certain subtrees are discarded, is given in Figure 11.2.

Example 11.1. An example for the first steps of the reduction of possibilities performed by the algorithm is given in Figure 11.3. It shows next to the first 4 bits of a known keystream the first 4 MSBs and the first 3 CBs of the guessed registers $R1$ and $R2$ and of the derived register $R3$. The algorithm proceeds as follows.

1. Compute $R3^{(0)}[22] = R1^{(0)}[18] \oplus R2^{(0)}[21] \oplus KS[0] = 0$.
2. $R1^{(0)}[8] \neq R2^{(0)}[10]$: Choose $R3^{(0)}[10] = 0 \neq R1^{(0)}[8]$ first and clock registers $R2$ and $R3$.

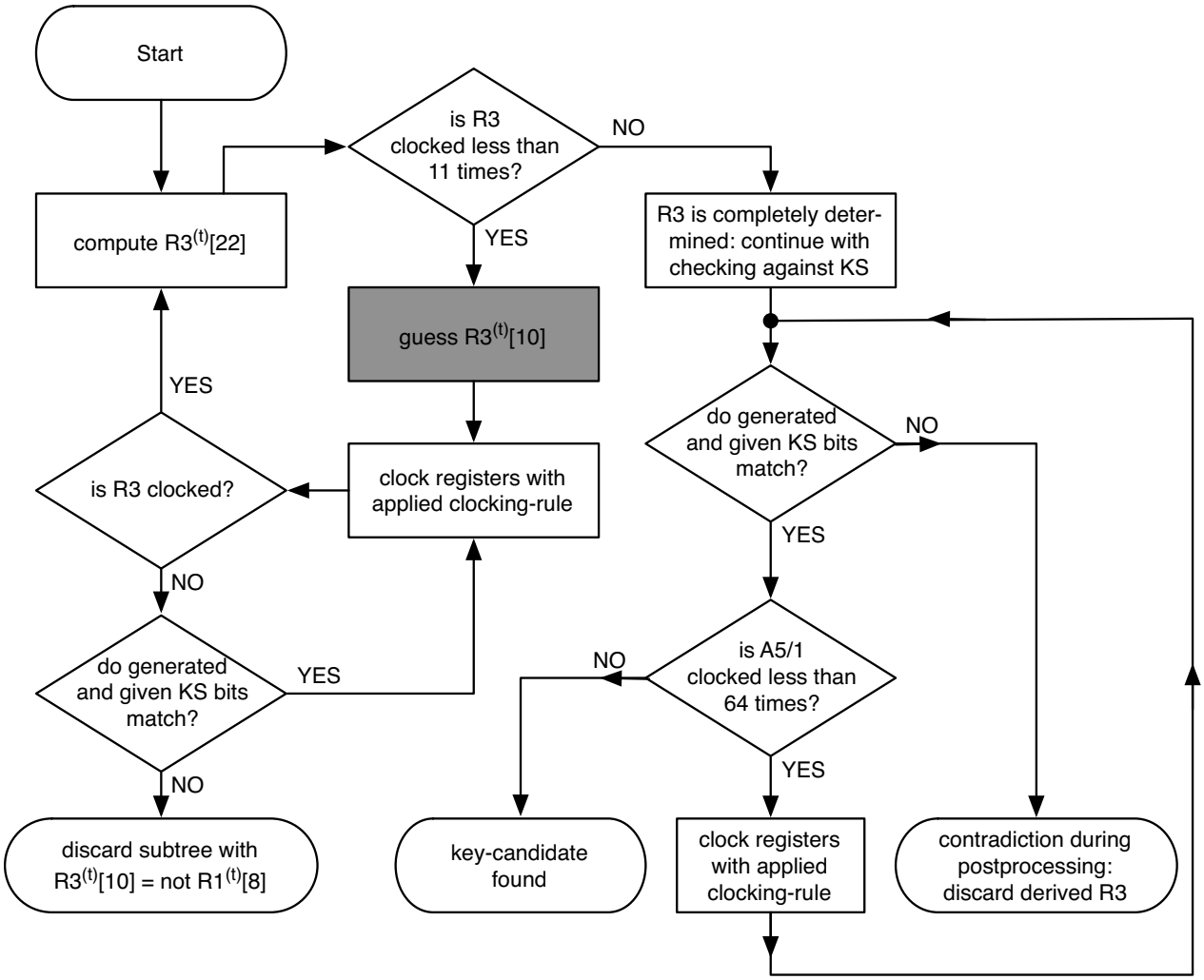

Figure 11.1: Flowchart of the FSM of a guessing-engine

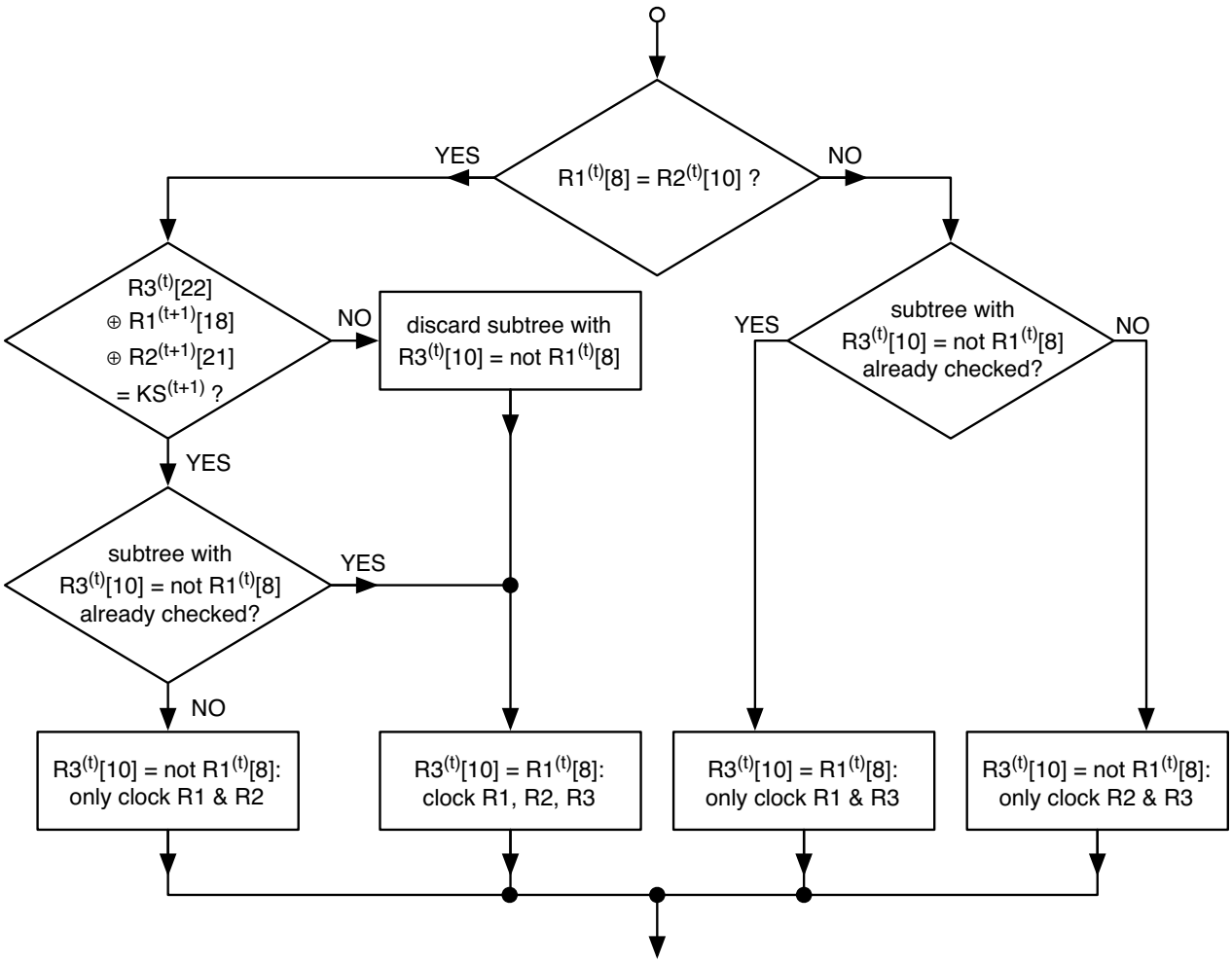

Figure 11.2: Guessing the clocking bit of $R3$ in detail

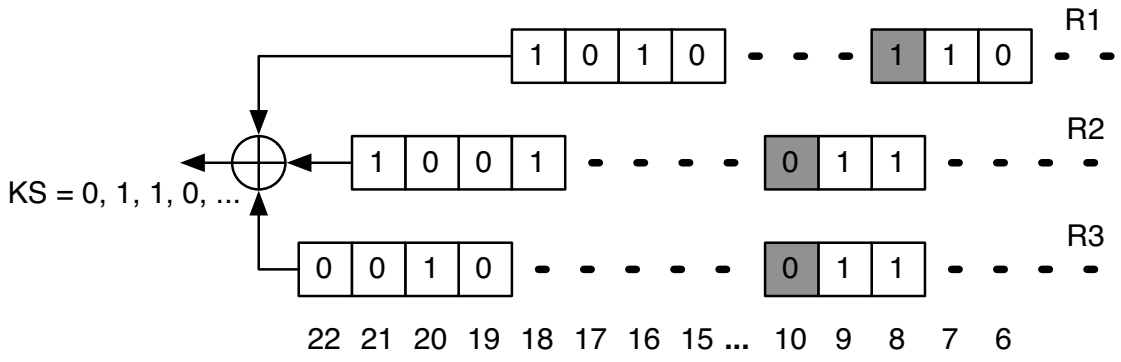

Figure 11.3: An example for a generated state candidate after 3 times guessing $R3^{(t)}[10]$

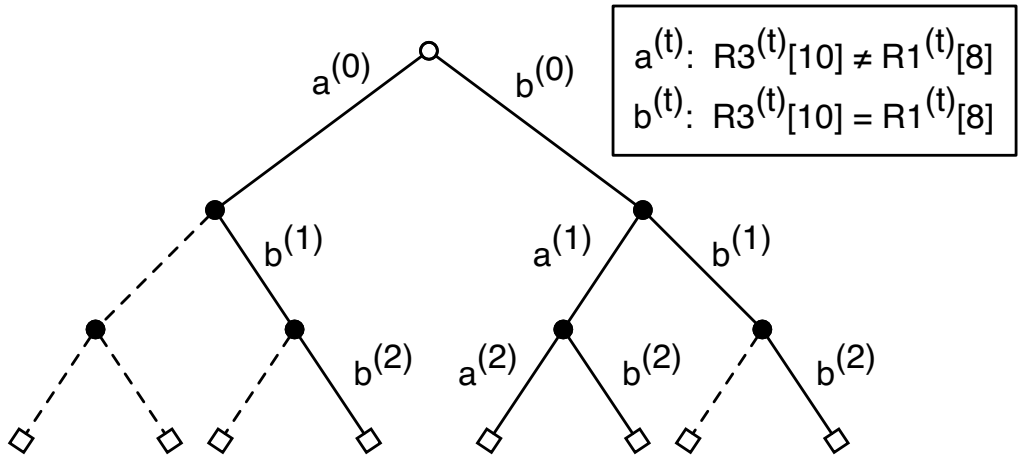

Figure 11.4: An example for a reduced binary decision tree of $R3^{(t)}[10]$

3. Compute $R3^{(1)}[22] = R3^{(0)}[21] = R1^{(0)}[18] \oplus R2^{(0)}[20] \oplus KS[1] = 0$.
4. $R1^{(0)}[8] = R2^{(0)}[9]$: Not clocking register $R3$ would result in a contradiction because $R1^{(0)}[17] \oplus R2^{(0)}[19] \oplus R3^{(0)}[21] \neq KS[2]$. Hence, discard the possibility $R3^{(1)}[10] = 0 = R3^{(0)}[9] \neq R1^{(1)}[8]$, instead choose $R3^{(1)}[10] = 1 = R3^{(0)}[9] = R1^{(0)}[8]$, and clock all registers $R1$, $R2$, $R3$.
5. Compute $R3^{(2)}[22] = R3^{(0)}[20] = R1^{(0)}[17] \oplus R2^{(0)}[19] \oplus KS[2] = 1$.
6. ...

The example ends here because it is apparent from Figure 11.4, which shows the binary decision tree for $R3^{(t)}[10]$ up to a depth of 3, that discarding possibilities for $R3^{(t)}[10]$ results in cutting whole subtrees. In the example above we chose edge $a^{(0)} = R3^{(0)}[10] = 0 \neq R1^{(1)}[8]$ at the root node first and then discarded the possibility $a^{(1)} = R3^{(1)}[10] = 0 \neq R1^{(1)}[8]$ at the corresponding node of depth 1.

Time Complexity of the Attack. Generating one possible state candidate during determination phase takes one clock-cycle for deriving $R3^{(0)}[22]$ and then eleven times clocking register $R3$ to determine the remaining MSBs of the register. With a probability of $P_{clk} = \frac{3}{4}$ for clocking a register of A5/1 it takes an expected number of $1 + \frac{4}{3} \cdot 11 = 15\frac{2}{3}$ clock-cycles to generate the state candidate for fixed registers $R1$ and $R2$ and the known keystream. Because every clock-cycle one bit of the known keystream is inspected, the expected number of needed known keystream bits to generate a state candidate corresponds to the number of clock-cycles needed for this process.

After having generated one state candidate it needs to be checked in the postprocessing phase further on against the remaining bits of the known keystream. To be able to perform this check immediately after the determination phase we additionally compute the feedback bits of register $R3$ with its linear feedback function. We start with this computation from the time when $R3^{(3)}[10] = R3^{(0)}[7]$ is guessed. So we already computed 8 of the 11 feedback bits of $R3$ when the state candidate is generated. The remaining 3 feedback bits are computed in parallel and we continue with performing A5/1. Now, the produced output is compared to the known keystream. A contradiction between the generated output and a known keystream bit is expected to occur with a probability of $\alpha = \frac{1}{2}$ in the first clock-cycle of postprocessing. Every cycle the algorithm is clocked further on, the probability of a contradiction is again $\frac{1}{2}$. Generally speaking, it is $\alpha_n = \frac{1}{2^n}$ for the n-th cycle after the determination phase and the algorithm will clock on with an expected value of $\frac{1}{\alpha} = 2$ further needed clock-cycles to inspect the output. If it is clocked without any contradiction up to the 64-th bit of the known keystream we found a valid state candidate for reconstructing the session key.

So, we get an expected number of $T = 15\frac{2}{3} + 2 = 17\frac{2}{3}$ clock-cycles to determine a state candidate and check it for consistency with the given keystream instead of just 14 clock-cycles as stated by Keller and Seitz. Thus, the time complexity of our whole attack is $C \approx 2^{41} \cdot (\frac{7}{4})^{11} \cdot 17\frac{2}{3} \approx 2^{54.02}$.

11.2 Hardware Architecture for COPACOBANA

This section presents an efficient implementation of a *guessing-engine* in hardware which performs the determination phase and the postprocessing phase of the attack. On every FPGA, several instances of this guessing-engine will be implemented. Therefore, we will additionally introduce a hardware-software-interface controlling these instances and providing intercommunication.

11.2.1 The Guessing-Engine

Figure 11.5 shows an overview of the guessing-engine with its different components. A large part of the architecture for implementing this guessing-engine consists of flip-flops (FFs) for storing the content of different registers. This is in detail the *state candidate register*, storing the computed register $R3$ and

the fixed guess of registers $R1$ and $R2$ in 64 bits. Additionally, we need FFs to store the 64 bits of known keystream and an additional simple shift register to evaluate a different known keystream bit every clock-cycle. To perform the consistency check in the postprocessing phase, all three *A5/1 LFSRs* have to be implemented, too. But the most important part of this architecture is the finite state machine (*FSM*) performing the determination phase and the postprocessing phase. Its functionality was already presented in Figures 11.1 and 11.2. The shown process is repeated until all possible state candidates, i.e., the whole binary decision tree of $R3^{(t)}[10]$, for one fixed guess of registers $R1$ and $R2$ have been checked. The fact, that the guess $R3^{(t)}[10] \neq R1^{(t)}[8]$ is always checked first corresponds to the binary decision tree of Figure 11.4. This binary decision tree storing the discarded or already checked possibilities is mapped into the *branching state register*.

The most straightforward way of mapping such a binary decision tree with a certain height h into hardware, is to use an h-bit wide binary counter. In our case all leaves are at a depth of $d = h = 11$. Turning left at a node of the tree, i.e., $R3^{(t)}[10] \neq R1^{(t)}[8]$, is represented by 0 in the corresponding counter bit and turning right at a node, i.e., $R3^{(t)}[10] = R1^{(t)}[8]$, is represented by 1. Now, to reach all leaves from the leftmost unto the rightmost one by one, we initialize the 11-bit wide counter to all 0 and read it in 11 clock-cycles bit by bit from the most significant bit (MSB) to the least significant bit (LSB). When having reached the leftmost leaf in such a manner, we increase the register by one and restart reading bit by bit at the MSB again. This will lead us to the second leaf from the left. To reach the rest of the leaves we count through this 11-bit wide register up to all bits being 1. Now it is claimed by the attack that certain subtrees of the binary decision tree are discarded (cf. Section 11.1.2). To be able to do that while passing through the tree, we have to set the corresponding bits of the 11-bit wide counter manually to 1 with an 1-to-11 bit demultiplexer. The FSM does this with bit number b every time a contradiction is detected at a node of depth $d = b + 1$ and a possibility of $R3^{(t)}[10]$ is discarded. This results in the reduced number of leaves of the binary decision tree of $(\frac{7}{4})^{11} \approx 471$ meaning the amount of possible state candidates for a fixed guess of $R1$ and $R2$.

11.2.2 The Control-Interface

Because several instances of the guessing-engine are implemented on one FPGA they need to be controlled continuously. This is done by the *control-interface* and there is exactly one instance of it implemented on each FPGA of

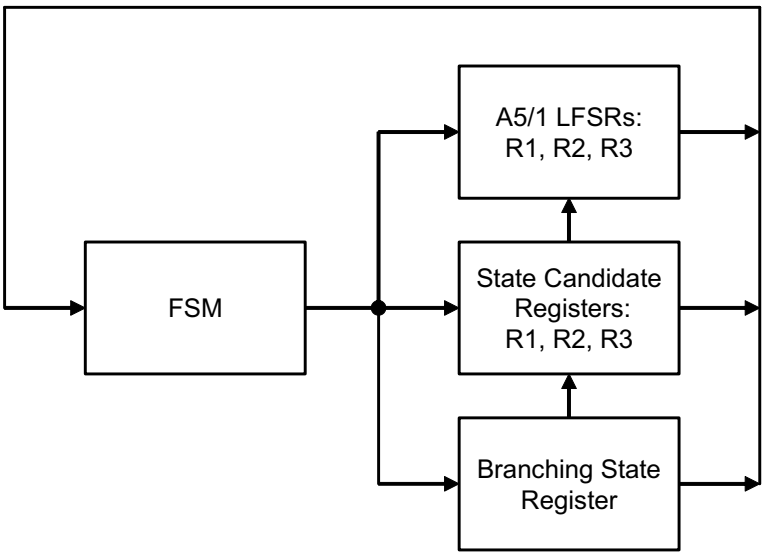

Figure 11.5: An overview of the guessing-engine

COPACOBANA. It accepts the 64 bit known keystream and a *sub-searchspace* which has to be searched by the FPGA. By sub-searchspace we mean a certain amount of fixed guesses for registers $R1$ and $R2$. Therefore, a software divides the *searchspace* consisting of the 2^{41} possibilities into these sub-searchspaces and transmits to each FPGA another one of them together with the known keystream. The control-interface of the FPGA then counts through this sub-searchspace and provides each guessing-engine with a fixed guess of registers $R1$ and $R2$ to be searched. Every time a guessing-engine finishes its search it sends a report to the control-interface whether it was successful or not on finding a state candidate and requests for another fixed guess of registers $R1$ and $R2$ out of the current sub-searchspace. In case of success the valid state candidate is propagated to the software. This is repeated until the whole sub-searchspace is searched by the FPGA. During the search, the software retrieves regularly at reasonable intervals the status information of each FPGA and assigns a new sub-searchspace to an FPGA if requested. The search is finished when all state candidates that can be generated with the 2^{41} possibilities for guessing $R1$ and $R2$, i.e., the whole searchspace, are checked for consistency.

11.2.3 Optimization: Storing Intermediate States

When completely passing through a binary decision tree, edges near the root node are traversed much more often than edges near the leaf nodes. The number of cycles $R3$ needs to be clocked to reach any leaf of the tree is 11 (cf.

Section 11.2.1). For example, when inspecting the two leftmost leaves we have to go bit by bit through the states 00000000000 and 00000000001 of the 11-bit wide counter corresponding to the tree. Apparently, the first ten edges up to the node of depth 10 for both leaves are identical. Therefore, we can create *recovery points* at some depth in the search tree. More precisely, it is possible to store the intermediate state (i.e., the content of all A5/1 registers) at such a point (node of tree) and search the subtree starting at this recovery point instead of starting at the root node. This apparently demands a larger area, but saves a certain amount of clock-cycles.

Let us assume that reloading takes exactly one clock-cycle. If we store and reload the intermediate states at depth $d = 10$, then the number of clock-cycles for $R3$ reduces from 11 to $\frac{11+1+1}{2} = 6.5$ on average: 11 times clocking $R3$ to reach the first leaf, one clock-cycle reloading the intermediate state, and one time clocking $R3$ to reach the next leaf from the reloaded state. If we store the intermediate states at depth $d = 9$, the corresponding subtree has 4 leaves. To reach the leftmost one takes 11 clock-cycles, but to reach the other 3 leaves will take just $1 + 2 = 3$ clock-cycles each. Therefore, the average number of times $R3$ needs to be clocked is in this case only $\frac{11+3+3+3}{4} = \frac{8+3\cdot 4}{4} = 5$.

Generalizing this approach of storing and reloading intermediate states at a depth of $d = 10$ or $d = 9$ to a depth of $d = b + 1$, where b denotes the number of the bit in the 11-bit wide counter consecutively numbered from 0 to 10, we need to clock $R3$

$$f(b) = \frac{b + (11 - b) \cdot 2^{(10-b)}}{2^{(10-b)}} \tag{11.1}$$

times on average to reach one leaf. The function has a minimum of 4.875 times clocking $R3$ on average to reach a leaf for storing and reloading intermediate states at a depth of $b_{min} = 7$ for $b \in \mathbb{N}$.

Taking also into account that some subtrees are discarded while passing through the tree (cf. Section 11.1.2) and the number of possibilities is reduced from 2 to $\frac{7}{4}$ for every guess, the function needs to be adapted:

$$g(b) = \frac{b + (11 - b) \cdot (\frac{7}{4})^{(10-b)}}{(\frac{7}{4})^{(10-b)}}. \tag{11.2}$$

Both functions $f(b)$ and $g(b)$ are shown in Figure 11.6. The value for the minimum of the function $g(b)$ now changes to approximately 5.31 at $b_{min} = 7$

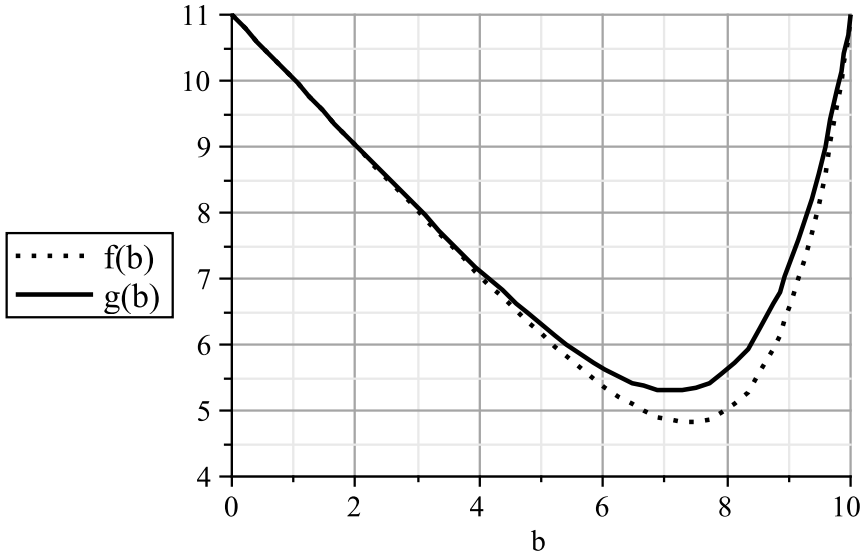

Figure 11.6: Functions $f(b), g(b)$: The average number of cycles clocking $R3$ to generate a state candidate with reloading intermediate states at recovery position b

for $b \in \mathbb{N}$. Therefore, the expected number of clock-cycles for generating and checking one state candidate is now

$$T_{opt} = 1 + \frac{4}{3} \cdot 5.31 + 2 \approx 10.10 \approx 2^{3.33}$$

instead of $T = 17\frac{2}{3}$ (cf. Section 11.1.2). This results in an optimized time complexity of

$$C_{opt} \approx 2^{41} \cdot 2^{8.88} \cdot 2^{3.33} \approx 2^{53.21}$$

and reduces the previous complexity of $C \approx 2^{54.02}$ by 0.81 bit. But when comparing the time complexities of the standard and the optimized guessing-engine we additionally have to take the required area into account. The optimized guessing-engine is expected to occupy a larger area because of the storing elements for intermediate states of several registers. Hence, we will be able to place less instances on one FPGA. This comparison of time-area products is done after the implementation process and will be discussed in Section 11.3.

Table 11.1: Implementation results for the control-interface and the guessing-engines

	slices	**flip-flops**	**look-up tables**	$\mathbf{f_{max}}$ **[MHz]**
control-interface	371	304	254	123.19
standard guessing-engine	202	179	256	112.84
optimized guessing-engine	311	312	412	115.01

11.3 Implementation Results for COPACOBANA

We used Xilinx ISE Foundation 9.2i to synthesize and implement all components for a Xilinx Spartan3-XC3S1000-FT256 FPGA used in COPACOBANA. The simulation of the hardware model was done in MentorGraphics ModelSim SE 6.3d.

First, we implemented and tested one single instance of the standard and optimized guessing engine together with the control-interface for one instance. Therefore, Table 11.1 shows the post place & route results of the implementation process for a single instance of the control-interface and both guessing-engines.

To decide whether it is worth or not implementing the optimized guessing-engine in spite of the increased area consumption we calculated the *time-area product*. Table 11.2 shows a comparison of the computing time T and T_{opt} in *clock-cycles* (cf. Sections 11.1.2 and 11.2.3), the number of slices needed, and the time-area product in *clock-cycles·slices* for our standard and optimized implementation of the guessing-engine. The last row shows the quotient of the values of both designs. The quotient of the time-area products shows an overall improvement of about 12% for one single optimized guessing-engine compared to the standard one. We omitted considering the operating frequencies in the time-area product because both implementations run at nearly the same speed.

After having tested a single instance of each guessing-engine together with the control-interface on one of the *Spartan3-XC3S1000* FPGAs we attempted to maximize the utilization ratio of the available hardware resources. For this purpose, we implemented as many instances as possible of both types of

Table 11.2: Comparison of the implementation results of both guessing-engines

	computing-time [clock-cycles]	**slices**	**time-area product [clock-cycles × slices]**
optimized	10.10	311	3,141.10
standard	17.67	202	3,568.73
$\frac{\textbf{optimized}}{\textbf{standard}}$	0.57	1.54	0.88

Table 11.3: Implementation results of the maximally utilized designs

	slices	**FFs**	**LUTs**	$\mathbf{f_{max}}$ [MHz]	$\mathbf{f_{test}}$ [MHz]
1 control-engine &					
∘ **36 standard**	6,953 (91 %)	10,730	10,576	81.85	72.00
∘ **32 standard**	6,614 (86 %)	9,636	9,417	102.42	92.00
∘ **23 optimized**	7,494 (98 %)	10,141	10,562	104.65	92.00
guessing-engines					
Spartan3-1000	7,680 (100 %)	15,360	15,360	300.00	—

guessing-engines with one instance of the control-interface. We were able to place & route 36 instances of the standard engine on one of the target FPGAs. However, the complexity of the control-interface grows with the number of guessing-engines. For 36 such engines the critical path was transfered to the control-interface creating the bottle-neck of the design. Therefore, the achieved maximum frequency of 81.13 MHz was relatively low. So we decided to implement less engines at a higher frequency instead. The best trade-off for the standard guessing-engine was to implement 32 instances at a maximum frequency of 102.42 MHz. In case of the optimized guessing-engine we were able to implement 23 instances running at 104.65 MHz. The implementation results of both complete designs are shown in Table 11.3. Additionally, the available resources of one FPGA are listed, too.

Table 11.3 also shows the frequencies the designs were tested with. Thus, we can calculate a preliminary estimation of the computation time to determine and check all possible state candidates. For the slow design with the standard

guessing-engine and a time complexity of $C = 2^{54.02}$ (cf. Section 11.1.2) we expect a computation time of

$$t_{est} = \frac{2^{54.02}}{120 \cdot 36 \cdot 72 \cdot 10^6} \cdot \frac{1}{3600}\,\text{h} \approx 16.31\,\text{h}.$$

This is an estimation for a fully equipped COPACOBANA with 120 FPGAs. In accordance to the previous calculation, the preliminary estimation of the computation time for the smaller but faster standard design (32 instances @ 92 MHz) is $t'_{est} \approx 14.36\,\text{h}$. For the optimized guessing-engine (23 optimized instances @ 92 MHz) with a time complexity of $C_{opt} = 2^{53.21}$ we expect a computation time of $t''_{est} \approx 11.40\,\text{h}$.

Time measurements of several extended test runs on COPACOBANA showed an average computation time of $t' = 13.58\,\text{h}$ for the small and fast standard design to perform a complete search for a given 64 bit known keystream. Comparing this result to the estimation of the computing time t'_{est} shows that the complexity differs only by 0.08 bit from our measurements. The optimized design took an average computation time of $t'' = 11.78\,\text{h}$ for a full search. This equals a variation of only 0.05 bit between the estimated and the measured computation time. Because these were the computation times for a full search (i.e., the worst case) the expected average time for finding the valid state candidate is 6.79 h for the standard design and 5.89 h for the optimized design, respectively.

11.4 Summary

In this Chapter we presented a guess-and-determine attack on the A5/1 stream cipher running on the special-purpose hardware device COPACOBANA. It reveals the internal state of the cipher in less than 6 hours on average demanding only 64 bits of known keystream. We like to stress that our attack is also very attractive with regard to financial costs which is a significant factor for the practicability of an attack: The acquisition costs for COPACOBANA are about 10,000 €. Since COPACOBANA has a maximum power consumption of only 600 W, the attack also features very low operational costs. For instance, assuming 10 cent per kWh the operational costs of an attack are only 36 cents.

We like to note that we just provided a machine efficiently solving the problem of recovering a state of A5/1 after warm-up given 64 bits of known keystream. To break the GSM communication, the cipher needs to be tracked back to evaluate the internal state in which the cipher appeared just after

loading the session key K. Albeit, this backtracking can be done efficiently and in a fraction of time on almost any platform, as discussed in Chapter 14.

Further technical difficulties will certainly appear when it actually comes to eavesdropping GSM calls. This is due to the frequency hopping method applied by GSM which makes it difficult to synchronize a receiver to the desired signal. Also the problem of obtaining known plaintext is still under discussion in pertinent news groups and does not seem to be fully solved. However, these are just some technical difficulties that certainly cannot be considered serious barriers for breaking GSM.

Chapter 12

Time-Memory Trade-off Attacks

The method of a time-memory trade-off (TMTO) attack was introduced by Martin Hellman in 1980 [Hel80]. The goal of the method was to reduce the time necessary to break the cipher, which is particularly important if such an attack is repeated frequently. The attack is accelerated by certain data which are precomputed in advance and stored in tables (denoted as TMTO tables).

The method was originally invented for block ciphers, namely for DES. Since then some improvements and modifications have been proposed. To save the number of memory accesses during the attack, which become the bottleneck of the method, Rivest [D. 82] proposed a modification based on so-called *distinguished points*. Biryukov and Shamir [BS00] combined the Hellman's method with an attack on stream ciphers developed by Babbage [Bab95] and Golic [Gol97]. The result of this combination is a *time-memory-data trade-off* (TMDTO) method, applicable particularly to stream ciphers. In 2003 Oechslin [Oec03] introduced so-called *rainbow tables* which are currently considered to be the most efficient variant for block ciphers. However, this variant is inferior to the original Hellman's approach in case of multiple input data (which is the case of stream ciphers). Recently, Barkan, Biham and Shamir [BBS06] presented a modification of rainbow tables, called the *thin-rainbow tables*. The thin-rainbow tables enable efficient implementation of the attack even in the case of multiple input data.

In the following text we bring an overview of the above mentioned methods. We follow and extend the summary introduced in [Rup08]. We start from the description of the Hellman's approach targeted to block ciphers, followed by the description of the Rivest's variant. Then we introduce Biryukov's and Shamir's idea of adopting the method to stream ciphers and finally we conclude with the rainbow tables and thin-rainbow tables methods. The discussion on the method which was finally selected for an attack on A5/1 and was implemented in COPACOBANA is presented in Chapter 13.

12.1 Original Hellman's Approach

Let $P \in \mathbb{P}$ be the plaintext, $k \in \mathbb{K}$ be the key, and $C \in \mathbb{C}$ be the ciphertext. The encryption function is a one-way function

$$E : \mathbb{P} \times \mathbb{K} \mapsto \mathbb{C}.$$

We denote

$$C = E(k, P) = E_k(P). \tag{12.1}$$

A common problem in cryptanalysis is to invert the encryption function E, i.e. for the given ciphertext C, and known corresponding plaintext P, to find a key k, s.t. Equation 12.1 holds.

Let

$$N = |\mathbb{K}|$$

be the cardinality of the search space.

Brute-Force Attack. One approach to invert the encryption function is to perform the brute-force attack, i.e. to encrypt the known plaintext P with all potential values of the key k until the match between the result of encryption and the given ciphertext C is obtained. This approach is (repeatedly) applicable only for ciphers with relatively short key sizes, since it demands a long computation time $T = N$. For example, a brute force attack on DES (which has a key being 56 bits long) takes approx. one week on average when run on COPACOBANA [KPP+06].

Table Look-Up. It is also possible to select certain plaintext P_0 which is likely to appear in the encrypted data. Such a plaintext can be e.g. the header of a file with a certain format, the sequence of spaces in the text files,

$$k_0 \xrightarrow{E} C_0 \xrightarrow{R} k_1 \xrightarrow{E} C_1 \xrightarrow{R} k_2 \xrightarrow{E} C_2 \xrightarrow{R} k_3 \ldots k_{t-1} \xrightarrow{E} C_{t-1} \xrightarrow{R} k_t$$

Figure 12.1: One chain in the Hellman TMTO

etc. For a selected fixed plaintext P_0 all pairs key-ciphertext $\{k_i, C_i\}$ are then precomputed and stored in the memory, sorted by the ciphertext. When performing the actual attack, the ciphertext is looked up in the memory and the key is simply retrieved. The precomputation phase (the *offline phase*) demands a high computation power, but this phase is run just once. The attack itself (also called the *online phase*) is then very fast, $T = 1$. However, this approach has an extreme space (memory) complexity, $M = N$. For example, in case of DES, the table would occupy $2^{56} \times (64 + 64)$ bits, which is 2^{60} bytes, or 2^{20} TB.

12.1.1 Basic Idea

Hellman's original idea [Hel80] was to make a compromise between the above two extreme approaches. The cryptanalysis is again divided into two phases, the *offline* phase, and the *online* phase. During the offline phase a huge amount of calculations is performed, but only some results are stored in the memory (tape, disk, ...). This phase may last weeks, months or even years. During the online phase, i.e. the actual attack, the encryption function is inverted in a relatively short time with the help of the results precomputed and stored in the memory.

12.1.2 Offline Phase

In the offline phase, one tries to precompute (ideally all) pairs *key-ciphertext* $\{k_i, C_i\}$. To reduce memory requirements, the pairs are organized in chains — ciphertext C_i, which is the product of the encryption P_0 under the key k_i, is used to generate the key k_{i+1} for the next encryption, as shown in Figure 12.1. The encryption function E might be considered as a pseudo-random function. Consequently, a pseudo-random walk in a key space $\mathbb{K}$ is performed.

The key k_{i+1} is generated from the ciphertext C_i using so-called *reduction and re-randomization function* R. This function modifies C_i e.g. by permuting some bits (if the key and the ciphertext are of the same length) and/or omits some bits of C_i (if the key is shorter than the ciphertext). The composition of

$$
\begin{aligned}
SP_1 = k_{10} &\xrightarrow{f} k_{11} \xrightarrow{f} k_{12} \xrightarrow{f} \cdots \xrightarrow{f} k_{1,t-1} \xrightarrow{f} k_{1,t} = EP_1 \\
SP_2 = k_{20} &\xrightarrow{f} k_{21} \xrightarrow{f} k_{22} \xrightarrow{f} \cdots \xrightarrow{f} k_{2,t-1} \xrightarrow{f} k_{2,t} = EP_2 \\
SP_3 = k_{30} &\xrightarrow{f} k_{31} \xrightarrow{f} k_{32} \xrightarrow{f} \cdots \xrightarrow{f} k_{3,t-1} \xrightarrow{f} k_{3,t} = EP_3 \\
&\vdots \\
SP_m = k_{m0} &\xrightarrow{f} k_{m1} \xrightarrow{f} k_{m2} \xrightarrow{f} \cdots \xrightarrow{f} k_{m,t-1} \xrightarrow{f} k_{m,t} = EP_m
\end{aligned}
$$

Figure 12.2: Time-memory trade-off table

an encryption function E and a re-randomization function R forms so-called *step function* f:

$$f(k) = R(E_k(P_0)).$$

It also holds

$$k_{i+1} = f(k_i). \tag{12.2}$$

As a part of the precomputation phase, m unique keys $k_{10}, k_{20}, \ldots, k_{m0} \in \mathbb{K}$ are chosen. Those keys serve as start points $SP_1, SP_2, \ldots, SP_m$ of chains as shown in Figure 12.2. Then, the step function f is t-times iteratively applied to each start point SP_j, generating t keys k_{ji} in each chain. The last generated element of each chain is called the end point EP_j:

$$EP_j = f^t(SP_j).$$

The intermediate results are discarded and only the pairs $\{SP_j, EP_j\}$ are stored, sorted by the end points. With the knowledge of the start point and the end point, the whole chain can be reconstructed later.

12.1.3 Online Phase

In the online phase, the cryptanalyst is given the ciphertext C_0. The task is to find a key K s.t.

$$C_0 = E_K(P_0).$$

To do so, first the re-randomization and reduction function R is applied to C_0 and the result

$$y_1 = R(C_0) = R(E_K(P_0)) = f(K)$$

is compared to all end points. If a match is found, $y_1 = EP_j$, then either, according to Equation 12.2, the searched key is $K = k_{j,t-1}$, or the end point EP_j has multiple inverse images (this case is called a *false alarm*). To restore the value of $k_{j,t-1}$, the chain is recomputed, i.e. the step function f is applied $(t-1)$-times starting from SP_j.

If no match is found or if a false alarm appears, then a new result

$$y_2 = f(y_1) = f(R(C_0))$$

is calculated and compared to all end points again. If a match is found now, then the key $k_{j,t-2} = f^{t-2}(SP_j)$ is recalculated and its validity is verified (e.g. by encrypting P_0 with $k_{j,t-2}$ and comparing the result with C_0). This approach is iteratively applied, i.e. the new results $y_i = f(y_{i-1})$ are successively calculated and compared to all end points until either valid match is found or until $i \geq t$. If the match is found in the i-th step, $y_i = EP_j$, then the candidate for the key K is $k_{j,t-i} = f^{t-i}(SP_j)$.

12.1.4 Characteristics

In many real examples, the step function f is non-injective. This is caused either by non-injectivity of the re-randomization and reduction function R (e.g. in case of DES, the 64-bit ciphertext is reduced to the 56-bit key) and/or by non-injectivity of the encryption function E itself (e.g. in case of A5/1 several distinct internal states can produce the same keystream). The non-injectivity of f leads to multiple occurrences of some keys k_{ji} in the table shown in Figure 12.2. However, even if the step function f were injective, then there is still certain probability that some generated key k_{ji} matches a start point SP.

Multiple occurrences of some keys lead to *chain merges* and *chain loops*. The chain merge appears if a duplicate key is generated in two distinct chains. From that moment on, both chains continue with generating the same sequence of keys. The chain loop appears if some key is generated twice in the same chain. Starting from the duplicate key, the chain repeats the sequence of already generated keys.

Both chain merges and chain loops cause reduced coverage of the key space $\mathbb{K}$. Hellman proved that it is not worth increasing the number of generated chains m, or the length of chains t, beyond the point, where

$$mt^2 = N, \tag{12.3}$$

since the coverage does not increase too much then. Equation 12.3 is called the *matrix stopping rule*. Hellman recommends to set $m = N^{\frac{1}{3}}$ and $t = N^{\frac{1}{3}}$. As the table contains less than $m \cdot t = N^{\frac{2}{3}}$ keys, Hellman recommends to generate $r = N^{\frac{1}{3}}$ tables, each having its unique re-randomization and reduction function R.

The following formulas are based on the analysis done in [Hel80].

Success Probability. The success probability of a single table was calculated by Hellman to be

$$P_{table} = \frac{1}{N} \sum_{i=1}^{m} \sum_{j=0}^{t-1} \left(\frac{N - (i \cdot t)}{N} \right)^{j+1} .$$

The success probability of r generated tables is approximated by

$$P_{total} = 1 - (1 - P_{table})^r .$$

Precomputation Time. This time is proportional to a total number of applications of the step function f. Therefore it is

$$PT = m \cdot t \cdot r.$$

Memory Complexity. Only the start point and the end point are stored out of each chain. If the start points were chosen to be e.g. consecutive numbers, then we need at least $b_{SP} = \lceil \log_2(m) \rceil$ bits to unambiguously identify each start point. To store information about the end point, we need $b_{EP} = \lceil \log_2(N) \rceil$ bits. We generate r tables, each containing m chains, therefore the total memory complexity would be

$$M = r \cdot m \cdot b_{pair},$$

where

$$b_{pair} = b_{SP} + b_{EP} = \lceil \log_2(m) \rceil + \lceil \log_2(N) \rceil .$$

Online Time. As described above, the search through a single table requires one application of R and up to $t - 1$ applications of f. If neglecting the time

to handle false alarms, we get the worst case estimation for the search through r different tables as

$$T = r \cdot t.$$

Table Accesses. After each application of R or f, the result is looked up in the tables. Therefore the maximum number of table accesses is

$$TA = T = r \cdot t.$$

Parameters Choice. For recommended values of $m = t = r = N^{\frac{1}{3}}$, we obtain $PT = N$ applications of f in the precomputation phase, $M = N^{\frac{2}{3}}$ pairs $\{SP, EP\}$ to be stored, and $T = N^{\frac{2}{3}}$ applications of f followed by the same number $TA = N^{\frac{2}{3}}$ of table accesses in the online phase.

12.2 Distinguished Points

An extreme number of table accesses TA in the online phase may jeopardize the attack. For example, in case of DES, the number of table accesses would be $TA = N^{\frac{2}{3}} = 2^{37\frac{1}{3}}$. If we assume that one table access takes 1 ms on average (currently stated values of an average disk access time are about 8 ms), then we would need 5.5 years to finish up the online phase.

To handle this problem, Rivest [D. 82] in 1982 proposed the modification of the original Hellman's approach, based on so-called *distinguished points*. A distinguished point (DP) is a key with a certain property which is easy to verify, e.g. a key having the 20 most significant bits being all zeros. The DP-property is usually characterized by a mask with the length of d bits.

12.2.1 Offline Phase

In the offline phase, each chain is generated until some distinguished point is reached. Therefore, the end point is always a distinguished point. Consequently, the chains do not have constant length; the parameter t on a length of chain is replaced by the specification of the DP-property, which usually states that DP is a point having d most significant bits being zero. The DP-property is usually combined with the range $[t_{min}, t_{max}]$ defining a minimum required and a maximum allowed length of the chain. The chains that do not reach the minimum required length t_{min} do not contribute significantly to the coverage

of the table and hence such chains are discarded to save memory requirements. The chains exceeding the maximum allowed length t_{max} are likely to run into the chain loop and for this reason such chains are discarded as well. Only the chains having the length within the specified boundaries are stored. Out of each chain, the triple $\{SP_j, EP_j, l_j\}$ is stored, where l_j is a length of the chain. However, the information on the length of chain can be omitted, as will be shown below.

Chain merges and loops. The detection of chain merges is very easy in this case. If two chains merge, then both chains end up in the same distinguished point — the end point. As both chains contain the same sequence of keys from the point of merge, only one chain is stored to save memory requirements. Obviously, it is the longer chain which is stored since it contributes to the coverage of the table more. Chain loops are prevented by setting the condition on the maximum allowed chain length t_{max}, as discussed above.

12.2.2 Online Phase

The online phase is similar to the online phase of the original Hellman's method. First, the re-randomization and reduction function R is applied to the given ciphertext C_0:

$$y_1 = R(C_0).$$

Since all end points are DPs, then the result y_1 is looked up in the table only when it is a DP. Otherwise, further results

$$y_i = f(y_{i-1}) \tag{12.4}$$

are iteratively calculated until the DP is reached or until $i \geq t_{max}$.

If the DP is reached but no match in the table is found, then the iterative calculation described in Equation 12.4 is not continued, since there is no chance of any further match (each chain contains exactly one DP — the end point). Instead, the next table is processed.

If the DP is reached in the i-th step and the result y_i matches to the end point EP_j, then the candidate for the key K is retrieved as

$$k_{j,l_j-i} = f^{l_j-i}(SP_j),$$

where l_j is a length of the chain which has been stored in the triple $\{SP_j, EP_j, l_j\}$.

If the length of the chain has not been stored, then the step function f is iteratively applied starting from SP_j

$$k_{j,\ell} = f^{\ell}(SP_j), \quad \ell = 1, 2, \ldots$$

until $k_{j,\ell} \equiv y_1$. Then the candidate for the key K is the point $k_{j,\ell-1}$.

12.2.3 Characteristics

The fact that chains do not have constant length, as well as the fact that some chains are discarded due to the chain merges, lead to more complicated evaluation of the characteristics of the method. The following overview is based on the analysis made in [SRQL02].

Let m be the number of start points in one table, r be the number of tables, k be the bit-length of points in the table, d be the number of mask bits defining the DP-property and $[t_{min}, t_{max}]$ be the minimum and maximum allowed length of chain.

The probability that a chain leads into a DP in less than l iterations of the step function is

$$P_1(l) = 1 - \prod_{i=0}^{l-1} \left(1 - \frac{2^{k-d}}{2^k - i}\right) \approx 1 - \left(1 - \frac{2^{k-d}}{2^k - \frac{l-1}{2}}\right)^l.$$

Since $l \ll 2^k$, the probability $P_1(l)$ can be further approximated

$$P_1(l) \approx 1 - \left(1 - \frac{2^{k-d}}{2^k}\right)^l = 1 - \left(1 - \frac{1}{2^d}\right)^l, \tag{12.5}$$

however, this approximation is not used in [SRQL02].

Chain Length. The average chain length can be calculated as

$$l_{avg} = \frac{\sum_{l=t_{min}}^{t_{max}} l \times P(\text{DP in exactly } l \text{ iterations})}{\sum_{l=t_{min}}^{t_{max}} P(\text{DP in exactly } l \text{ iterations})}. \tag{12.6}$$

The denominator of Equation 12.6 represents the ratio between the number of chains having the length $l \in [t_{min}, t_{max}]$ and the total number of chains. It is expressed as

$$\begin{aligned}\gamma &= P_1(t_{max}) - P_1(t_{min} - 1) \\ &\approx \left(1 - \frac{2^{k-d}}{2^k - \frac{t_{min}-2}{2}}\right)^{t_{min}-1} - \left(1 - \frac{2^{k-d}}{2^k - \frac{t_{max}-1}{2}}\right)^{t_{max}}.\end{aligned}$$

Consequently, the number of chains having the length $t_{min} \leq l \leq t_{max}$ is

$$\begin{aligned}m' &= m \cdot \gamma \\ &\approx m \cdot \left(\left(1 - \frac{2^{k-d}}{2^k - \frac{t_{min}-2}{2}}\right)^{t_{min}-1} - \left(1 - \frac{2^{k-d}}{2^k - \frac{t_{max}-1}{2}}\right)^{t_{max}}\right).\end{aligned} \tag{12.7}$$

As some chains are sorted out if chain merge appears, the final number $\widehat{m}$ of chains in one table is further reduced, $\widehat{m} \leq m'$.

The average chain length is approximated as

$$l_{avg} \approx \frac{\left((1-x)^{t_{min}-2}\left(t_{min} + \frac{1-x}{x}\right) - (1-x)^{t_{max}-1}\left(t_{max} + \frac{1-x}{x}\right)\right)}{\gamma},$$

where

$$x = \frac{2^{k-d}}{2^k - \frac{t_{max}+t_{min}}{4}}.$$

As some chains are sorted out if the chain merge appears, the actual average chain length l'_{avg} differs from l_{avg}. Since longer chains are more likely to merge than shorter chains, the actual average length l'_{avg} is lower than l_{avg}, as will be confirmed later in Section 13.3. The actual average chain length l'_{avg} is decreasing with growing m' since chain merges are more frequent.

Success Probability. The number of distinct keys covered by one table has been approximated as

$$s(\gamma m) \approx N \cdot \left(1 - \left(\frac{N}{-l_{avg}\gamma m + l_{avg}^2 \gamma m + N}\right)^{\frac{1}{l_{avg}-1}}\right).$$

Therefore the success probability of one table is

$$P_{table} = \frac{s(\gamma m)}{N} \approx 1 - \left(\frac{N}{-l_{avg}\gamma m + l_{avg}^2 \gamma m + N} \right)^{\frac{1}{l_{avg}-1}} .$$

Having r tables the total success probability can be approximated by

$$P_{total} \approx 1 - (1 - P_{table})^r .$$

Precomputation Time. The average number of the step functions applied to one chain during the precomputation phase is

$$\begin{aligned} \delta &= t_{max} \cdot (1 - P_1(t_{max})) + l_{avg} \cdot P_1(t_{max}) \\ &= t_{max} + (l_{avg} - t_{max}) \cdot P_1(t_{max}) . \end{aligned} \tag{12.8}$$

As there are $r \cdot m$ chains to be precomputed, the precomputation time is

$$\begin{aligned} PT &= r \cdot m \cdot \delta \\ &\approx r \cdot m \cdot \left(t_{max} + (l_{avg} - t_{max}) \cdot \left(1 - \left(1 - \frac{2^{k-d}}{2^k - \frac{t_{max}-1}{2}} \right)^{t_{max}} \right) \right) \end{aligned}$$

step functions.

Memory complexity. Each table contains $\frac{s(\gamma m)}{l'_{avg}}$ chains. To store information on chains from r tables, we need

$$M = \frac{s(\gamma m)}{l'_{avg}} \cdot r \cdot b_{triple} \approx \frac{s(\gamma m)}{l_{avg}} \cdot r \cdot b_{triple}$$

bits of memory, where

$$b_{triple} = (\lceil \log_2(m) \rceil) + (\lceil \log_2(N) \rceil - d) + (\lceil \log_2(t_{max}) \rceil)$$

is the least number of bits necessary to store information on one triple $\{SP_j, EP_j, l_j\}$.

Online Time. The online time may be lower bounded by

$$T \leq r \cdot l'_{avg} \approx r \cdot l_{avg}.$$

Table Accesses. The maximum number of table accesses may be bounded by a number of tables

$$TA = r,$$

since there is at most one look-up per one table.

12.3 Time-Memory-Data Trade-off Attacks

The idea of a time-memory-data tradeoff attack, suitable for stream ciphers, was independently introduced by Babbage [Bab95] and Golic [Gol97]. Biryukov and Shamir [BS00] later combined this approach with the original Hellman's method.

Let g be the function mapping each one of N internal states $x \in \mathbb{X}$ of the stream cipher to the output prefix $y \in \mathbb{Y}$. Output prefix y here is the string consisting of the first $\log_2(N)$ bits produced by the cipher from this internal state

$$g : \mathbb{X} \mapsto \mathbb{Y}.$$

We denote

$$y = g(x). \tag{12.9}$$

The typical cryptanalytical task in case of stream ciphers is to invert the function g, i.e. for the given output prefix y to find such x that Equation 12.9 holds.

Let w be the number of the available keystream bits. Since sometimes $w > \log_2(N)$, it is possible to derive $D = w - \log_2(N) + 1$ output prefixes $y_0, y_1, \cdots y_{D-1}$ from the keystream, as shown in Figure 12.3. To break the stream cipher, it is sufficient to find the internal state x_i for any of the D output prefixes y_i. When the internal state x_i is found, then the cipher can be run forward to derive the remainder of the keystream. In some cases, the cipher can also be run backwards, as shown in Chapter 14 for the case of A5/1.

The birthday paradox says that two subsets of the space with N points are likely to intersect if the product of their sizes exceeds N. The idea of Babbage and Golic was to apply this property to the table look-up attack, which was described in Section 12.1. As we have D output prefixes available, then the size of the table is reduced to a fraction of $\approx \frac{N}{D}$ pairs $\{x, y\}$.

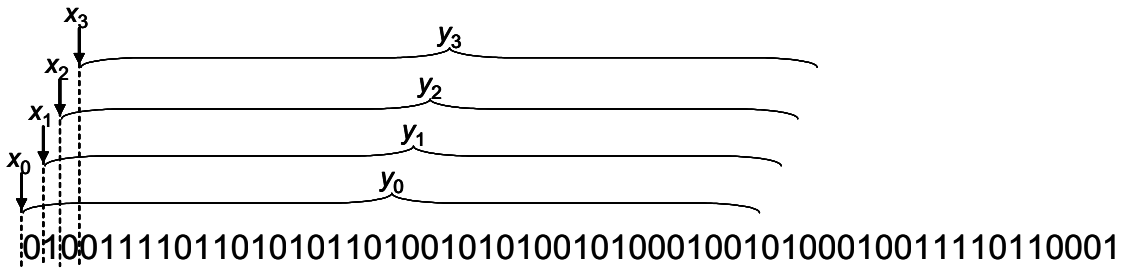

Figure 12.3: Sampling the keystream into D output prefixes

The idea of Biryukov and Shamir was to apply this approach to the Hellman's method. Consequently, the number of generated tables is D times smaller, $r_D = \frac{r}{D}$, which saves both the precomputation time

$$PT_D = \frac{PT}{D}$$

and the memory

$$M_D = \frac{M}{D}.$$

The online complexity remains unchanged — for each data point we have to perform calculations and table look-ups over the reduced number of tables r_D, however these operations are repeated for each of D data points. Therefore the online time is

$$T_D = D \cdot \frac{T}{D} = T$$

and the number of table accesses is

$$TA_D = D \cdot \frac{TA}{D} = TA.$$

12.4 Rainbow Tables

The idea of the rainbow tables, introduced in 2003 by Oechslin, is to apply a different re-randomization and reduction function R_i in each step of chain generation. The method was first described for the use with block ciphers.

12.4.1 Offline Phase

The step function f changes from step to step creating the sequence of t different step functions in each chain

$$f_1 f_2 f_3 \dots f_t$$

Chain loops are completely prevented since each step function is used exactly once. Chain merges are possible only if two identical data points are generated in the same column of the table, i.e. after application of the same step function f_i in two distinct chains. Since the probability of chain merges is significantly reduced in comparison to the Hellman's approach, the number m of chains contained in one table can be increased to the value for which $m \cdot t \approx N$. Consequently, a single table or just a few tables are typically generated in the precomputation phase.

12.4.2 Online Phase

The online phase differs from the Hellman's approach as there are different reduction functions used in each step of the chain generation. For a given ciphertext C_0, first the result $R_t(C_0)$ is calculated and compared to all end points. If no match is found, then the result $f_t(R_{t-1}(C_0))$ is calculated and looked up in the table, then the result $f_t(f_{t-1}(R_{t-2}(C_0)))$, etc. If a match is found, then, similarly to the Hellman's method, the corresponding chain is reconstructed to retrieve the candidate for the key.

12.4.3 Characteristics

The following results have been adopted from the analysis provided in [Oec03].

Success Probability. The success probability of one table has been estimated as

$$P_{table} = 1 - \prod \left(1 - \frac{m_i}{N}\right),$$

where $m_1 = m$ and $m_{n+1} = N\left(1 - e^{-\frac{m_n}{N}}\right)$. The success probability of r tables is, similarly to Hellman, estimated as

$$P_{total} = 1 - (1 - P_{table})^r .$$

Precomputation Time. The precomputation time is estimated in the same way like for Hellman's method

$$PT = m \cdot t \cdot r.$$

Memory Complexity. Also the estimation of memory complexity is similar to Hellman's method

$$M = r \cdot m \cdot b_{pair},$$

where

$$b_{pair} = b_{SP} + b_{EP} = \lceil \log_2(m) \rceil + \lceil \log_2(N) \rceil$$

is the number of bits necessary to store one pair $\{SP, EP\}$.

Online Time. In the i-th iteration of the online phase, $0 \leq i \leq t-1$, we first apply the function R_{t-i}, followed by i step functions $f_{t-i+1}, f_{t-i+2}, \cdots f_{t-1}, f_t$. The maximum number of calculations for one table is then $\frac{t(t+1)}{2}$ and the worst case time to search through r tables is then

$$T = r \cdot \frac{t(t+1)}{2}.$$

However, as mentioned above, the number of tables r is typically a very small number.

Table Accesses. Each iteration of the online phase is finished with a table look-up. Therefore the maximum number of table accesses into one table is equal to t, and the worst case number of the accesses into r tables is

$$TA = r \cdot t.$$

Parameters Choice. For recommended values of $m = N^{\frac{2}{3}}$, $t = N^{\frac{1}{3}}$ and $r = 1$, we obtain $PT = N$ applications of f in the precomputation phase, $M = N^{\frac{2}{3}}$ pairs $\{SP, EP\}$ to be stored, and $T = \frac{N^{\frac{2}{3}}}{2}$ calculations with $TA = N^{\frac{1}{3}}$ table accesses in the online phase.

12.5 Thin-Rainbow Tables

The rainbow tables method is inferior to the standard Hellman's method, whenever multiple data points are available, as has been shown in [BBS06]. Therefore, a pure rainbow method is not suitable for cryptanalysis of the stream ciphers.

The fact of having multiple data points enables to save precomputation time and memory, similarly to the Hellman's TMDTO described in Section 12.3. However, the online complexity is increased. The online time is

$$T_D = D \cdot \left\lceil \frac{r}{D} \right\rceil \cdot \frac{t(t+1)}{2}$$

and the number of table accesses is

$$TA_D = D \cdot \left\lceil \frac{r}{D} \right\rceil \cdot t,$$

where D is the number of data points. Since typically $r < D$ (in many cases $r = 1$), then the product $D \cdot \left\lceil \frac{r}{D} \right\rceil > r$, which indicates the increased online complexity.

To cope with this problem, Barkan, Biham and Shamir [BBS06] proposed a modified scheme called the *thin-rainbow method*. In this scheme, the shorter *rainbow sequence* consisting of S different step functions is t-times repeated during the generation of one chain, as shown below:

$$f_1 f_2 \dots f_S f_1 f_2 \dots f_S \dots\dots\dots f_1 f_2 \dots f_S.$$

Each chain contains t rainbow sequences, each rainbow sequence consists of S different step function. Consequently, each chain consists of $tS + 1$ points (including the start point and the end point). By keeping t to be greater than D, $t \geq D$, the optimum trade-off may be reached whenever the multiple data is available. If multiple tables are generated, then each table has its unique rainbow sequence $f_1 f_2 \dots f_S$. Let us note that by setting $S = 1$ we obtain the standard Hellman's method (with multiple data) as described in Section 12.3.

12.5.1 Thin-Rainbow Tables with Distinguished Points

The method can be combined with the Rivest's approach of the distinguished points to reduce the number of table accesses. In this case the DP-property

is not checked after each application of the step function, but only after each application of f_S, which is the last step function of each rainbow sequence. In the following text we focus on this variant.

12.5.2 Offline Phase

Similarly to previously described methods we generate m chains in each of r tables. As the distinguished points approach is applied, the chains do not have a constant length. Therefore, the definition of t is replaced by the definition of the DP-property being specified by a d-bit mask. We also set the region $[t_{min}, t_{max}]$ of a minimum required and a maximum allowed length of the chain, which is here expressed in the number of rainbow sequences. Only the chains that reach the DP after $t_{min} \leq l \leq t_{max}$ applications of a rainbow sequence are stored in the table. If two chains merge (which is again detected by equal end point), then the shorter one is sorted out.

12.5.3 Online Phase

In the online phase we are given D data points y. Let's focus on one data point y_i, and one table. First, we calculate $R_S(y_i)$ and check it for the DP-property. If a DP has been reached, then we look-up the table. If not, then we follow with ℓ iterations of the rainbow sequence $f_1 f_2 \dots f_S$ until DP is reached or until $\ell > t_{max}$. In the second step we calculate $f_S(R_{S-1}(y_i))$ first, check it for the DP-property and, if necessary, we continue the calculation by ℓ rainbow sequences. In the next step, we start calculation with $f_S(f_{S-1}(R_{S-2}(y_i)))$, etc. In the last step, we start the calculation with $f_S(f_{S-1} \dots f_2(R_1(y_i))))$, followed by ℓ rainbow sequences, if necessary.

If the table has been completely processed/searched for the data point y_i, and no success has been reached yet, then we follow the search with the same data point y_i in the next table. After having processed the last table, the whole procedure is repeated for other data points y. If a match is found, then, similarly to the Hellman's method, the corresponding chain is reconstructed to retrieve the candidate for the key/internal state.

12.5.4 Characteristics

The analysis of this method has been presented in [A.12]. This analysis is based on approaches used in [SRQL02] and [Oec03]. The analysis in [A.12] is

made for one table, here we present the results for r tables. However, in many cases, $r = 1$.

Success Probability. To approximate the probability that the DP is reached in less than l applications of a rainbow sequence (or, in other words, in less then l applications of f_S), we use Equation 12.5:

$$P_1(l) \approx 1 - \left(1 - \frac{1}{2^d}\right)^l.$$

The average length of a chain (in the number of rainbow sequences) is, from Equation 12.6, approximately

$$l_{avg} \approx \frac{\sum_{l=t_{min}}^{t_{max}} l \times \left(\left(1 - \frac{1}{2^d}\right)^{l-1} - \left(1 - \frac{1}{2^d}\right)^l\right)}{\left(1 - \frac{1}{2^d}\right)^{t_{min}-1} - \left(1 - \frac{1}{2^d}\right)^{t_{max}}}. \tag{12.10}$$

The number of chains having the length $t_{min} \leq l \leq t_{max}$ rainbow sequences (or, in other words, having the length $t_{min}S \leq lS \leq t_{max}S$ step functions) is, similarly to Equation 12.7, approximately

$$m' \approx m \cdot \left(\left(1 - \frac{1}{2^d}\right)^{t_{min}-1} - \left(1 - \frac{1}{2^d}\right)^{t_{max}}\right). \tag{12.11}$$

As mentioned above, due to the chain merges the final number $\widehat{m}$ of chains in one table is lower, $\widehat{m} \leq m'$, since the shorter ones of the merged chains are sorted out. To evaluate the number of chains $\widehat{m}$, we used the iterative expression for m_i, which represents the number of *distinct* points in the i-th column of the table

$$\begin{aligned} m_i &= \left(N - \sum_{k=1}^{\lfloor \frac{i}{S} \rfloor} m_{i-kS}\right)\left(1 - \left(1 - \frac{1}{N}\right)^{m_{i-1}}\right) \\ &\approx \left(N - \sum_{k=1}^{\lfloor \frac{i}{S} \rfloor} m_{i-kS}\right)\left(1 - e^{\frac{m_{i-1}}{N}}\right). \end{aligned} \tag{12.12}$$

Upon setting $m_1 = m'$ we obtain $\widehat{m} = m_{l_{avg}S+1}$. Then the success probability of one table and one data point y is approximately

$$P_{table} \approx \frac{\widehat{m} l_{avg} S}{N}$$

and the success probability of r tables and D data points y consequently is approximately

$$P_{total} \approx 1 - \left(1 - \frac{\widehat{m} l_{avg} S}{N}\right)^{rD}. \tag{12.13}$$

Precomputation time. Similarly to Equation 12.8 the average number of rainbow sequences applied to one chain is again equal to

$$\delta = t_{max} \cdot (1 - P_1(t_{max})) + l_{avg} \cdot P_1(t_{max}). \tag{12.14}$$

Since (i) each rainbow sequence is S step functions long, (ii) we (initially) generate m chains in each table, and (iii) we generate r tables, then the precomputation time (expressed in the number of step functions) can be approximated as

$$PT \approx r \cdot m \cdot \delta \cdot S. \tag{12.15}$$

Memory Complexity. Each table contains $\widehat{m}$ chains. To store information on the chains from r tables we need

$$M = \widehat{m} \cdot r \cdot b_{triple} \tag{12.16}$$

bits of memory, where

$$b_{triple} = (\lceil \log_2(m) \rceil) + (\lceil \log_2(N) \rceil - d) + (\lceil \log_2(t_{max}) \rceil)$$

is the least number of bits necessary to store information on one triple $\{SP_j, EP_j, l_j\}$.

Online Time. For one data point and one table we have to compute at most S chains, each being about $S \cdot \delta$ step functions long. Therefore, for D data points and r tables, we obtain the worst case online complexity to be

$$T = r \cdot D \cdot S \cdot S \cdot \delta = r \cdot D \cdot S^2 \cdot \delta. \tag{12.17}$$

Table Accesses. Calculation of each chain is finished by one table look-up (if the DP is reached before t_{max} iterations of the rainbow sequence). Therefore, the worst case number of the table accesses is

$$TA = r \cdot D \cdot S. \tag{12.18}$$

Parameters Choice. Let assume that only one table is generated, $r = 1$. We set such a DP-criterion and the region $[t_{min}, t_{max}]$ to obtain $l_{avg} \approx D$. If $P_1(t_{max}) > 0.5$, then, according to Equation 12.14, we may roughly approximate $\delta \approx l_{avg} \approx D$. The length of one rainbow sequence we set as $S \approx \frac{N^{\frac{1}{3}}}{D}$. The number of chains to be precomputed is set as $m \approx \frac{N^{\frac{2}{3}}}{D}$.

After substitution to Equations 12.13–12.18, we obtain a success probability $P_{total} \approx 1 - \left(1 - \frac{1}{D}\right)^D \approx 1 - \frac{1}{e}$, precomputation time $PT \approx \frac{N}{D}$, memory complexity $M < \frac{N^{\frac{2}{3}}}{D}$ triples $\{SP, EP, l\}$, online time $T \approx N^{\frac{2}{3}}$ and the number of table accesses $TA \approx N^{\frac{1}{3}}$. As obvious from these very rough estimations, on careful selection of the parameters, the characteristics of the thin-rainbow DP TMDTO method are comparable to the Hellman's DP TMDTO method, discussed in Sections 12.2 and 12.3.

Chapter 13

Time-Memory-Data Trade-off Attack on A5/1

In this chapter we describe our implementation of TMDTO attack on A5/1. COPACOBANA is used to perform calculations in both the precomputation and the online phase.

In the first part we focus on the precomputation phase. Upon analysis made in Chapter 12 we make a selection of the method to be implemented in COPACOBANA. After that we analyze potential design approaches for hardware architecture and we select the approach that permits achievement of maximum performance. Then the description of the architecture for the table precomputation follows. We continue with a description of a method for the fast sort of disk-stored TMTO tables. The first part we conclude with implementation results which we obtained after the test run of the precomputation phase. We compare theoretical and practical results and we identify some of potential sources making differences between them.

In the second part we focus on the online phase. We describe the architecture of the online engine which has been implemented in COPACOBANA. The online engine is more complex than the precomputation one. After that, we discuss the method used for the fast look-up in disk-stored TMDTO table.

This has been a collaborative work with Andy Rupp. Some results of this work have already been published in [A.12].

13.1 Table Precomputation

To make our attack realistic, we assume that the number of known consecutive keystream bits is relatively small, but still we consider that we know at least one whole keystream block of 114 bits. Hence, we have $D = 114 - 64 + 1 = 51$ data points available to break the cipher.

13.1.1 Chosen Method

Both Hellman's original tradeoff and rainbow tables are well suited for parallelization in hardware. Since the chains have a fixed length, the control of the calculation in the precomputation phase is simple. However, as discussed above, the Hellman's method requires a large number of disk accesses during the online phase, and the rainbow table tradeoff curve is inferior to Hellman's one whenever we have multiple data [BBS06].

Ergo, we considered two candidates for the implementation: the Hellman's DP TMDTO method, described in Sections 12.2 and 12.3, and the thin-rainbow DP TMDTO method described in Section 12.5. Both methods provide comparable characteristics with regard to success probability P_{total}, precomputation time PT, memory complexity M, online time T, as well as a low number of table accesses TA. Nevertheless, there is a certain important difference in both methods: in case of the Hellman's DP method, the chain can reach its end point after *any* application of the step function f, while in case of the thin-rainbow DP method, the chain can reach its end point only after application of the last step function of the rainbow sequence, f_S. In other words, in case of the Hellman's DP we have to test the DP-property after *each* application of f, while in case of the thin-rainbow DP, we have to test it after f_S only. By exploiting this property we may save considerable amount of hardware resources necessary for the units testing the DP-property, and therefore we may gain higher performance. Hence, for our implementation, we have selected the thin-rainbow DP method. In the case of multiple data this approach allows simple and efficient hardware implementation, while exhibiting a low number of disk accesses during the online phase.

13.1.2 Design Approach

Most designs realized in FPGAs usually do not fully use flip-flops available on the chip. Typically, the designs are limited by the number of combinational

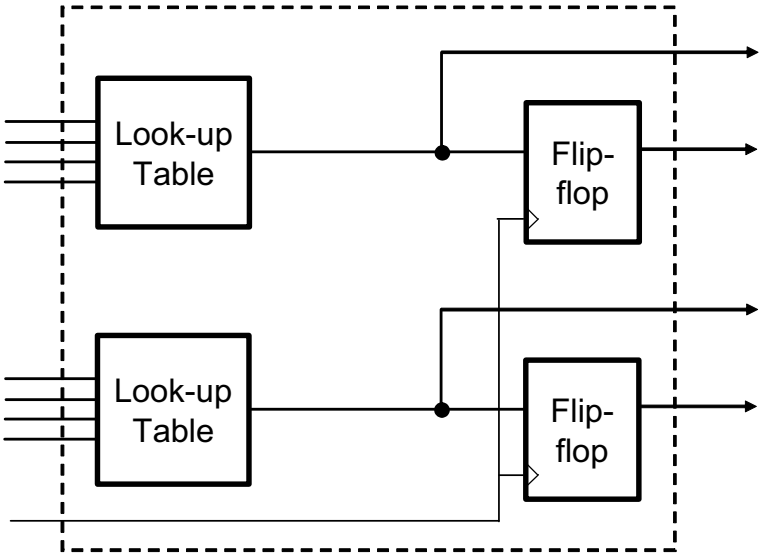

Figure 13.1: Simplified diagram of Xilinx FPGA slice

resources (look-up tables, LUTs) available. In case of A5/1 it is different: a demand for flip-flops prevails a demand for combinational logic.

As generally known, the best performance may be gained by exploiting the properties of a target architecture. Let's focus on it now. Xilinx Spartan 3-1000 FPGA used in COPACOBANA contains 7680 slices [Xil07]. Each slice contains 2 LUTs and 2 flip-flops (and other circuits, like the fast carry logic or dedicated multiplexers), see Figure 13.1. Look-up tables are used to implement combinational logic, but in some slices the LUTs can also be configured to work as a shift register with a maximum length of 16 bits (denoted as SRL16). This property enables to implement much bigger shift-register-based circuits. For example, using all 15360 flip-flops we may build the circuit containing equivalent of $15360/64 = 240$ A5/1 cores (without any controller and other circuits). On the other hand, using SRL16s we can implement up to 480 A5/1 cores, still leaving enough LUTs and flip-flops for a controller and other necessary circuits.

However, the usage of SRL16s brings some limitations to the circuit design. To allow the synthesis tool to utilize this property, we have to avoid any parallel input or output to the shift register, using only serial input and output. Hence we rejected the idea of building the pipeline as at the DES engine [KPP+06], since such a pipeline would require parallel access to all bits of registers in each pipeline stage. Instead, we decided to implement an array of small, encapsulated, independent processing units (we call them *TMTO elements*), each having one serial input and one serial output.

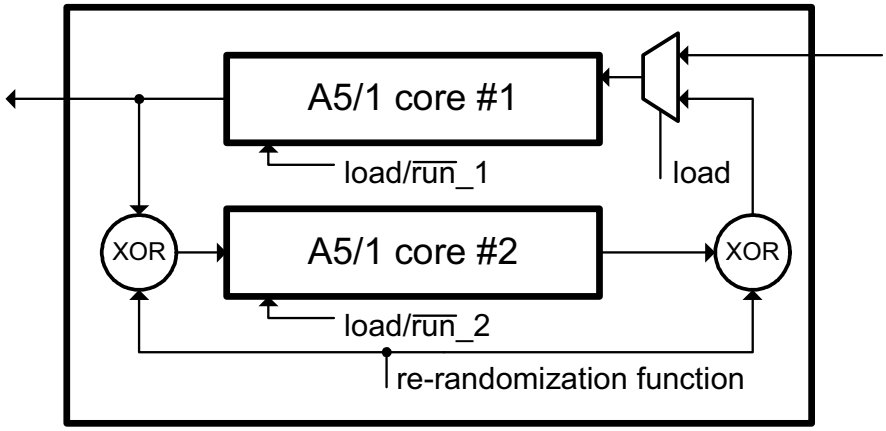

Figure 13.2: TMTO element — a processing unit calculating one chain of the table

13.1.3 The TMTO Element

The diagram of the TMTO element is shown in Figure 13.2. The element consists of two modified A5/1 cores, which can be operated either in RUN mode, or in LOAD mode. In RUN mode the Core operates as a standard A5/1 unit, i.e. it produces the keystream. When switched to LOAD mode, the registers $R1$, $R2$ and $R3$ are interconnected to create one large 64-bit shift register with one serial input and one serial output.

Each TMTO element is calculating one chain of points, i.e. one row in the TMTO table. It works like a two-stroke two-piston engine. In *odd* steps of the rainbow sequence, the Core #1 produces the keystream $y_{j,i}$ that is on-the-fly re-randomized and loaded to the Core #2 as the new internal state $x_{j,i+1}$. In *even* steps of the rainbow sequence, the functionalities of both cores are swapped — now the Core #2 is in RUN mode, while the Core #1 is in LOAD mode. When the rainbow sequence is finished (i.e. after application of f_S), the result stored in the Core #1 is shifted out to be checked for the DP-property. As obvious, our design requires the rainbow sequence to be even number of steps long (S must be an even number), however, this fact does not represent any significant limitation.

As a source of re-randomization, we use the output of the long-period LFSR. The output of LFSR is XORed with the keystream.

13.1.4 Architecture of the Table Precomputation Engine

An architecture of the precomputation engine, occupying one FPGA, is presented in Figure 13.3. Since COPACOBANA hosts up to 120 FPGA, we may place the same number of precomputation engines in one COPACOBANA.

We were able to place 234 TMTO elements into one FPGA. The TMTO elements are depicted in the upper part of the diagram. To gain the maximum frequency, we rejected the idea of parallel access to the TMTO elements, since the number of them is relatively large. Instead, we connected all TMTO elements into one large chain. The communication overhead, increased due to the serial access, still represents only a small fraction of the overall computation time, which is more than compensated by the increased frequency.

All TMTO elements share one *DP checker* depicted on the left side of the diagram. After the application of f_S, the results held in TMTO elements are shifted out to be checked for the DP-property by this unit.

The 64-bit *start point generator* consists of a 40-bit register holding the static value of upper bits, and a 24-bit counter generating the value of lower bits. By incrementation, the counter generates the sequence of consecutive start points. Upon startup of COPACOBANA the start point generator is initialized with a certain initial value; each precomputation engine (i.e. each FPGA) is assigned its unique initial value by the host computer. Besides that, each precomputation engine is also assigned the range of the start point values to be generated. When this range is exhausted, the host computer assigns a new range — and a new initial value of the start point generator — to the precomputation engine, and the engine is reset.

The *point register* is a 64-bit shift register with a parallel load. Newly generated start point is loaded from the start point generator to the point register, from which it is then shifted out to the corresponding TMTO element.

The *timer* is incremented at the beginning of each rainbow sequence. It is used to determine the length of each generated chain, as discussed below.

For each chain, generated in the specific TMTO element, a value of its start point and its 'birthdate' is stored in the *chain memory*. The 'birthdate' is a value of the timer at time of start point generation.

As discussed above, we use the long-period LFSR as a *re-randomization function generator*. This unit is again shared by all TMTO elements, since all TMTO elements use the same re-randomization function R_i at the same time.

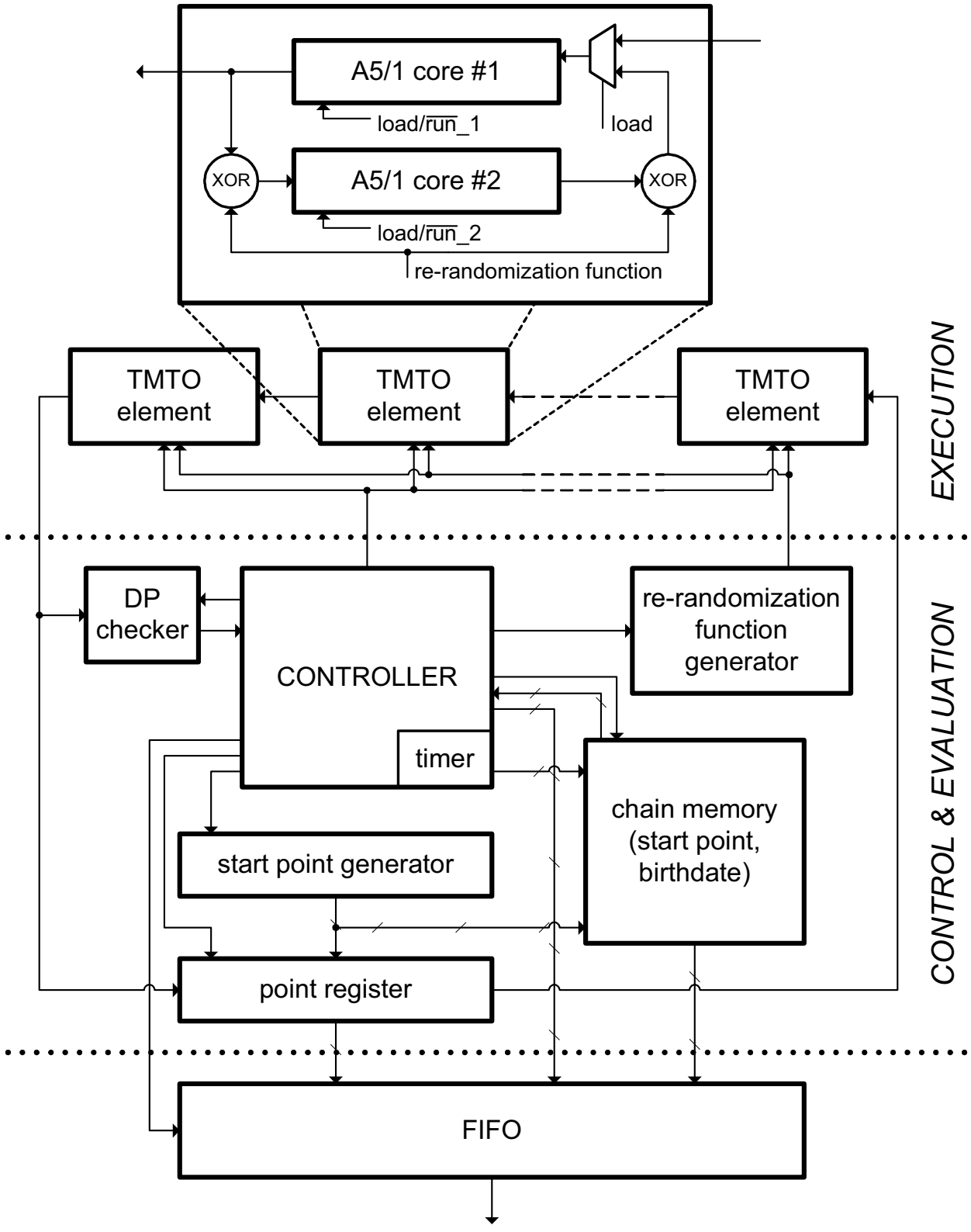

Figure 13.3: Architecture of an A5/1 precomputation engine

The information on the start point, the end point and the length of each successfully generated chain is collected in a *FIFO*, from which it is read in bursts by the host computer.

How It Works. After the reset, the controller runs the *initialization phase*, during which all TMTO elements are initialized with start points. This phase is run exactly once. After that, during a standard operation, phases of *execution* are alternated with phases of *evaluation*.

Initialization. During initialization the TMTO elements are switched to build, together with point register, one large shift register. Start points generated for all TMTO element are loaded to the point register and shifted to the corresponding TMTO elements. The values of start points and their 'birthdates' (time of their generation) are concurrently stored in the chain memory. Besides that the timer, the re-randomization function generator and some other auxiliary circuits are initialized.

Execution. Execution itself is performed in TMTO elements. One phase of execution lasts exactly one rainbow sequence. The function has been already described in Subsection 13.1.3.

Evaluation. When the rainbow sequence is finished, all TMTO elements are again switched into one large shift register and the results are shifted-out. The results are on-the-fly checked for the DP-property, and the length l of each chain is calculated as a difference between a 'current date' (which is the current value of the timer) and the 'birthdate' of the corresponding chain, which is stored in the chain memory. For each chain one of the following four cases may appear:

1. If the result satisfies the DP-property (it is the *end point*) and if the length l of the chain is within bounds, $t_{min} \leq l \leq t_{max}$, then the chain information (start point, end point, length) is stored into FIFO — the value of the end point is acquired from the point register and the value of the start point is acquired from the chain memory.

   Concurrently, a new start point is generated, as a new chain will be calculated in the corresponding TMTO element. This new start point replaces the end point in the point register. At the same moment, both the new start point and the 'birthdate' are stored in the chain memory.

Information stored in FIFO is later read out by the host computer and stored on the disk.

2. If the end point has been achieved, but the chain is too short, $l < t_{min}$, then the chain is discarded and the generation of a new chain is initiated in the same TMTO element — the procedure is the same as above (the new start point is generated, loaded into the point register and, together with a 'birthdate', stored in the chain memory), but the chain information is not stored in FIFO.

3. If the chain becomes too large, $l > t_{max}$, then again the chain is discarded and the generation of a new chain is initiated in the same TMTO element.

4. In all other cases (i.e. the end point has not been achieved and the chain is not too long yet) the result is simply shifted back to the TMTO element to continue with the next rainbow sequence.

During the phase of evaluation we also initialize the re-randomization function generator to generate the same rainbow sequence in the next phase of execution.

13.1.5 Data Transfer from COPACOBANA to the Host Computer

The host computer regularly polls every FPGA, reads out the chains information collected in FIFO, and stores it on the disk. As evident from the diagram of COPACOBANA (see Figure 10.3), the data is transfered via the bus and the interface. For its electrical properties the bus represents relatively high risk of data corruption.

As clear from the previous discussions, all stored data must be correct, otherwise they may lead to incorrect results during the online phase of the attack. On the other hand, lost of any chain information does not represent any significant problem — a new chain may be calculated instead.

We have chosen a Hamming code to secure the correctness of the transfered data. The 64-bit start points and end points are encoded with the Hamming code(72, 64) and the length l is encoded with the Hamming code(16, 11). Though these codes are commonly used as SEC-DED (single error correction, double error detection) codes, we use them as TED (triple error detection) code for further improvement in the error resistance. If an error appears, then the chain information is simply discarded. Besides the detection of 100% of

single, double and triple errors, the variant of the Hamming code(72, 64) used ensures also the detection of 99.19% of all quadruple errors [Hsi70].

13.1.6 Selection of Parameters

The A5/1 table precomputation engine can run at a maximum frequency of 156 MHz. Computing the step-function f_i takes 64 clock cycles. One FPGA contains 234 TMTO elements (each consisting of two A5/1 cores), hence the whole COPACOBANA can perform approximately 2^{36} step-functions per second. In contrast to that, a contemporary PC can execute just about 2^{22} step-functions per second [BBK06]. The performance of COPACOBANA in the online phase is about $2^{34.65}$ step-functions per second, see Section 13.4.

To select the TMDTO parameters (like the length S of the rainbow sequence, the number d of bits defining the DP-criterion, the interval $I_l = [t_{min}, t_{max}]$ defining the minimum and maximum number of rainbow sequence applications as well as the number m of start points) requires special attention, since this highly influences the precomputation time (PT), the disk usage (M), as well as the time needed in the online phase for the chain computations (T), the number of table accesses (TA) and the success rate (P_{total}). Table 13.1 summarizes the results for different sets of parameter choices. In all cases we consider generation of just one table ($r = 1$). The estimations are based on the analysis presented in Subsection 12.5.4 under the assumption that $D = 64$. Furthermore, we assumed that COPACOBANA is used also for the online phase. Due to this, it is worth trading higher online complexity, e.g., for lower demand for disk space (compare rows 4 and 5).

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

13.2 Fast Sort of Disk Stored TMTO Tables

The generated TMDTO table has to be sorted according to the end points. The average disk access time disables implementation of any "random access" sorting algorithm like QuickSort, HeapSort etc. on the TMDTO table stored

Table 13.1: A5/1 TMDTO: Expected runtimes and memory requirements

#	m	S	d	I_l	PT [days]	M [TB]	T [secs]	TA	P_{total}
1	2^{41}	2^{15}	5	$[2^3,2^6]$	337.5	7.49	70.5	2^{21}	0.86
2	2^{39}	2^{15}	5	$[2^3,2^7]$	95.4	3.25	92.0	2^{21}	0.67
3	2^{40}	2^{14}	5	$[2^4,2^7]$	95.4	4.85	27.6	2^{20}	0.63
4	2^{40}	2^{14}	5	$[2^3,2^6]$	84.4	7.04	17.7	2^{20}	0.60
5	2^{39}	2^{15}	5	$[2^3,2^6]$	84.4	3.48	70.5	2^{21}	0.60
6	2^{40}	2^{14}	5	$[2^4,2^6]$	84.4	5.06	21.5	2^{20}	0.55
7	2^{37}	2^{15}	6	$[2^4,2^8]$	47.7	0.79	186.3	2^{21}	0.42
8	2^{36}	2^{16}	6	$[2^4,2^8]$	47.7	0.39	745.3	2^{22}	0.42

on the disk. For example, let assume that we will sort 4.85 TB of data, representing about $2^{38.5}$ triples $\{SP, EP, l\}$ (row 3 in Table 13.1). To sort $\widehat{m} = 2^{38.5}$ data records, we need to perform about $\widehat{m} \times \log_2(\widehat{m}) = 2^{38.5} \times 38.5 \approx 2^{43.8}$ data comparisons and *transfers*. If we assume that the average access time is just 1 ms (currently stated values of the average access time are about 8 ms), then we would need about $2^{33.8}$ seconds (more than 450 years) to sort the table.

Standard *RadixSort* [Knu98], sometimes also denoted as LSD (least significant digit) RadixSort, is more suitable in this case. Let R denotes the value of radix used in the algorithm. Sorting in RadixSort consists of several *passes* (iterations) during which all data is transfered from one disk to other disk. In each pass, the transfered data is distributed among R files in the target disk; each file contains all data sharing the same value of the digit in currently inspected order of their keys. The number of passes $k = \lceil \log_R(N) \rceil$ depends on the range N of values used as a key in the algorithm, and the radix R.

To minimize fragmentation of target files, which minimizes the number of disk seeks, it is worth implementing a set of buffers in a computer RAM, as shown in Figure 13.4. The sorted data is distributed among the buffers; when a buffer is full, then the whole contents of the buffer is at once transfered to the disk and the buffer is emptied. The performance of RadixSort is then limited by the disk data transfer rate, which is stated to be about $f_B = 100$ MB/s at maximum (however, we measured the transfer rate of about 50 MB/s only).

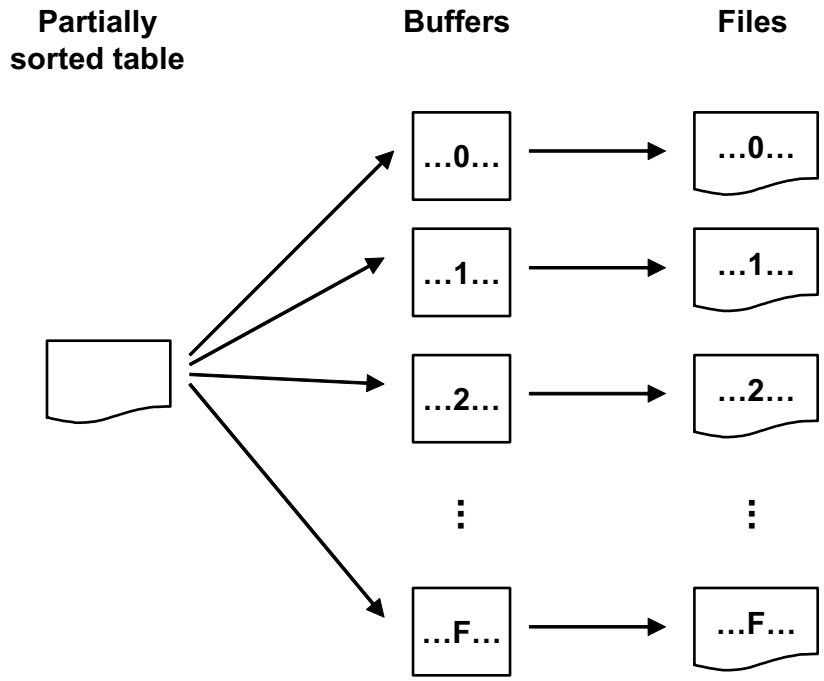

Figure 13.4: Memory buffers minimize the fragmentation of files in RadixSort

Example 13.1. Let's focus on the set of parameters presented in the third row of Table 13.1. One pass, i.e one transfer of overall data, lasts at least $T_{pass} = \frac{M}{f_B} = \frac{4.85 \text{ TB}}{100 \text{ MB/s}} = 48500$ seconds ≈ 13.5 hours in this case. However, in reality, this time would be at least four times longer (≈ 54 hours). It is for two reasons: (i) the real transfer rate is just 50 MB/s and (ii) reading from the source disk and writing to the target disk are not performed concurrently. At any time, we either read the data from the source disk and sort them among memory buffers, or we empty a memory buffer to the target disk. Concurrent reading and writing is also feasible, however, it demands for sophisticated control.

Since we use end points as the keys for the sorting algorithm, then $N = 2^{64-d} = 2^{64-5}$ in our case, where d is the number of bits defining the DP-property. If we choose $R = 16$, then the number of passes is $k = 15$, and a complete sort of the table will last at least $T_{sort} = k \times T_{pass} = 15 \times 48500 = 727500$ seconds ≈ 8.4 days. However, as discussed above, we expect this time to be at least four times longer, i.e. $T_{sort} \approx 33.7$ days. If we choose $R = 256$, then $k = 8$ and the sort time is at least 4.5 days (or 18 days).

We would like to stress that the value of radix R should be carefully chosen. Too small value of R leads to enormous number of passes k. On the other hand, too big value of R increases fragmentation of target files (hence the number of disk seeks), since the memory buffers are too small in this case.

13.2.1 Implemented Method

The speed of RadixSort algorithm is limited by the disk data transfer rate. Hence the minimization of the data volume to be transfered from/to disk represents the way to accelerate the sorting of the data stored on the disk. In our implementation we have decided to use the combination of MSD (most significant digit) RadixSort and any "RAM-efficient" standard sorting algorithm. This enables sorting the table in k passes, where $k = 3$ for typical sizes M of TMDTO tables (between 1 TB and 10 TB) and typical installed capacities of RAM (at least 1 GB).

In the first $k-1$ passes, we split the data records from the unsorted TMDTO table into n files according to the most significant digits of end points, as shown in Figure 13.5. In the first pass we split the unsorted table into $R_1 \approx \sqrt[k-1]{n}$ files according to the most significant digit of end points, in the second pass we split each of R_1 files into $R_2 \approx \sqrt[k-1]{n}$ smaller files according to the second most significant digit, etc. To minimize fragmentation of the target files we again use memory buffers shown in Figure 13.4. Since the values of end points are almost evenly distributed, the target files obtained after $(k-1)$-th pass are of similar sizes, which are $\approx \frac{M}{n}$. The number n is chosen sufficiently big to ensure that the size of each file is smaller than the size of a RAM installed in the computer. Files splitting typically cannot be performed in just one pass because the number n is relatively large (some operating systems have the limit on the number of concurrently open files; moreover, with increasing number of files the size of memory buffers is decreasing, which leads to increased number of disk seeks, etc.).

Each of n target files can be then sorted in a RAM (which is faster than the sorting via disk data transfers) using any standard sorting algorithm (RadixSort, QuickSort, HeapSort, etc.). This is done during the k-th pass — the whole files are successively loaded to the memory, sorted (using HeapSort in our case), shorter chains with duplicate end points are rejected, and the results are merged on the target disk. The time necessary to sort the TMDTO table using presented combination of MSD RadixSort and "RAM-efficient" standard sorting algorithm is equal to the time of k passes of RadixSort plus the time of RAM-efficient sorting algorithm.

Example 13.2. We will again focus on the set of parameters presented in the third row of Table 13.1. Let assume that the computer is equipped with RAM of capacity 2 GB. Then we may choose $n = 4096$, $k = 3$ and $R_1 = R_2 = 64$. The size of each file will be then $\frac{4.85 \text{ TB}}{4096} \approx 1.2$ GB.

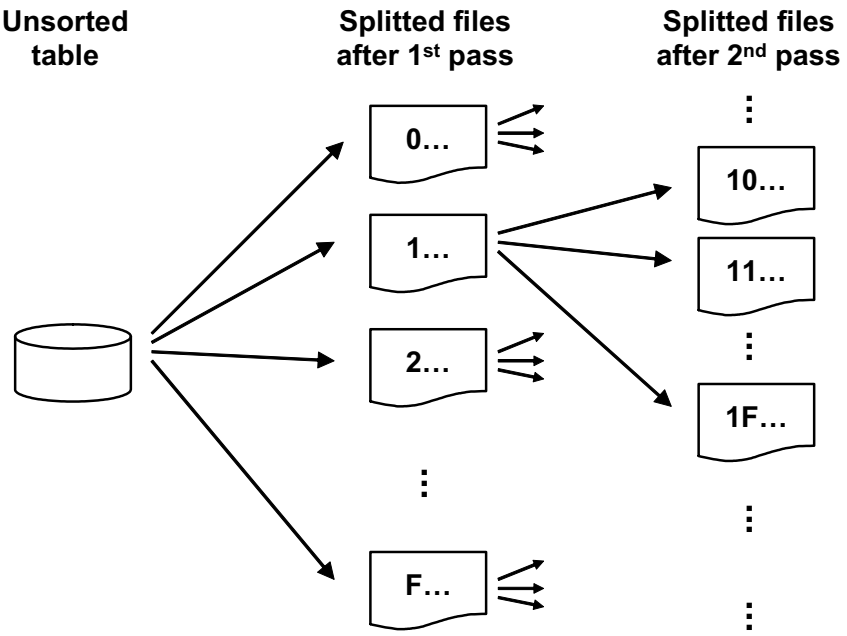

Figure 13.5: Splitting an unsorted table into files according to the MSDs of the key

The expected sort time is at least $T_{sort} = k \times T_{pass} + T_{RAMsort} = 3 \times 48500 + T_{RAMsort} = 145500$ seconds $+ T_{RAMsort} \approx 1.68$ days (or ≈ 6.73 days) $+T_{RAMsort}$, where $T_{RAMsort}$ is the time for RAM-oriented sorting algorithm.

13.3 Implementation Results — the Precomputation Phase

As mentioned in Subsection 13.1.5, chain information is encoded with the Hamming code before transportation over the COPA bus (which has 64 data lines). Moreover, some status information is added, too. Therefore, the information on triple $\{SP, EP, l\}$ of each successfully generated chain is transfered in three 64-bit words (192 bits).

For the set of parameters presented in the third row of Table 13.1, we obtain the following results: The number of chains having the lengths between 2^4 and 2^7 rainbow sequences is $m' \approx 2^{39.27}$. The precomputation time is $T_{pr} \approx 2^{22.97}$ seconds, therefore, COPA generates about $2^{16.30}$ chains per second on average, which represents the data rate of $2^{16.30} \times 192 \approx 15.5$ Mbits/s.

To connect COPACOBANA with a host computer, two versions of an interface have been developed: the USB interface and the 1 Gbit/s Ethernet one. Unfortunately, for some technical problems, none of them provides sufficient data throughput; the maximum measured data rate of the USB interface is about 10 kbits/s, and the maximum measured data rate of the Ethernet interface is about 300 kbits/s. Since the technical problems do not seem to be solvable (the problems are likely caused by a third-party hardware and/or software modules), a new interface is currently under development.

To partially compensate limited throughput of the interface, we have decided to use a set of parameters presented in the eighth row of Table 13.1 for our initial implementation. A prolonged rainbow sequence ($S = 2^{16}$) and a higher value of d ensure the generation of longer chains, which in turn represents lower demand for transfered data. Concretely, during $T_{pr} \approx 2^{21.97}$ seconds COPA generates $m' \approx 2^{35.63}$ chains having the length between 2^4 and 2^8 rainbow sequences, which represents the demand for the data rate of $\frac{2^{35.63}}{2^{21.97}} \times 192 \approx 2.5$ Mbits/s. This is still well beyond the possibilities of the interface, however, the ratio between captured data and all generated data increases from about 2 % to about 12 %. On the other hand, for this improvement we pay by an increased number of table accesses in the online phase.

We used two COPACOBANAs, COPA1 and COPA2, for table precomputation. Both machines were running for a couple of months. During this time we have successfully collected 14117445263 $\approx 2^{33.717}$ chains generated by COPA1 and 11772717296 $\approx 2^{33.455}$ chains generated by COPA2 (altogether 25890162559 $\approx 2^{34.592}$ chains). The chains were sorted according to their end points, and in cases of duplicate end points, the shorter chains were sorted out. We would like to stress that all chains acquired from COPACOBANAs have their length in the range of $\left[2^4, 2^8\right]$ rainbow sequences; hence the number of them corresponds to number m'. Chains having their length out of this range are discarded inside COPACOBANAs. Let us note that it is impossible to verify the correctness of Equation 12.11, as many chains were lost due to the slow communication interface. Concretely, according to Equation 12.11, the number of chains generated in COPA2 will be $m \approx 11772717296/0.772 \approx 15252470254 \approx 2^{33.828}$ chains. However, during its running time COPA2 generated $m = 107305012412 \approx 2^{36.643}$ chains, which is about 7 times more.

An average chain length calculated by Equation 12.10 is $l_{avg} \approx 73.459$ rainbow sequences. An average length of chains acquired from COPA2 was $l_{avg} = 73.389$ rainbow sequences — it differs from the theoretical value by

$\approx 0.1\%$. Unfortunately, we have not measured this length in the set of chains acquired from COPA1.

As discussed in Section 12.2, the average chain length l'_{avg} decreases after rejection of chains with duplicate end points. To investigate the dependency of l'_{avg} on the number $\widehat{m}$ of chains with unique end points, and also to investigate the dependency of $\widehat{m}$ on the number of generated chains m', we split the data generated by COPA2 into 10 sets 0, 1, ... 9 of volumes 2^{25}, 2^{25}, 2^{26}, 2^{27}, 2^{28}, 2^{29}, 2^{30}, 2^{31}, 2^{32} and $2^{31.568}$ chains. Each set was separately sorted, the shorter chains with duplicate end points were rejected, and both the number of chains $\widehat{m}$ and their average length l'_{avg} were measured. Then the sets were successively merged, until all sets were united. After each merge shorter chains with duplicate end points were again rejected and the values $\widehat{m}$ and l'_{avg} were measured. Since some unions of sets have the same volumes m' as some single sets (e.g. the union of sets 0 and 1 has the same volume as the set 2), we obtain multiple measurements for some volumes m'. Finally, we added the set generated by COPA1.

All results are summarized in Table 13.2. As visible from Table 13.2 and from Figure 13.6, the average chain length l'_{avg} is decreasing with growing $\widehat{m}$, as has been discussed in Subsection 12.2.3.

Table 13.2 brings also theoretical values of $\widehat{m}$, calculated by iterative Equation 12.12. As evident from Table 13.2, as well as from the chart in Figure 13.7, the real measured values of $\widehat{m}$ are smaller than the theoretical ones. This is mainly caused by the difference of assumptions under which Equation 12.12 has been derived, and a real behavior of the step functions f together with the real operating conditions.

Equation 12.12 is based on the assumption that the step function f_i has probabilistic behavior, i.e. every data point can be produced with the same probability. However, this does not conform to the real behavior of the step function composed from the encryption function g of the A5/1 cipher and the re-randomization function R_i. The main problem lies in the non-injective and non-surjective behavior of the A5/1 cipher [Gol97] (see also Chapter 14), which reduces the set of points generated by the step function f_i.

We also considered all start points to be randomly chosen, or, more precisely, we considered the choice of start points not to be influencing the number $\widehat{m}$. However, in our implementation, the start points are generated as a sequence of consecutive values. This fact, together with the non-injectivity of A5/1 cipher, causes (at least) $\frac{1}{8}$ of generated chains to be merging another chain

Table 13.2: Theoretical and measured values of the number of chains in the table ($\widehat{m}$) and their average length (l'_{avg}) after rejection of chains with duplicate end points. Values presented here are for different volumes of sets m' (sets with chains having their length in range $\left[2^4, 2^8\right]$ rainbow sequences).

			Measured values			Theoretical values	
	sets	m'	l'_{avg}	$\widehat{m}$	$\frac{\widehat{m}}{m'}$	$\widehat{m}$	$\frac{\widehat{m}}{m'}$
COPA2	0	2^{25}	73.383	$2^{24.828}$	0.887	2^{25}	1
	1	2^{25}	73.424	$2^{24.836}$	0.893	2^{25}	1
	0–1	2^{26}	73.399	$2^{25.829}$	0.888	2^{26}	1
	2	2^{26}	73.408	$2^{25.878}$	0.919	2^{26}	1
	3	2^{27}	73.39	$2^{26.939}$	0.959	2^{27}	1
	2–3	$2^{27.585}$	73.382	$2^{27.502}$	0.944	$2^{27.582}$	0.998
	0–3	2^{28}	73.372	$2^{27.893}$	0.928	$2^{27.997}$	0.998
	4	2^{28}	73.381	$2^{27.95}$	0.966	$2^{27.997}$	0.998
	5	2^{29}	73.33	$2^{28.914}$	0.942	$2^{28.992}$	0.995
	4–5	$2^{29.585}$	73.292	$2^{29.502}$	0.944	$2^{29.574}$	0.992
	6	2^{30}	73.226	$2^{29.903}$	0.935	$2^{29.985}$	0.99
	7	2^{31}	73.064	$2^{30.887}$	0.925	$2^{30.97}$	0.979
	9	$2^{31.568}$	72.915	$2^{31.432}$	0.91	$2^{31.523}$	0.97
	6–7	$2^{31.585}$	72.904	$2^{31.446}$	0.908	$2^{31.54}$	0.969
	4–7	$2^{31.907}$	72.79	$2^{31.752}$	0.898	$2^{31.851}$	0.962
	0–7	2^{32}	72.752	$2^{31.837}$	0.893	$2^{31.94}$	0.959
	8	2^{32}	72.743	$2^{31.84}$	0.895	$2^{31.94}$	0.959
	8–9	$2^{32.8}$	72.29	$2^{32.578}$	0.857	$2^{32.696}$	0.931
	0–9	$2^{33.455}$	71.708	$2^{33.154}$	0.812	$2^{33.293}$	0.894
COPA1		$2^{33.717}$	71.329	$2^{33.351}$	0.776	$2^{33.523}$	0.874
COPA1 & COPA2		$2^{34.592}$	70.043	$2^{34.049}$	0.687	$2^{34.243}$	0.785

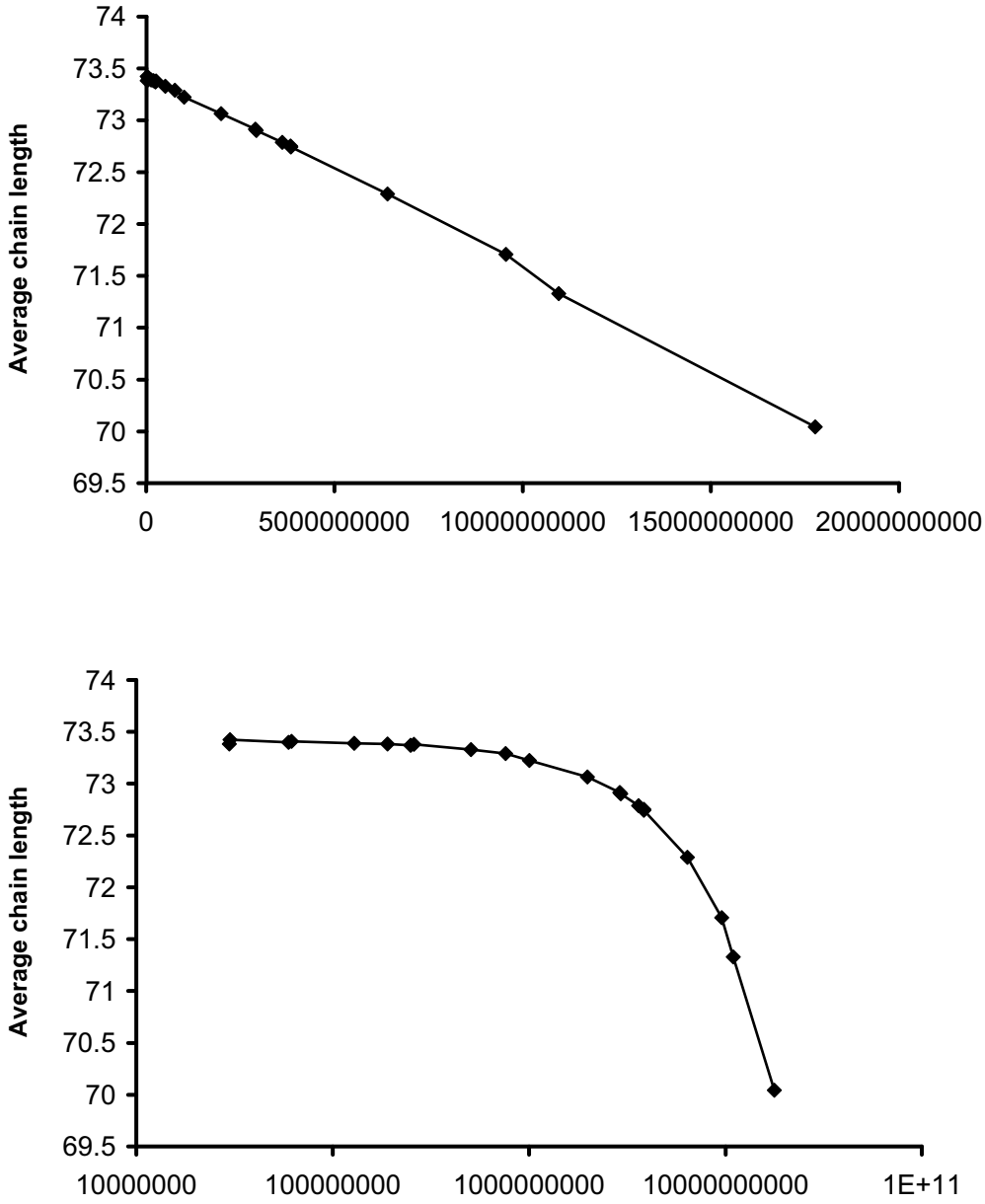

Figure 13.6: The average length of the chain after the rejection of the duplicate end points is decreasing with growing number of chains a) linear scale b) semi-logarithmic scale.

after the first step function, as shown below. Other chain merges occur due to other reasons.

13.3.1 Chains Merging One Step after the Start Point

It is easy to find two internal states (data points) x that produce the same keystream y. If such two data points appear in the same column of the TMDTO table, then their chains inevitably merge in the next step. Figure 13.8 gives an example of such two internal states, x_a and x_b. Both states have the same successor, the internal state x_c. While in the state x_a all three registers are clocked, in the state x_b only registers $R2$ and $R3$ are clocked and the register $R1$ is stopped.

The states are constructed in a following way: The internal state x_a is a state, where

$$R3_a[10] = R2_a[10] = R1_a[8] \neq R1_a[7] \text{ (clocking bits)}. \tag{13.1}$$

The internal state x_b is constructed from the state x_a by setting

$$\begin{aligned} R3_b &= R3_a, \\ R2_b &= R2_a, \\ R1_b &= \text{“clocked” } R1_a, \end{aligned} \tag{13.2}$$

i.e. $R1_b[i] = R1_a[i-1], 1 \leq i \leq 18$ and $R1_b[0] = R1_a[18] \oplus R1_a[17] \oplus R1_a[16] \oplus R1_a[13]$. This ensures that both states have the same successor x_c, and, therefore, starting from the second bits, the keystreams y_a and y_b are inevitably identical .

To ensure also the equality of the first bits of the keystreams y_a and y_b, $y_a[0] = y_b[0]$, the most significant bits of $R1_a$ and $R1_b$ have to be equal,

$$R1_a[18] = R1_b[18] \equiv R1_a[17]. \tag{13.3}$$

Every 16^{th} internal state satisfies conditions defined in Equations 13.1 and 13.3, i.e. for $\frac{1}{16}$ of all internal states we can construct the “partner” states (x_b) by the above described method.

In our implementation, the internal state x is represented as a concatenation of the registers $R3\&R2\&R1$. Since the start points are generated as a sequence of consecutive values, then, according to Equation 13.2, the pairs $\{x_a, x_b\}$ are

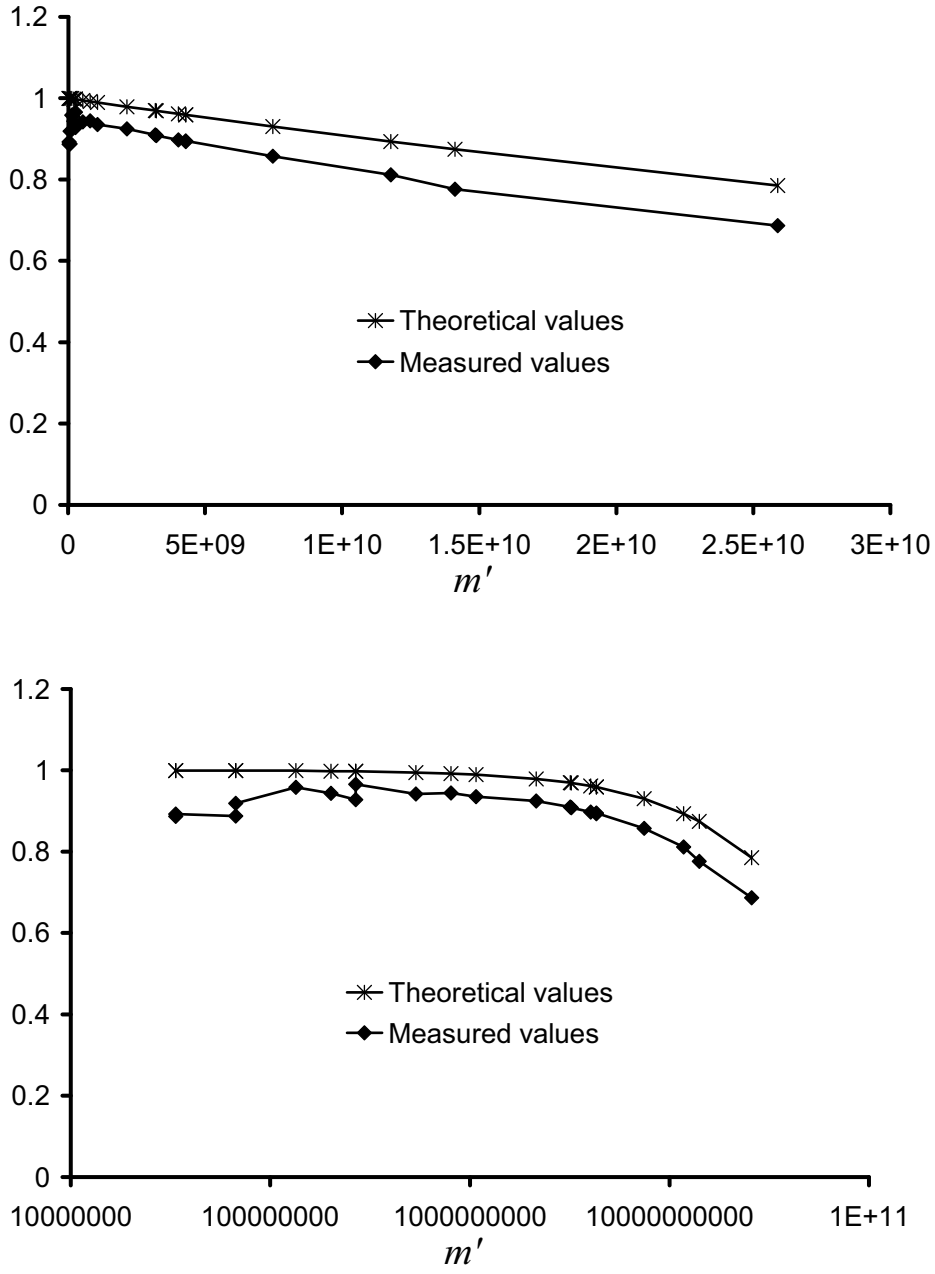

Figure 13.7: The ratio between the number of chains after rejection of the duplicate end points, and the number of generated chains a) linear scale b) semi-logarithmic scale.

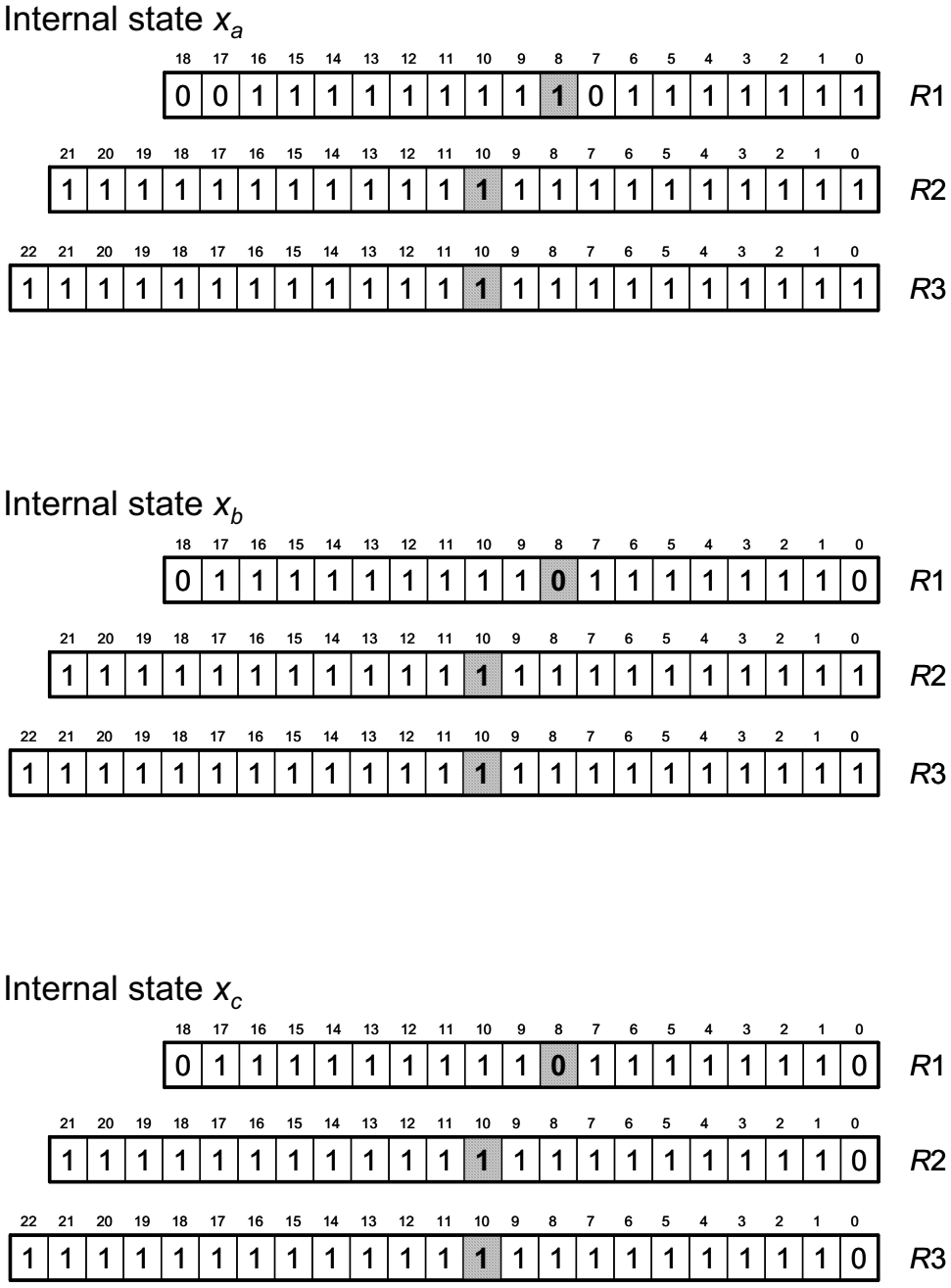

Figure 13.8: Both internal state x_a and internal state x_b have the same successor — internal state x_c. Both x_a and x_b produce the same output keystream, $y_a = y_b$.

generated even in a relatively small sets of start points. In other words, even in relatively small sets of start points, about $\frac{1}{16}$ of generated chains are rejected due to the above described *deterministic* merge with another chain.

Above described merges may be prevented by smarter generation of start points. For example, since the host computer generates upper 40 bits of start points (by doing this the host computers defines a region of start points to be generated in particular FPGA), it may be forced to generate only such values, where $R3_a[10] \neq R2_a[10]$. Such start points then violate the condition defined in 13.1. However, many other reasons for chain merges would remain.

13.4 Online Engine

As discussed in Subsection 12.5.3, in the online phase we have to compute S chains for each data point y_i. Each chain consists of rainbow sequences, where the first rainbow sequence is incomplete. It starts with the re-randomization function R_j, where $1 \leq j \leq S$, and follows with the step functions $f_{j+1}f_{j+2}\ldots f_S$. The following rainbow sequences are of the standard format $f_1 f_2 \ldots f_S$. The chain is generated until either DP is reached, or the number of rainbow sequences exceeds the maximum length of chain t_{max}.

For calculation of results during the online phase, we use an engine similar to the table precomputation engine presented in Figure 13.3. To handle the first, irregular, rainbow sequence, we had to make several modifications described below.

13.4.1 Online TMTO element

Each chain is again calculated in a dedicated online TMTO element, depicted in Figure 13.9. The online TMTO element is equipped with a *step counter*, which controls the generation of the first rainbow sequence. As the re-randomization function generator distributes the uniform sequence of functions $R_1 R_2 \ldots R_S$ to all TMTO elements, the main task of the step counter is to start the generation of the chain at the moment when the re-randomization function generator supplies the function R_j.

Before the element starts calculation of a new chain, the Core #1 is loaded with the data point y_i and the step counter is loaded with the (initial) counter value j. Then the rainbow sequence lasting S steps begins. The step counter keeps signals *re-randomize* and *full-step* inactive during the first $j-1$ steps of

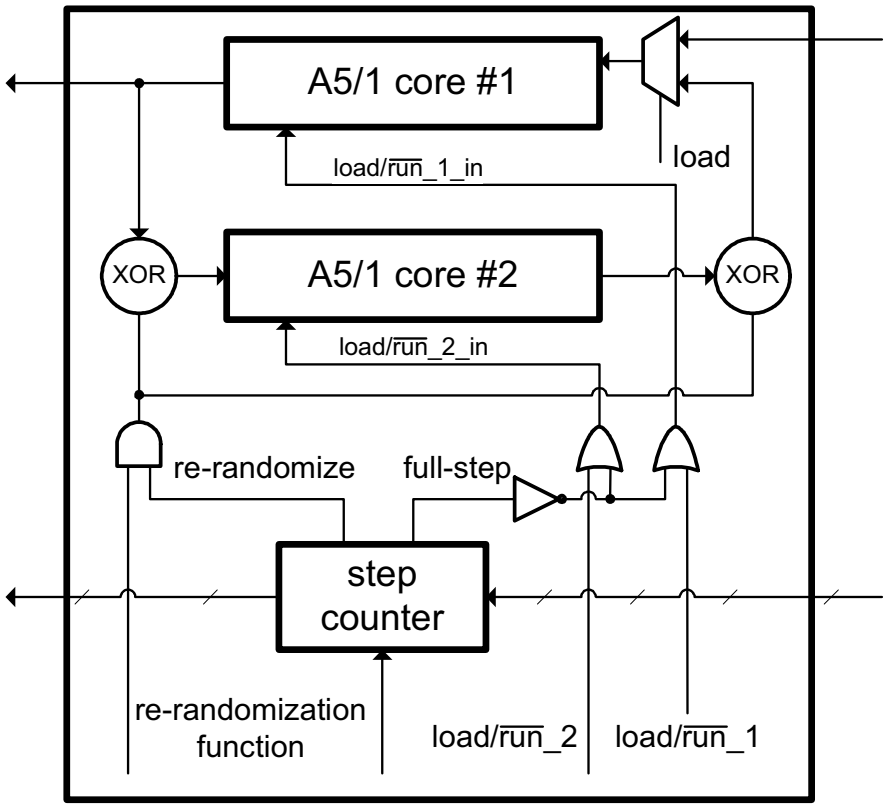

Figure 13.9: Architecture of the online TMTO element

the first rainbow sequence. Therefore, Cores #1 and #2 are both in the LOAD mode, and re-randomization is switched-off. The data point y_i is swapped between both Cores without any change. Then, in j-th step, the signal *re-randomize* is activated, and the re-randomization function R_j is applied to the data point y_i. In the next, $(j+1)$-th step, also the signal *full-step* is activated. From that moment, the function of the online TMTO element is equivalent to the function of the offline one.

13.4.2 Architecture of the A5/1 Online Engine

Each FPGA hosts one A5/1 online engine; its architecture is depicted in Figure 13.10. The task of the engine is to calculate q chains for a given data point y_i. At the beginning, the engine is by the host computer assigned the data point y_i and the interval of counter values $[j_{min}, j_{max}]$, where $j_{max} = j_{min} + q$ and $1 \leq j_{min} < j_{max} \leq S$. Then the chains having their first re-randomization functions in range $R_{j_{min}} \ldots R_{j_{max}}$ are generated. When finished, the online engine is assigned the new data point and the new interval.

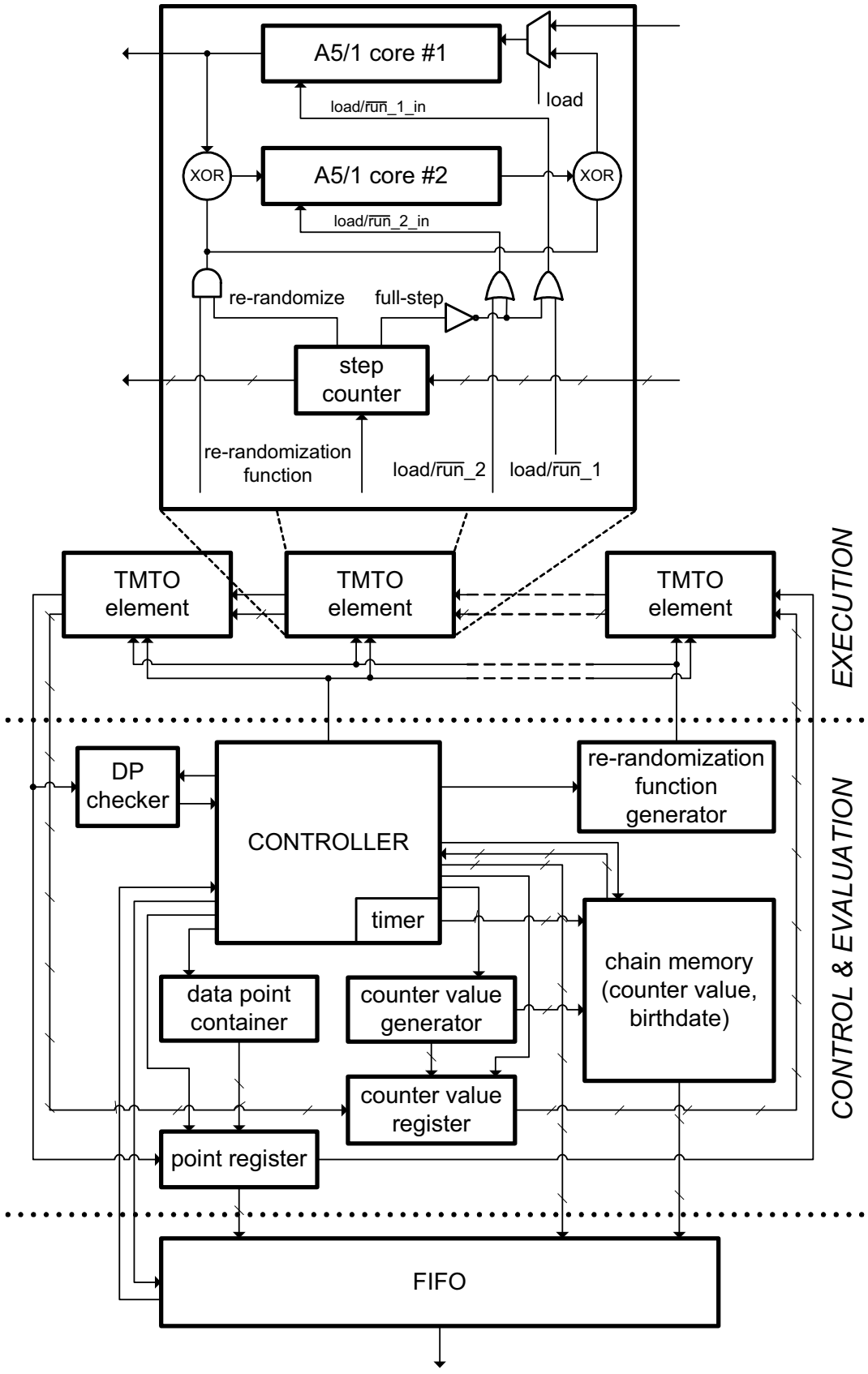

Figure 13.10: Architecture of an A5/1 online engine

There are several differences in the architecture of the online engine in comparison to the architecture of the table precomputation engine (Figure 13.3). The start point generator is replaced with the *data point container*, which holds the value of the data point y_i. A *counter value generator*, which is a simple counter, generates all values of j from the interval $[j_{min}, j_{max}]$. Upon generation, those values are loaded to a *counter value register* and shifted to the step counter inside corresponding TMTO element. The chain memory does not hold values of start points. Instead, the counter values j are stored.

After the reset, the controller runs the *initialization phase*, during which all TMTO elements are initialized with the value of data point y_i and the counter values j. This phase is run exactly once. After that, during standard operation, phases of *execution* are alternated with phases of *evaluation*.

Initialization. During the initialization, the TMTO elements are switched to build, together with point register, one large shift register. Data point y_i is repeatedly loaded to the point register and shifted to Cores #1 of all TMTO elements. Concurrently, the counter values j are generated and loaded to the counter value register to be shifted to the step counters of corresponding TMTO elements. The 'birthdate' and the counter value j of each individual chain are stored in the chain memory. Besides that, the timer, the re-randomization function generator and some other auxiliary circuits are initialized.

Execution. Execution itself is performed in TMTO elements. One phase of execution lasts exactly one rainbow sequence. The function has been already described in Subsection 13.4.1.

Evaluation. When the rainbow sequence is finished, all TMTO elements are again switched into one large shift register and the results are shifted-out. The results are on-the-fly checked for the DP-property, and the length of each chain is calculated as a difference between a 'current date' (which is the current value of the timer) and the 'birthdate' of the corresponding chain, which is stored in the chain memory. For each chain, one of following three cases may appear:

1. If the result satisfies the DP-property (it is the *end point*) and if the length l of the chain does not exceed the maximum length of chain, $l \leq t_{max}$, then the chain information (end point, length, counter value)

is stored into FIFO — the value of the end point is acquired from the point register and the counter value is acquired from the chain memory.

Concurrently, the end point, held in the point register, is replaced with the data point y_i, and a new counter value is generated and loaded to the counter value register. At the same moment both the new counter value and the 'birthdate' are stored in the chain memory.

Information stored in FIFO is later read out by the host computer and stored on the disk.

2. If the chain becomes too large, $l > t_{max}$, then the chain is discarded and the generation of a new chain is initiated in the same TMTO element — the procedure is the same as above, but the chain information is not stored in FIFO.

3. In all other cases (i.e. the end point has not been achieved and the chain is not too long yet) the result is simply shifted back to the TMTO element to continue with the next rainbow sequence.

During the phase of evaluation we also initialize the re-randomization function generator to generate the same rainbow sequence in the next phase of execution.

Unlike the precomputation phase, the loss of any chain information would be harmful in the online phase. Therefore, if the FIFO becomes full, then the phase of evaluation is interrupted until some space in FIFO is released. The data transfered to the host computer are again encoded with the Hamming code, however, in this case we use it as a SEC-DED code.

13.4.3 Implementation Results

In comparison to the precomputation TMTO element, the online TMTO element occupies larger area , as it is extended by the step counter. Therefore, we were able to place only 160 online TMTO elements on one FPGA. For a higher complexity of the design, the maximum achievable frequency was only 80 MHz. If the number of online TMTO elements was decreased to just 120 of them, the maximum achievable frequency was increased to 120 MHz, since synthesis and implementation tools were able to better utilize some optimization techniques (register replication, better routing etc.). We have selected the second implementation, as it has 12.5% higher performance than the first one.

Let's summarize: the A5/1 online engine can run at a maximum frequency of 120 MHz. Computing the step-function f_i takes 64 clock cycles. One FPGA contains 120 TMTO elements, the whole COPACOBANA contains 120 FPGAs. Therefore, the whole COPACOBANA can perform

$$\frac{120 \times 120 \times 120 \cdot 10^6}{64} \approx 2^{34.65}$$

step-functions per second in the online phase. Values of online time T for different sets of parameters are summarized in Table 13.1.

13.5 Fast Search at Disk-Stored TMTO Tables

Distinguished points calculated by COPACOBANA in the online phase are looked-up in the TMDTO table. If the table is stored on the disk, then simple binary search method cannot be used, as it needs several tens of hours.

Example 13.3. For example, upon selecting parameters presented in the third row of Table 13.1, we generate the TMDTO table containing information on $\widehat{m} = 2^{38.52}$ chains. Binary search needs 39 table accesses to accomplish one look-up in such a table. Since the disk access lasts about 8 ms on average, we would need up to $2^{20} \times 39 \times 8$ ms ≈ 327156 s $\approx$ 91 hours to look-up 2^{20} entries calculated in the online phase.

This approach takes even longer time than simple reading of the whole table. As the table occupies $M = 4.85$ TB and the disk data rate is stated to be $f_B = 100$ MB/s, the whole table may be read (and data checked for the presence of any calculated result) in just $\frac{4850000 \text{ MB}}{100 \text{ MB/s}} = 48500$ s ≈ 13.5 hours.

To improve look-ups, we use a method discussed in [Bal08]. The TMDTO table is divided into "sectors" of equal length. The values of the end points (keys) at the sector borders are stored in a separate table, as shown in Figure 13.11. The length of the sector is chosen to be as small as possible; it is limited only by maximum possible number of sectors, as the table containing border points must fit into RAM installed in the computer.

Before starting search of entries calculated by COPACOBANA, the table containing border points is loaded from the disk to RAM. For each entry, we first run the binary search in the RAM table to find in which sector the entry is potentially found. Then the whole sector is loaded from the disk into RAM

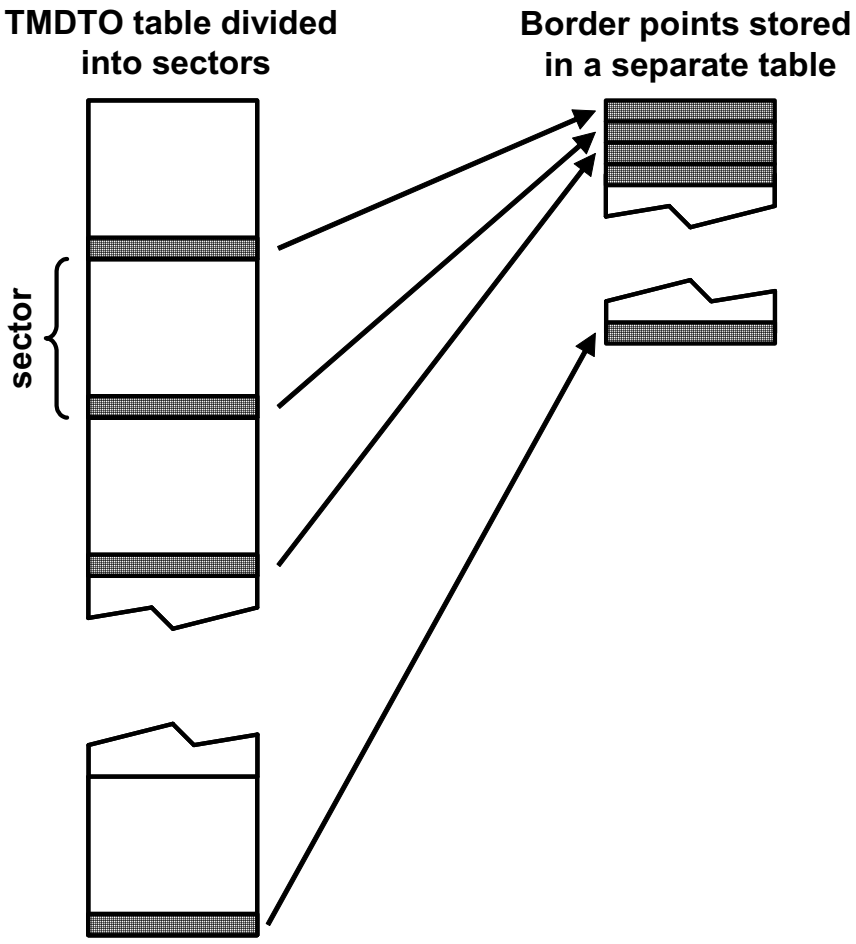

Figure 13.11: The TMDTO table is divided into sectors. Border points are stored in a separate table.

and the search is finalized. Since the sector is relatively small, loading the whole sector takes shorter time than one disk access. Binary search performed in RAM is also very fast. Therefore, time for one table look-up is almost equal to one disk access.

To further accelerate search, we sort entries calculated by COPACOBANA prior to their search in TMDTO table. This improvement reduces an average disk seek time, since the disk heads are moved only in one direction and for shorter distances. Concretely, we measured an average access time to be about 12 ms when the entries were unsorted. This time was reduced to about 4 ms after sorting the entries.

Example 13.4. Let's assume that we have a computer equipped with 16 GB of RAM. Such memory may hold at least 2^{31} end points. Then the TMDTO table presented in the third row of Table 13.1 may be divided into the same number of sectors, each being 2.425 kB long. Searching all 2^{20} entries then lasts about $2^{20} \times 4$ ms ≈ 4194 s ≈ 1.2 hour.

To further reduce the search time, the TMDTO table may be divided among N disks to perform the search in parallel. For example, if $N = 10$, then for above Example 13.4 the search time is reduced to about 7 minutes.

Nevertheless, we think that this Section will be soon outdated. Growing capacities and dramatically dropping prices of Flash memories inevitably lead to a replacement of the disks with this kind of storage elements. For example, 1 GB of Flash memory can be currently (January 2009) purchased for less than 2 €. Therefore, the Flash memory of a total capacity 4.85 TB (the third row of Table 13.1) may be currently purchased for less than 10,000 €, which is feasible not only for government agencies, but also for many other institutions and private persons. The same capacity at the disks may be currently purchased for less than 500 €. In contrast to the disks, the Flash memories provide significantly faster random access, which allows efficient implementation of the binary search. On the other hand, they suffer from frequent rewriting its content, which is not, however, our case.

13.6 Summary and Final Remarks

In this Chapter we presented a time-memory-data trade-off attack on A5/1 cipher. The attack accepts known keystream, revealing, with a certain probability, the internal state producing such a keystream. The attack is supported

by a low-cost special-purpose hardware device COPACOBANA in both the precomputation and online phases. Upon the analysis described in Chapter 12, we have selected the thin-rainbow method to be used for our attack.

When designing the precomputation engine we employed the features of the underlying FPGA architecture to gain maximum performance of the engine. The architecture approach that we used might be reused when designing similar engines for cryptanalysis of stream ciphers. COPACOBANA is able to perform 2^{36} step functions per second in a precomputation phase — each one of 120 FPGAs performs about 2^{29} step functions per second. To compare, a current PC is able to perform just 2^{22} step-function per second [BBK06]. When assuming purchase price of COPACOBANA to be 10,000 €, and the purchase price of a current PC to be 200 €, we obtain about 330 times better cost-performance ratio in favor of COPACOBANA. We get even better comparison when assuming variable costs represented mainly by energy consumption. For example, if we assume that fully equipped COPACOBANA consumes less than 600 W of power, and the PC consumes about 150 W of power, than we get about 4,000 times better consumption-performance ratio in favor of COPACOBANA.

Besides a careful design of the precomputation engine, we also designed the online engine. COPACOBANA is able to perform $2^{34.65}$ step-functions per second in the online phase. We also brought solutions for the problems of an efficient sort of large disk-stored tables, as well as fast look-ups in such large tables.

We have selected a set of parameters providing a reasonable precomputation time and a reasonable size of the tables, as well as a relatively small number of table accesses. For our selection of parameters, we obtain a precomputation time of 95.4 days and the memory capacity of 4.85 TB. As this is a probabilistic attack, success rate, as well as online complexity and the number of table accesses depend on the number of data samples available. If the number of data points available is $D = 64$, then the success rate is $P_{total} = 63\%$, COPACOBANA calculates all results in 27.6 seconds, and the results are then looked-up in 10 disks in less than 7 minutes. If the number of data points increases to $D = 204$, then the success rate is $P_{total} = 96\%$, however, also the time for calculations in COPACOBANA increases to 88 seconds and the time for table look-ups in 10 disks increases to about 22.3 minutes.

Due to a reduced communication bandwidth caused by technical problems with the interface, we decided to use another set of parameters for our initial implementation. We have run a series of measurements for data sets of different sizes. The measurements confirm that, as predicted, the average chain

length l'_{avg} decreases after the rejection of (shorter) chains with duplicate end points. This decrease is stronger with a larger set of chains. We also measured how the number of (chains with) unique end points $\widehat{m}$ depends on the number of generated chains m'. As has been shown, theoretical results were too optimistic. In reality the chain merges are more frequent, which is mainly caused by non-injective behavior of A5/1 cipher. We have identified one source of merges — by careful generation of start points, this source can be eliminated.

We have run several functional tests of the online phase to verify correctness of the implementation. To evaluate the statistical behavior of the attack — mainly the success ratio — extensive tests of the online phase are necessary. We expect these tests to be significantly time consuming. For example, for the above assumed parameters we calculated the time necessary for table look-ups to be about 22.3 minutes. Therefore, running table look-ups for 1000 instances of this attack is expected to last about 15.5 days.

Similarly to the attack presented in Chapter 11 this attack recovers the internal state which generated some of given 64-bit data points (keystream samples). In comparison to the attack presented in Chapter 11, this recovery is done significantly faster (in a matter of seconds or minutes), on the other hand, there is certain probability that no internal state is revealed. Moreover, successful recovery of one internal state does not ensure successful attack, since, as shown, multiple distinct internal states may generate identical keystream.

To break GSM communication, the cipher needs to be tracked back from recovered state(s) to evaluate the internal state in which the cipher appeared just after loading the session key K. Backtracking of A5/1 cipher is discussed in Chapter 14.

Chapter 14

Backtracking A5/1

Internal state recovery is sufficient for breaking many stream ciphers [BS00]. The cipher can be then run forward from the revealed internal state (x_r), decrypting the remainder of the ciphertext. Since the GSM communication is divided into 114 bits long frames, only the recovery of a single internal state is not sufficient — we would be able to decrypt the remainder of one frame, while other frames would remain encrypted.

To break the whole phone call we have to find either the session key K or some internal state which is common for all frames. Such an internal state is the state just after loading the session key, x_k. In this Chapter we discuss how the recovered internal state x_r can be used for derivation of the state x_k.

In the following text the symbol x_n denotes the internal state, i.e. the contents of all three registers $R1$, $R2$ and $R3$. The symbol $x_{n+\ell}$ then denotes the internal state obtained from x_n by running the A5/1 cipher for ℓ clock cycles.

14.1 Detailed View on Algorithm of A5/1

Let's recapitulate the algorithm of A5/1 cipher. The sequence of internal states is depicted in Figure 14.1.

At the beginning of each frame, the contents of all three LFSRs $R1$, $R2$ and $R3$ is cleared. Then, the 64 bit session key K is loaded to the registers $R1$, $R2$ and $R3$; the i-th bit of the key K is XORed with LSBs of all 3 registers in the

$$0 \xrightarrow[64\ clock\ cycles]{K} x_k \xrightarrow[22\ clock\ cycles]{IV} x_i \xrightarrow[100\ clock\ cycles]{warm-up} x_w \xrightarrow[1\ clock\ cycle]{} x_e$$

Figure 14.1: Sequence of internal states in A5/1

i-th clock cycle. Let x_k be the internal state after loading the key K. Since the session key K is common for all frames of the same phone call, also the internal state x_k is common for all frames of the same phone call. Therefore, with the knowledge of x_k, the phone call can be decrypted.

The initialization phase continues with loading the initialization vector IV. Again, 22 bits of IV are successively added modulo 2 to LSBs of all 3 registers. Let x_i be the internal state after loading IV. Since the initialization vector IV is derived from the *publicly known* frame number FN, it is possible to derive x_k, whenever x_i is known — to obtain x_k from the known x_i, we need to clock registers $R1$, $R2$ and $R3$ backwards and to subtract bits of IV in a reverse order.

The initialization phase is followed by the warm-up phase, which lasts 100 clock cycles. During this phase the registers are irregularly clocked and the output is discarded. Let $x_w = x_{i+100}$ be the internal state after the warm-up phase.

In the executional phase, the registers are also clocked irregularly. In each clock cycle, the registers are clocked first and then the keystream bit is generated. Let $x_e = x_{w+1} = x_{i+101}$ be the internal state producing the first bit of the keystream, which is also the internal state associated with the first 64-bit keystream sample y_0 by the function g in Equation 12.9.

The cipher produces two 114 bits long keystreams used to encrypt uplink frame and to decrypt downlink frame. In the following we assume that the first of two keystreams is known. From 114 bits long keystream, we can derive 51 keystream samples $y_0 \dots y_{50}$, each being 64 bits long, as shown in Figure 12.3. These samples were produced from the internal states $x_e \dots x_{e+50}$. Our methods, described in Chapters 11 and 13, focus on finding $x_{e+\ell}$ for a given $y_\ell, 0 \leq \ell \leq 50$. When the state $x_{e+\ell} = x_{i+101+\ell}$ is found, we have to derive corresponding x_k or, equivalently, x_i. Upon recovering x_k the GSM communication can be decrypted.

14.2 Previous Work

The method described in [Gen08] derives x_i from x_e, however it is extendable for any $x_{e+\ell}$. From the internal state x_e Gendrullis calculates all 101 predecessor states for each register $R1$, $R2$ and $R3$. By combining all predecessor states he produces all $101^3 \approx 2^{20}$ candidates for the state x_i. From each x_i candidate, the cipher is clocked for 101 clock cycles and the internal state obtained is compared with x_e. If the match is obtained, then the valid candidate for x_i has been found.

The algorithm finds 95% of x_i candidates in $\approx 2^{20}$ steps. To find all candidates, the algorithm has to perform $101^3 \times 101 \approx 2^{27}$ steps.

In this chapter we describe the method with a maximum measured complexity of $\approx 2^{13}$ steps for x_e and $\approx 2^{14}$ steps for x_{e+50}. The method is based on the work of Golic [Gol97].

14.3 Proposed Method

The idea of the algorithm described here consists in clocking the cipher from the state $x_{e+\ell}$ backwards to the state x_i. Each internal state has up to 4 potential predecessors, therefore, when backtracking the cipher we may search a tree having up to $4^{101+\ell}$ leaves. However, this may happen only when the internal state of the cipher consists of all '1's or all '0's which is almost improbable and may be easily detected.

When searching all potential predecessors x_{n-1} of the internal state x_n we have to inspect bits $R1[9]$, $R1[8]$, $R2[11]$, $R2[10]$, $R3[11]$ and $R3[10]$ of the state x_n, since following 4 combinations of these 6 bits could create a triple of the clocking bits of the state x_{n-1}:

- $\{R1[9], R2[11], R3[11]\}$ (all 3 registers were clocked in the state x_{n-1})
- $\{R1[9], R2[11], R3[10]\}$ ($R1$ and $R2$ clocked, $R3$ stopped in the state x_{n-1})
- $\{R1[9], R2[10], R3[11]\}$ ($R1$ and $R3$ clocked, $R2$ stopped in the state x_{n-1})
- $\{R1[8], R2[11], R3[11]\}$ ($R2$ and $R3$ clocked, $R1$ stopped in the state x_{n-1})

Table 14.1: Three examples of the internal states and the candidates for their predecessors. Clocking bits are highlighted.

			Example 1	Example 2	Example 3
State x_n		$R1[9]$ $R1[8]$	0 **0**	0 **0**	0 **0**
		$R2[11]$ $R2[10]$	0 **1**	0 **0**	0 **1**
		$R3[11]$ $R3[10]$	0 **1**	1 **1**	1 **0**
Candidates for state x_{n-1}	No. 1	clock $R1$+$R2$	O.K.	O.K.	n/a
		$R1[8]$ $R1[7]$	**0** 0	**0** 0	**0** 0
		$R2[10]$ $R2[9]$	**0** 1	**0** 0	**0** 1
		$R3[11]$ $R3[10]$	0 **1**	1 **1**	1 **0**
	No. 2	clock $R1$+$R3$	O.K.	n/a	n/a
		$R1[8]$ $R1[7]$	**0** 0	**0** 0	**0** 0
		$R2[11]$ $R2[10]$	0 **1**	0 **0**	0 **1**
		$R3[10]$ $R3[9]$	**0** 1	**1** 1	**1** 0
	No. 3	clock $R2$+$R3$	n/a	n/a	n/a
		$R1[9]$ $R1[8]$	0 **0**	0 **0**	0 **0**
		$R2[10]$ $R2[9]$	**0** 1	**0** 0	**0** 1
		$R3[10]$ $R3[9]$	**0** 1	**1** 1	**1** 0
	No. 4	clock $R1$+$R2$+$R3$	O.K.	n/a	n/a
		$R1[8]$ $R1[7]$	**0** 0	**0** 0	**0** 0
		$R2[10]$ $R2[9]$	**0** 1	**0** 0	**0** 1
		$R3[10]$ $R3[9]$	**0** 1	**1** 1	**1** 0
# Candidates			3	1	0

Table 14.1 shows three examples of internal states and their potential predecessor states. Only bits $R1[9]$, $R1[8]$, $R2[11]$, $R2[10]$, $R3[11]$ and $R3[10]$ and their position in the potential predecessor states are shown. As obvious, only some predecessor states are possible (denoted by O.K.), while the other ones are impossible to occur (denoted by n/a). For example, the state x_n in Ex. 1 could not be reached from the state x_{n-1} by clocking registers $R2$ and $R3$ only (Candidate No. 3) — if we clock registers $R2$ and $R3$ back from the state x_n, we obtain the state candidate x_{n-1} having all clocking bits being '0'. Applying the clocking rule, we have to clock all 3 registers now, therefore the successor of Candidate No. 3 is different from x_n.

As obvious from our examples in Table 14.1, some internal states have only few or even no (see Ex. 3) predecessor states. We can formulate the following set of rules:

- If $R1[9] = R2[11] = R3[11]$, then the predecessor state x_{n-1} obtained from the state x_n by clocking back the registers $R1$, $R2$ and $R3$ is valid
- If $R1[9] = R2[11] \neq R3[10]$, then the predecessor state x_{n-1} obtained from the state x_n by clocking back the registers $R1$ and $R2$ only is valid
- If $R1[9] = R3[11] \neq R2[10]$, then the predecessor state x_{n-1} obtained from the state x_n by clocking back the registers $R1$ and $R3$ only is valid
- If $R2[11] = R3[11] \neq R1[8]$, then the predecessor state x_{n-1} obtained from the state x_n by clocking back the registers $R2$ and $R3$ only is valid

Tables 14.2 and 14.3 summarize all 64 combinations of bits $R1[9]$, $R1[8]$, $R2[11]$, $R2[10]$, $R3[11]$ and $R3[10]$ of the state x_n. For each combination, all valid candidates for the predecessor states are enumerated. As obvious, only for 2 combinations (all '0's and all '1's), which represent only 3.125% of all cases, we obtain 4 valid predecessors. For 6 combinations (9.375%), we get 3 predecessors, for another 6 combinations (9.375%), we get 2 predecessors, for 26 combinations (40.625%), we get only one predecessor, and 24 combinations (37.5%) do not have any valid predecessor. All 64 combinations have in total 64 predecessors, i.e. each internal state has one predecessor on average.

Based on these observations, we propose a recurrent procedure BACKWARD_STEP depicted in Algorithm 14.1. The procedure is invoked with the initial parameters $R1$, $R2$ and $R3$ representing the internal state $x_{e+\ell} = x_{i+101+\ell}$, and the parameter $DEPTH = 101 + \ell$. The procedure outputs all candidates for the state x_i. The function CLOCK_BACKWARDS used in the procedure returns the value of the register clocked back by one clock cycle.

14.4 Testing the Method for A5/1 Backtracking

To observe the statistical behavior of the method we have run the set of following tests.

Table 14.2: Predecessors of the states — part 1

State x_n (important bit values)						Candidates for the state x_{n-1} (by clocking)				# Cand.
$R1[9]$	$R1[8]$	$R2[11]$	$R2[10]$	$R3[11]$	$R3[10]$	$R1$ + $R2$	$R1$ + $R3$	$R2$ + $R3$	all	
0	0	0	0	0	0				√	1
0	0	0	0	0	1	√			√	2
0	0	0	0	1	0					0
0	0	0	0	1	1	√				1
0	0	0	1	0	0		√		√	2
0	0	0	1	0	1	√	√		√	3
0	0	0	1	1	0					0
0	0	0	1	1	1	√				1
0	0	1	0	0	0					0
0	0	1	0	0	1					0
0	0	1	0	1	0			√		1
0	0	1	0	1	1			√		1
0	0	1	1	0	0		√			1
0	0	1	1	0	1		√			1
0	0	1	1	1	0			√		1
0	0	1	1	1	1			√		1
0	1	0	0	0	0			√	√	2
0	1	0	0	0	1	√		√	√	3
0	1	0	0	1	0					0
0	1	0	0	1	1	√				1
0	1	0	1	0	0		√	√	√	3
0	1	0	1	0	1	√	√	√	√	4
0	1	0	1	1	0					0
0	1	0	1	1	1	√				1
0	1	1	0	0	0					0
0	1	1	0	0	1					0
0	1	1	0	1	0					0
0	1	1	0	1	1					0
0	1	1	1	0	0		√			1
0	1	1	1	0	1		√			1
0	1	1	1	1	0					0
0	1	1	1	1	1					0

Table 14.3: Predecessors of the states — part 2

State x_n (important bit values)						Candidates for the state x_{n-1} (by clocking)				# Cand.
$R1[9]$	$R1[8]$	$R2[11]$	$R2[10]$	$R3[11]$	$R3[10]$	$R1$ + $R2$	$R1$ + $R3$	$R2$ + $R3$	all	
1	0	0	0	0	0					0
1	0	0	0	0	1					0
1	0	0	0	1	0		√			1
1	0	0	0	1	1		√			1
1	0	0	1	0	0					0
1	0	0	1	0	1					0
1	0	0	1	1	0					0
1	0	0	1	1	1					0
1	0	1	0	0	0	√				1
1	0	1	0	0	1					0
1	0	1	0	1	0	√	√	√	√	4
1	0	1	0	1	1		√	√	√	3
1	0	1	1	0	0	√				1
1	0	1	1	0	1					0
1	0	1	1	1	0	√		√	√	3
1	0	1	1	1	1			√	√	2
1	1	0	0	0	0			√		1
1	1	0	0	0	1			√		1
1	1	0	0	1	0		√			1
1	1	0	0	1	1		√			1
1	1	0	1	0	0			√		1
1	1	0	1	0	1			√		1
1	1	0	1	1	0					0
1	1	0	1	1	1					0
1	1	1	0	0	0	√				1
1	1	1	0	0	1					0
1	1	1	0	1	0	√	√		√	3
1	1	1	0	1	1		√		√	2
1	1	1	1	0	0	√				1
1	1	1	1	0	1					0
1	1	1	1	1	0	√			√	2
1	1	1	1	1	1				√	1
TOTAL										64

Algorithm 14.1 BACKWARD_STEP(R1, R2, R3, DEPTH) — a recurrent procedure for A5/1 backtracking

Input: Internal state (registers R1, R2, R3); DEPTH (#steps until x_i)
Output: All candidates for the state x_i

1: **if** $DEPTH = 0$ **then**
2: **return** $R1, R2, R3$
3: **else**
4: $R1_{back} \leftarrow CLOCK\_BACKWARDS(R1)$
5: $R2_{back} \leftarrow CLOCK\_BACKWARDS(R2)$
6: $R3_{back} \leftarrow CLOCK\_BACKWARDS(R3)$
7: **if** $R1[9] = R2[11] = R3[11]$ **then**
8: $BACKWARD\_STEP(R1_{back}, R2_{back}, R3_{back}, DEPTH - 1)$
9: **end if**
10: **if** $R1[9] = R2[11] \neq R3[10]$ **then**
11: $BACKWARD\_STEP(R1_{back}, R2_{back}, R3, DEPTH - 1)$
12: **end if**
13: **if** $R1[9] = R3[11] \neq R2[10]$ **then**
14: $BACKWARD\_STEP(R1_{back}, R2, R3_{back}, DEPTH - 1)$
15: **end if**
16: **if** $R2[11] = R3[11] \neq R1[8]$ **then**
17: $BACKWARD\_STEP(R1, R2_{back}, R3_{back}, DEPTH - 1)$
18: **end if**
19: **end if**

14.4.1 Test 1: Clocking A5/1 Forward and Backward for 101 Clock Cycles

From randomly generated state x_i, we derived the state x_e by (forward) clocking the cipher for 101 clock cycles. Then, using the procedure BACKWARD_STEP, we clocked the cipher back to the depth of 101 clock cycles. For each individual case, we observed the number of candidates for the state x_i (one of them was the original x_i), and the number of steps to seek through the whole search tree.

The test was run for 10^8 (pseudo)random cases. An average and a maximum number of candidates for the state x_i was 13.13 and 170, respectively. A histogram of the candidates for the internal state x_i is depicted in Figure 14.2a. An average and a maximum number of steps to seek the whole search tree was 717.24 and 8019, respectively. A histogram of the number of steps is depicted in Figure 14.2b.

14.4.2 Test 2: Clocking A5/1 Forward and Backward for 151 Clock Cycles

From randomly generated state x_i, we derived the state x_{e+50} by clocking the cipher (forward) for 151 clock cycles. Then, using the procedure BACKWARD_STEP, we clocked the cipher back to the depth of 151 clock cycles. For each individual case, we observed the number of candidates for the state x_i (one of them was the original x_i), and the number of steps to seek through the whole search tree.

The test was run for 10^8 (pseudo)random cases. An average and a maximum number of candidates for the state x_i was 18.04 and 207, respectively. A histogram of the candidates for the internal state x_i is depicted in Figure 14.3a. An average and a maximum number of steps to seek the whole search tree was 1448.71 and 14885, respectively. A histogram of the number of steps is depicted in Figure 14.3b.

14.4.3 Test 3: Clocking A5/1 Backward Only for 101 Clock Cycles

From randomly generated state x_e, we clocked the cipher back to the depth of (up to) 101 clock cycles. For each individual case, we observed the number

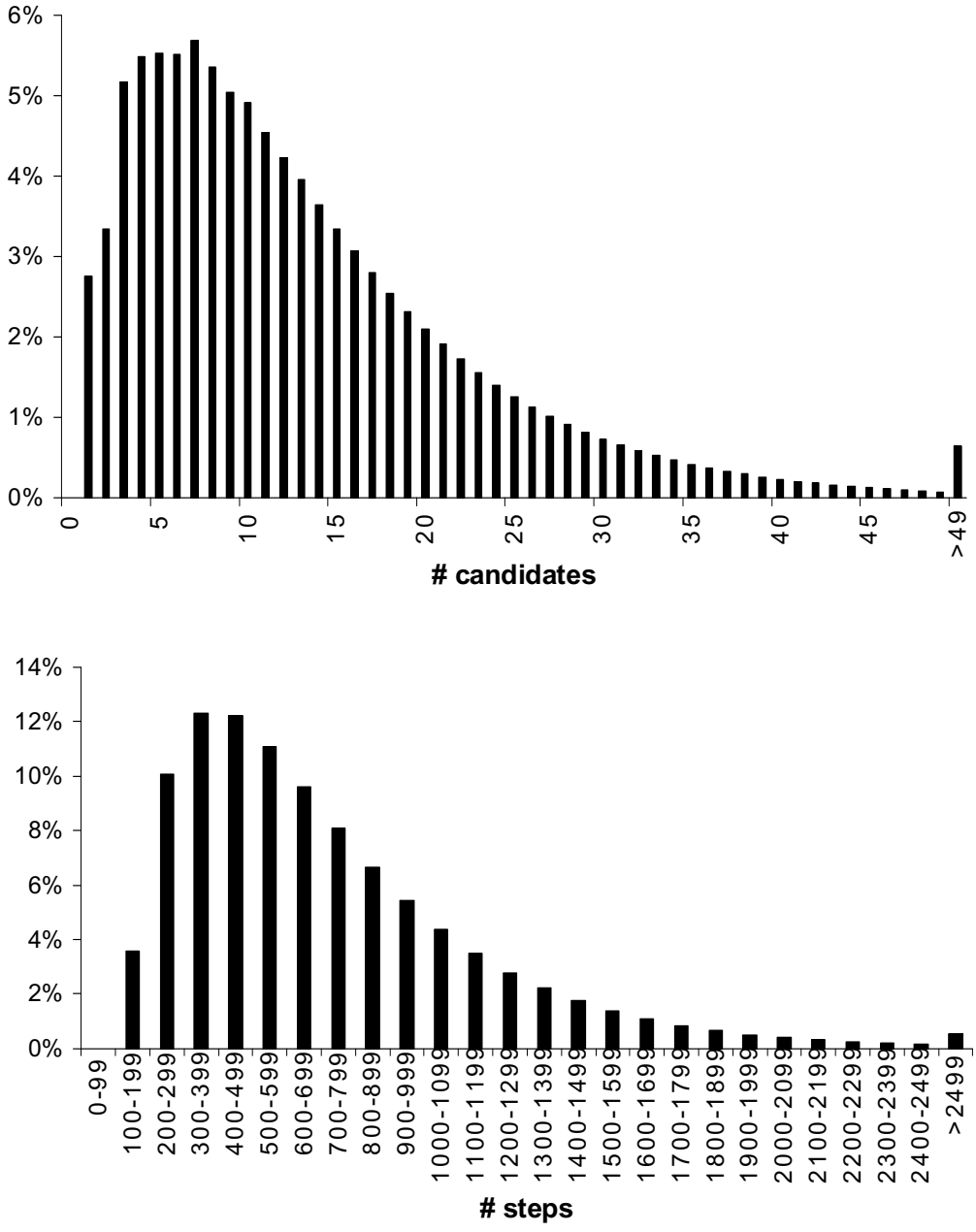

Figure 14.2: Test 1: a) Histogram of the candidates for the state x_i, b) Histogram of the number of steps to seek the whole search tree

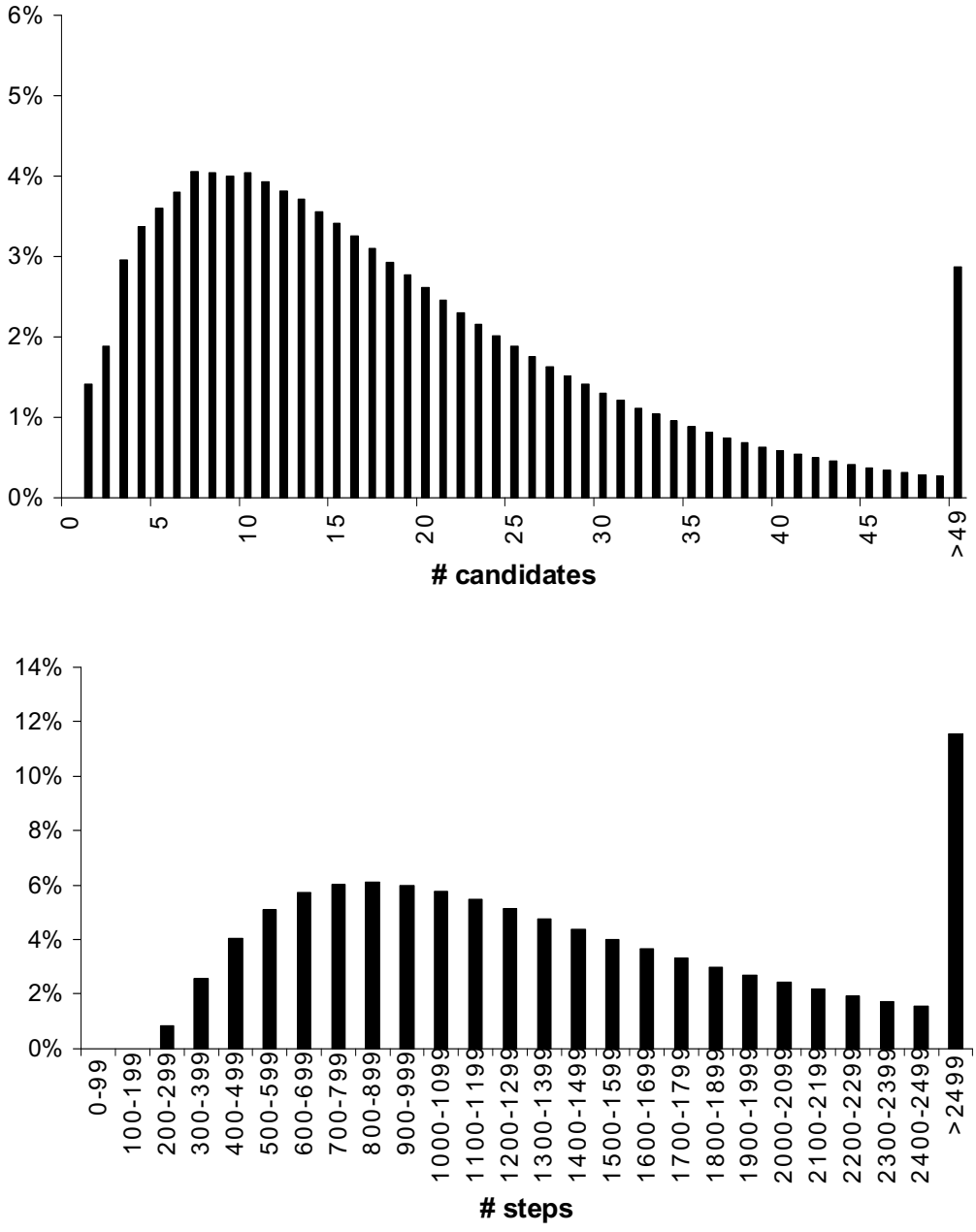

Figure 14.3: Test 2: a) Histogram of the candidates for the state x_i, b) Histogram of the number of steps to seek the whole search tree

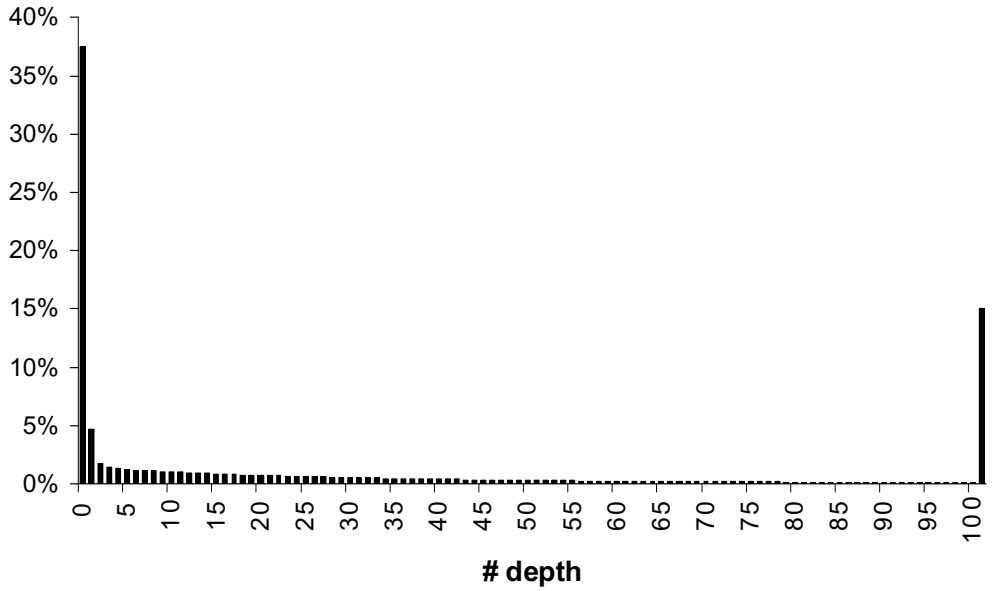

Figure 14.4: Test 3: Histogram of the maximum depth reached in the search tree

of candidates for the state x_i, the number of steps to seek through the whole search tree and the maximum depth reached in the search tree.

The test was run for 10^8 (pseudo)random cases. As obvious from Figure 14.4, only in 15% of cases the search reached the depth of 101 clock cycles and a candidate for the state x_i has been found, while other 85% cases led to no candidate (see also Figure 14.5a). It is also obvious from Figure 14.4 that in 37.5% cases the depth of the search tree was 0, which corresponds to our observations presented in Tables 14.2 and 14.3.

If at least one candidate for the state x_i has been found, then the average number of candidates was 6.65. The maximum number of candidates was 128. The average depth of the search tree was 28.45 steps.

An average number of steps to seek the whole search tree was 101.00, which conforms to observation that each internal state has one predecessor on average (see Section 14.3). The maximum number of steps to seek the whole search tree was 6518. A histogram of the number of steps is depicted in Figure 14.5b.

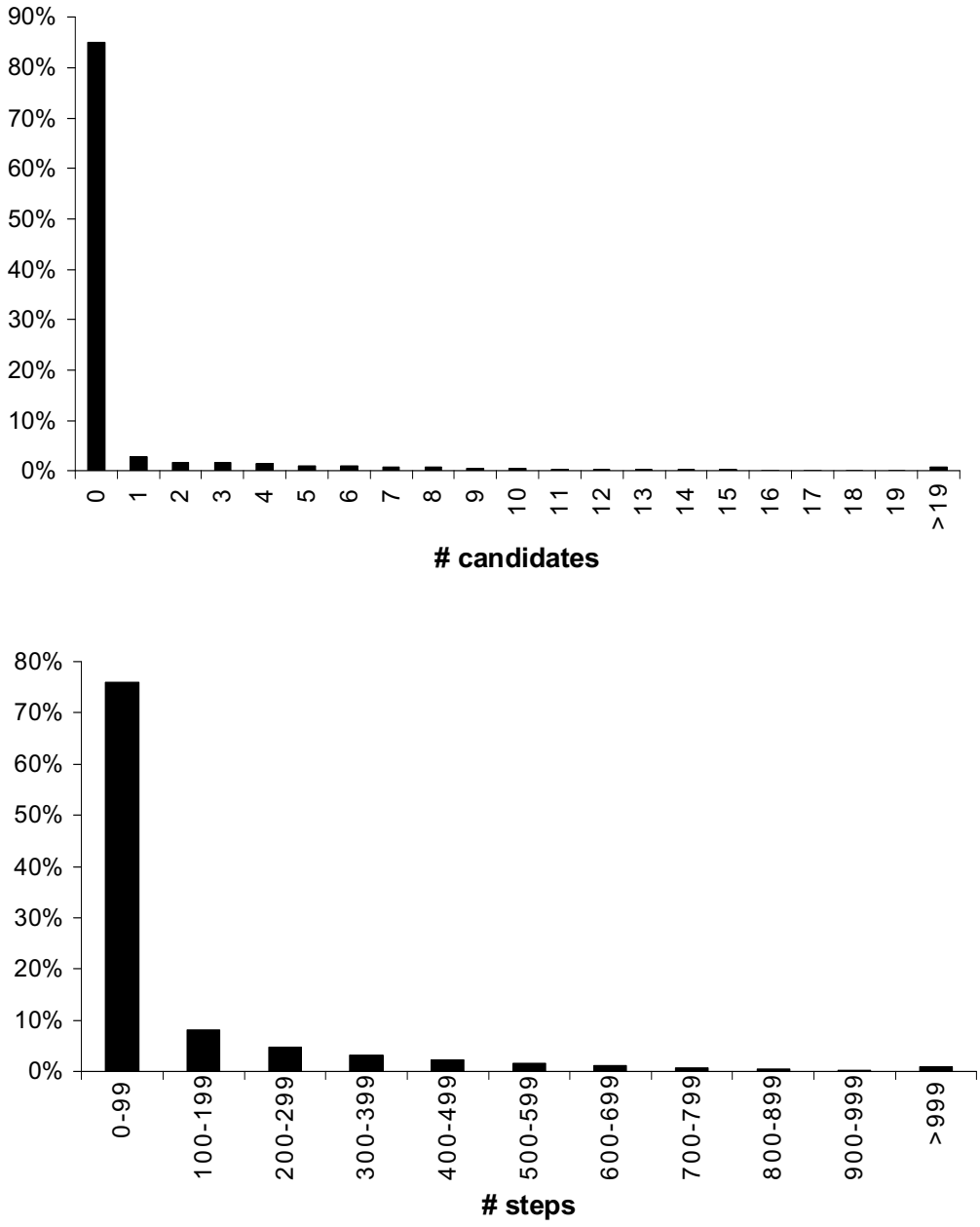

Figure 14.5: Test 3: a) Histogram of the candidates for the state x_i, b) Histogram of the number of steps to seek the whole search tree

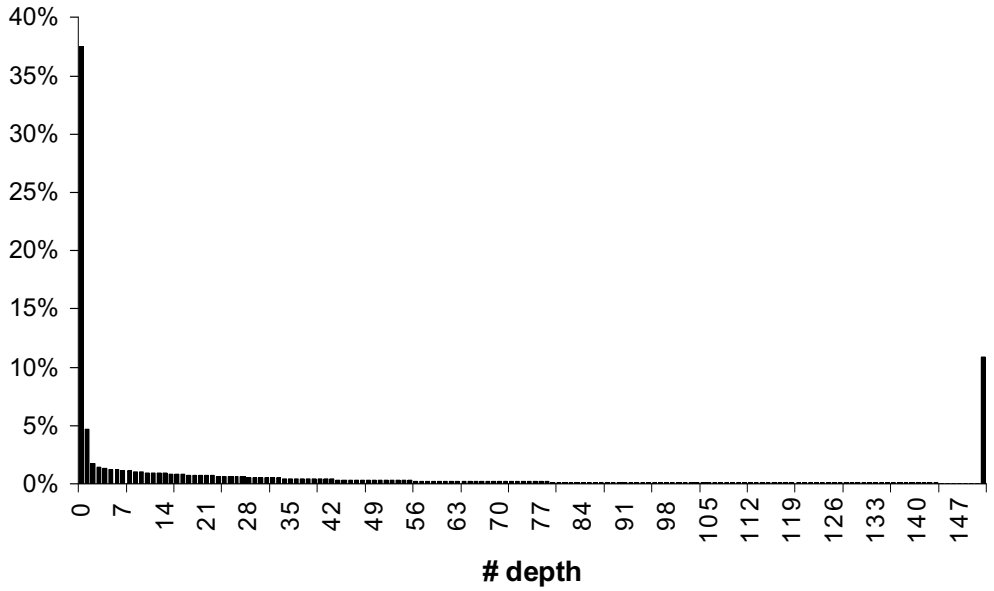

Figure 14.6: Test 4: Histogram of the maximum depth reached in the search tree

14.4.4 Test 4: Clocking A5/1 Backward Only for 151 Clock Cycles

From randomly generated state x_{e+50}, we clocked the cipher back to the depth of (up to) 151 clock cycles. For each individual case, we observed the number of candidates for the state x_i, the number of steps to seek through the whole search tree and the maximum depth reached in the search tree.

The test was run for 10^8 (pseudo)random cases. As obvious from Figure 14.6, only in 10.9% of cases the search reached the depth of 151 clock cycles and a candidate for the state x_i has been found, while other 89.1% cases led to no candidate (see also Figure 14.7a). It is also obvious from Figure 14.6 that in 37.5% cases the depth of the search tree was 0, which corresponds to our observations presented in Tables 14.2 and 14.3.

If at least one candidate for the state x_i has been found, then the average number of candidates was 9.19. The maximum number of candidates was 151. The average depth of the search tree was 34.80 steps. An average and a maximum number of steps to seek the whole search tree was 151.05 and 11272, respectively. A histogram of the number of steps is depicted in Figure 14.7b.

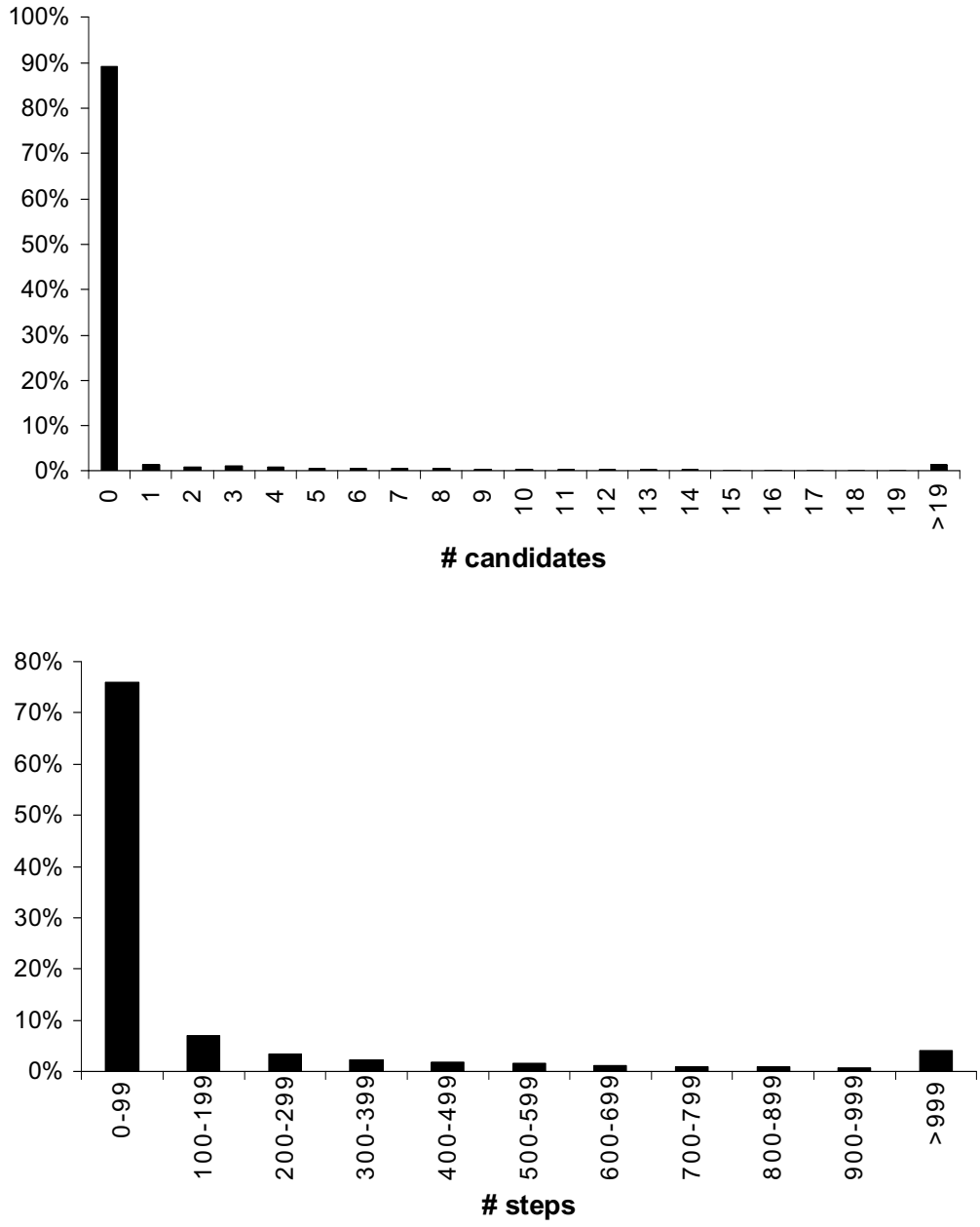

Figure 14.7: Test 4: a) Histogram of the candidates for the state x_i, b) Histogram of the number of steps to seek the whole search tree

14.5 Summary and Final Remarks

The method described here has an average complexity of $\approx 2^{10}$ steps and a maximum measured complexity of $\approx 2^{13}$ steps when backtracking the state x_i from the state x_e (Test 1 and Test 3), which represents a very small stretch of time. Running 10^8 instances of Test 1 and Test 2 took 15 hours and 30 hours, respectively, on an Intel Pentium Dual CPU T2390 1.86 GHz. One instance took 0.54 ms and 1.08 ms, respectively, on average.

The tests reveal one potential problem in cryptanalysis of A5/1. Due to non-injectivity of the cipher the number of candidates for the state x_i is relatively high. The number of candidates for x_i is about 10 on average, reaching about 200 in the worst case. From each candidate for x_i the candidate for x_k is derived by clocking the registers $R1$, $R2$ and $R3$ back and subtracting the bits of initialization vector in reverse order. Then, each candidate for x_k must be tested by trial decryption of other frames.

There is an alternative way for identification of the valid x_k. If we were able to recover internal states in two distinct frame keystreams of the same phone call, then we would derive two sets of candidates for x_k. Since all frames of the same phone call share the same state x_k, then the valid x_k would lie in an intersection of both sets.

Chapter 15

Conclusions of Part II

In this part we focused on cryptanalysis of GSM communication. The data transfers between the base transceiver station and the mobile phone are divided into 114 bit frames, encrypted with the stream cipher A5/1. Each frame is encrypted by its own keystream, generated with the usage of a common session key K and unique initialization vector IV.

We have presented two attacks and their implementation for the low-cost FPGA-based super-computing cluster COPACOBANA. Like many other attacks on stream ciphers, also our two attacks reveal the internal state that generated a given known keystream. Since the GSM communication is divided into frames, the recovered internal state is then used to derive the internal state x_k common for all frames. The state x_k represents the content of internal registers after their reset and loading the session key K. In most cases there are multiple candidates for the state x_k (about 10 on average and about 200 at maximum), which can slightly complicate the attack.

The first of our two attacks belongs to the family of guess-and-determine attacks. It reveals the internal state of the cipher in less than 6 hours on average demanding only 64 bits of known keystream. The attack is based on the proposal made by Keller and Seitz, however, we made several modifications and improvements of the attack. While the attack of Keller and Seitz has only 18% success ratio, our attack recovers the internal state in 100% of cases. We also corrected some estimations made by Keller and Seitz. Our attack is also very attractive with regard to financial costs which is a significant factor for the practicability of an attack: The acquisition costs for COPACOBANA are

about 10,000 €. Since COPACOBANA has a maximum power consumption of only 600 W, the attack also features very low operational costs. For instance, assuming 10 cent per kWh the operational costs of an attack are only 36 cents.

The second of our two attacks belongs to the family of time-memory-data tradeoff attacks. We have chosen the thin-rainbow method for the implementation. COPACOBANA is used in both the precomputation phase and the online phase of the attack. When designing the precomputation engine, we have utilized the features of underlying FPGA architecture to gain the maximum performance. The proposed design approach can be reused when designing similar attacks on other stream ciphers. Besides designing the hardware architectures for the precomputation and online phases, we also discussed solutions for relatively fast sort of a huge amount of acquired data and for relatively fast search in large table. The precomputation phase takes several month with one COPACOBANA; the TMDTO table occupies several terabytes of memory. The online phase is then finished in a fraction of time — COPACOBANA executes all calculations in a matter of (tens of) seconds and subsequent search of results in the TMDTO table is finished in a matter of minutes. Since this is a probabilistic attack, both the success ratio and the online complexity increase with the amount of known keystream data.

Both attacks complement each other perfectly. Upon eavesdropping the GSM communication and extraction of the keystream, we run fast TMDTO attack first. If the valid internal state x_k is not found (it is a probabilistic attack), then the (slower, but 100% successful) guess-and-determine attack is executed.

In our work we have not focused on eavesdropping GSM calls. Also the problem of obtaining a known plaintext is still under discussion in pertinent news groups and does not seem to be fully solved. However, these are just some technical difficulties that certainly cannot be considered serious barriers for breaking GSM.

Acronyms and Symbols

All acronyms are defined when first used in the text, with the exception of frequently used ones.

A5/1	stream cipher used in GSM communication
ASIC	Application-Specific Integrated Circuit
BTS	Base Transceiver Station
CB	Clocking Bit
CLB	Configurable Logic Block
COPACOBANA	Cost-Optimized Parallel Code Breaker
CPU	Central Processing Unit
DCM	Digital Clock Manager
DP	Distinguished Point
EC	Elliptic Curve
ECC	Elliptic Curve Cryptography
DLP	Discrete Logarithm Problem
ECDLP	Elliptic Curve Discrete Logarithm Problem
FIFO	First-in First-out; queue
FF	Flip-Flop
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
$GF(q)$	Finite Field (Galois Field)
GSM	Global System for Mobile communication
ITT	inversion algorithm by Itoh, Teechai and Tsuji
IV	Initialization Vector

LSB	Least Significant Bit
LUT	Look-Up Table
LFSR	Linear Feedback Shift Register
MSB	Most Significant Bit
MUX	Multiplexer
NB	Normal Basis
NIST	National Institute of Standards and Technology
ONB	Optimal Normal Basis
RAM	Random Access Memory
RFID	Radio-Frequency IDentification
RSA	asymmetric cipher by Rivest, Shamir and Adleman
TMDTO	Time-Memory-Data Trade-Off
TMTO	Time-Memory Trade-Off
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
VLSI	Very Large Scale Integration
XOR	eXclusive OR (reduction modulo 2)

Bibliography

[AMOV91] G. B. Agnew, R. C. Mullin, I. M. Onyszchuk, and S. A. Vanstone. An implementation for a fast public-key cryptosystem. *Journal of Cryptology*, pages 63–79, 1991.

[And94] R. Anderson. A5 (was: Hacking digital phones). sci.crypt, 17 June 1994.

[ANR99] G. C. Ahlquist, B. Nelson, and M. Rice. Optimal Finite Field Multipliers for FPGAs. In *FPL 1999: Proceedings of the 9th International Workshop on Field Programmable Logic and Applications*, pages 51–60, New York, NY, USA, 1999. Springer-Verlag New York, Inc.

[ANS97] ANSI. ANSI X9.30:1-1997, Public Key Cryptography for the Financial Services Industry: Part 1: The Digital Signature Algorithm (DSA) (revision of X9.30:1-1995), 1997.

[ANS98a] ANSI. ANSI X9.31-1998, Digital Signatures Using Reversible Public Key Cryptography for the Financial Services Industry (rDSA), 1998.

[ANS98b] ANSI. ANSI X9.62-1998, Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), 1998.

[Bab95] S. Babbage. A Space/Time Tradeoff in Exhaustive Search Attacks on Stream Ciphers. In *European Convention on Security and Detection*, May 1995.

[Bal08] M. Balík. Methods of Fast Search in Disk Stored Tables. Private communication via email, February 2008.

[BBK03] E. Barkan, E. Biham, and N. Keller. Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communications. In *Proc. of Crypto'03*, volume 2729 of *LNCS*. Springer-Verlag, 2003.

[BBK06] E. Barkan, E. Biham, and N. Keller. Instant Ciphertext-only Cryptanalysis of GSM Encrypted Communication (full-version). Technical Report CS-2006-07, Technion, 2006.

[BBS06] E. Barkan, E. Biham, and A. Shamir. Rigorous Bounds on Cryptanalytic Time/Memory Tradeoffs. In *Proc. of CRYPTO'06*, volume 4117 of *LNCS*, pages 1–21. Springer, 2006.

[BGW99] M. Briceno, I. Goldberg, and D. Wagner. A Pedagogical Implementation of the GSM A5/1 and A5/2 "voice privacy" Encryption Algorithms, 1999.

[BS00] A. Biryukov and A. Shamir. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In *Proc. of Asiacrypt'00*, volume 1976 of *LNCS*, pages 1–13. Springer, 2000.

[BSGEG04] S. Bajracharya, C. Shu, K. Gaj, and T. El-Ghazawi. Implementation of Elliptic Curve Cryptosystems over $GF(2^n)$ in Optimal Normal Basis on a Reconfigurable Computer. In *FPL2004: Proceeding of Field Programmable Logic and Applications*, pages 1001–1005. Springer, 2004.

[BSW01] A. Biryukov, A. Shamir, and D. Wagner. Real Time Cryptanalysis of A5/1 on a PC. In *Proc. of FSE'00*, volume 1978 of *LNCS*, pages 1–18. Springer-Verlag, 2001.

[Cer97] Certicom. The Elliptic Curve Cryptosystem. A Certicom Whitepaper, 1997.

[D. 82] D. E. R. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[DH76] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22:644–654, 1976.

[Ele98] Electronic Frontier Foundation. *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design*. O'Reilly & Associates Inc., July 1998.

[FIP00] FIPS. FIPS PUB 186-2, Digital Signature Standard, February 2000.

[Gao93] S. Gao. *Normal Bases over Finite Fields*. PhD thesis, University of Waterloo, Ontario, 1993.

[Gen08] Timo Gendrulis. Hardware-based cryptanalysis of the gsm a5/1 encryption algorithm. Master's thesis, Ruhr University Bochum, 2008.

[Gol97] J. Golic. Cryptanalysis of Alleged A5 Stream Cipher. In *Proc. of Eurocrypt'97*, volume 1233 of *LNCS*, pages 239–255. Springer-Verlag, 1997.

[Gol00] J. Golic. Cryptanalysis of three mutually clock-controlled stop/go shift registers. *IEEE Transactions on Information Theory*, 46:1081–1090, May 2000.

[GPP07] T. Gueneysu, C. Paar, and J. Pelzl. Attacking Elliptic Curve Cryptosystems with Special-Purpose Hardware. In *Proc. of FPGA'07*, pages 207–215. ACM Press, 2007.

[GS00] L. Gao and G.E. Sobelman. Improved VLSI designs for multiplication and inversion in $GF(2^M)$ over normal bases. In *Proceedings of 13th Annual IEEE International ASIC/SOC Conference*, pages 97–101, 2000.

[GWG99] I. Goldberg, D. Wagner, and L. Green. The Real-Time Cryptanalysis of A5/2. Presented at the Rump Session of Crypto'99, 1999.

[Hel80] M. E. Hellman. A Cryptanalytic Time-Memory Trade-off. *IEEE Transactions on Information Theory*, 26:401–406, 1980.

[Hsi70] M. Y. Hsiao. A Class of Optimal Minimum Odd-weight-column SEC-DED Codes. *IBM Journal of Research and Development*, 14(4):395–401, 1970.

[IEE00] IEEE. IEEE 1363. Standard for Public-key Cryptography, 2000.

[ITT86] T. Itoh, O. Teechai, and S Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^t)$ using normal bases. *J. Society for Electronic Communications (Japan)*, 44:31–36, 1986.

[KGKH04] S. Kwon, K. Gaj, C. H. Kim, and P. C. Hong. Efficient Linear Array for Multiplication in $GF(2^m)$ Using a Normal Basis for Elliptic Curve Cryptography. In *CHES 2004*, volume LNCS 3156, pages 76–91. Springer-Verlag, 2004.

[Knu98] D. E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1998.

[Kob87] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

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

[KS98] C. Koc and B. Sunar. Low-Complexity Bit-Parallel Canonical and Normal Basis Multipliers for a Class of Finite Fields. *IEEE Transactions on Computers*, 47:353–356, 1998.

[KS01] J. Keller and B. Seitz. A Hardware-Based Attack on the A5/1 Stream Cipher, 2001.

[Kwo03] S. Kwon. A low complexity and a low latency bit parallel systolic multiplier over $GF(2^m)$ using an optimal normal basis of type II. In *16th IEEE Symposium on Computer Arithmetic*, pages 196–202, 2003.

[LD99] J. Lopez and R. Dahab. Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation. In *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume LNCS 1717, pages 316–327. Springer-Verlag, 1999.

[LL02] P. H. W. Leong and I. K. H. Leung. A Microcoded Elliptic Curve Processor Using FPGA Technology. *IEEE Transactions on VLSI Systems*, 10:550–559, 2002.

[LL03] C. Lee and J. Lee. A Scalable Structure for a Multiplier and an Inversion Unit in $GF(2^m)$. *ETRI Journal*, 25:315–320, October 2003.

[LMWL00] K. H. Leung, K. W. Ma, W. K. Wong, and P. H. W. Leong. FPGA Implementation of Microcoded Elliptic Curve Cryptographic Processor. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 68–76, Napa Valley, California, USA, 2000.

[Mas97] MasterCard International, Inc. and Visa International Service Association. *SET Secure Electronic Transaction Specification*, May 31 1997.

[MBG+93] A. J. Menezes, I. F. Blake, XuHong Gao, R. C Mullin, S. A. Vanstone, and T. Yaghoobian. *Applications of Finite Fields*. Kluwer Academic Publishers, 1993.

[Mil86] V. S. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology - Crypto 85*, volume LNCS 218, pages 417–426. Springer-Verlag, 1986.

[MO86] J. Massey and J. Omura. Computational Method and Apparatus for Finite Field Arithmetic. U.S. patent number 4,587,627, 1986.

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

[MOVW89] R. Mullin, I. Onyszchuk, S. Vanstone, and R. Wilson. Optimal Normal Bases in $GF(p^n)$. *Discrete Applied Mathematics*, 22:149–161, 1989.

[Oec03] P. Oechslin. Making a Faster Cryptanalytic Time-Memory Trade-Off. In *Proc. of CRYPTO'03*, volume 2729 of *LNCS*, pages 617–630. Springer, 2003.

[Paa99] C. Paar. Implementation Options for Finite Field Arithmetic for Elliptic Curve Cryptosystems. In *ECC 1999*, 1999.

[PS00] T. Pornin and J. Stern. Software-hardware Trade-offs: Application to A5/1 Cryptanalysis. In *Proc. of CHES'00*, volume 1965 of *LNCS*, pages 318–327. Springer-Verlag, 2000.

[RMH03] A. Reyhani-Masoleh and M. A. Hasan. Low Complexity Sequential Normal Basis Multipliers over $GF(2^m)$. In *16th IEEE Symposium on Computer Arithmetic (ARITH-16 '03)*, pages 188–195, 2003.

[RSA78] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[Rup08] Andy Rupp. *Computational Aspects of Cryptography and Cryptanalysis*. PhD thesis, Ruhr University Bochum, 2008.

[SK01] B. Sunar and C. K. Koc. An efficient optimal normal basis type II multiplier. *IEEE Transactions on Computers*, 50:83–88, 2001.

[SRQL02] F. Standaert, G. Rouvroy, J. Quisquater, and J. Legat. A Time-Memory Tradeoff using Distinguished Points: New Analysis & FPGA Results. In *Proc. of CHES'02*, volume 2523 of *LNCS*, pages 596–611. Springer, 2002.

[SWPS08] J. Schrder, L. Wienbrandt, G. Pfeiffer, and M. Schimmler. Massively Parallelized DNA Motif Search on the Reconfigurable Hardware Platform COPACOBANA. In *Third IAPR International Conference on Pattern Recognition in Bioinformatics (PRIB 2008)*, volume LNCS 5265, pages 436–447, 2008.

[Uni05] University of California, Berkeley. Seti@Home Website, 2005.

[Wal] M. Wall. M: GAlib: A C++ Library of Genetic Algorithm Components. [Online].

[Xil99] Xilinx. XC4000E and XC4000X Series Field Programmable Gate Arrays, May 1999.

[Xil07] Xilinx. Spartan-3 FPGA Family: Complete Data Sheet, DS099, November 2007.

Refereed Publications of the Author

[A.1] J. Schmidt, M. Novotný, M. Jäger, M. Bečvář, and M. Jáchim. Exploration of Design Space in ECDSA. In *Field-Programmable Logic and Applications 2002*, LNCS 2438, pages 1072–1075. Springer Verlag, Montpellier 2002.

[A.2] J. Schmidt and M. Novotný. Normal Basis Multiplication and Inversion Unit for Elliptic Curve Cryptography. In *Proceedings of the 10th IEEE International Conference on Electronics, Circuits and Systems 2003*, pages 82–85. IEEE, Piscataway 2003.

[A.3] J. Schmidt and M. Novotný. Optimum Shifter Synthesis Using a Genetic Algorithm. In *Recent Trends in Multimedia Information Processing*, pages 146–149. Sdělovací technika, Praha 2003.

[A.4] J. Schmidt and M. Novotný. Scalable Shifter Synthesis for a Finite Field Arithmetic Unit. In *Proceedings of 7th IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop*, pages 195–198. Institute of Informatics, Slovak Academy of Sciences, Bratislava, 2004.

[A.5] J. Schmidt and M. Novotný. Scalable Normal Basis Arithmetic Unit for Elliptic Curve Cryptography. *Acta Polytechnica*, 45:55–60, 2005

[A.6] M. Novotný and J. Schmidt. General Digit-Serial Normal Basis Multiplier. In *Proceedings of 8th IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop*, pages 99–104. University of Western Hungary, Sopron, 2005.

[A.7] M. Novotný and J. Schmidt. Normal Basis Multipliers of General Digit Width Applicable in ECC. In *Proceedings of 9th IEEE Design and Di-*

agnostics of Electronic Circuits and Systems Workshop, pages 145–146. CTU Publishing House, Praha, 2006

[A.8] M. Novotný and J. Schmidt. General Digit Width Normal Basis Multipliers with Circular and Linear Structure. In *Field-Programmable Logic and Applications - FPL2006*, pages 873–876, Madrid 2006

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

[A.10] M. Novotný and J. Schmidt. General Digit-Serial Normal Basis Multiplier with Distributed Overlap. In *Proceedings of 10th Euromicro Conference on Digital System Design*, pages 94–101, IEEE Computer Society, Lübeck 2007

[A.11] T. Gendrullis, M. Novotný and A. Rupp. A Real-World Attack Breaking A5/1 within Hours. In *Proceedings of the 10th Workshop on Cryptographic Hardware and Embedded Systems (CHES 2008)*, pages 266–282, Springer Verlag, Washington, D.C., 2008

[A.12] T. Güneysu, T. Kasper, M. Novotný, C. Paar and A. Rupp. Cryptanalysis with COPACOBANA. In *IEEE Transactions on Computers*, 2008, vol. 57, no. 11, pages 1498–1513

Curriculum Vitae

Personal Data

Name	Martin Novotný
Born	May 17th, 1969, in Prague, Czechoslovakia (currently the Czech Republic)
Contact	`novotnym@fit.cvut.cz`

Education and Qualifications

1987–1992	Master Studies (Ing.) Faculty of Electrical Engineering Czech Technical University in Prague, Czech Republic Branch of Study: Electronic Computers (Graduated with Honours)
since 2003	Ph.D. study Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic.
2007–2008	Research Stay & Ph.D. study Chair for Embedded Security, Horst Görtz Institute for IT Security, Ruhr-University Bochum, Germany.

Employment History

1992–2009	Department of Computer Science and Engineering Faculty of Electrical Engineering Czech Technical University in Prague
1994–2005	Administrator of the Hardware Laboratory, Teacher
2005–2009	Assistant Professor
since 2009	Department of Digital Design Faculty of Information Technology Czech Technical University in Prague
since 2009	Assistant Professor