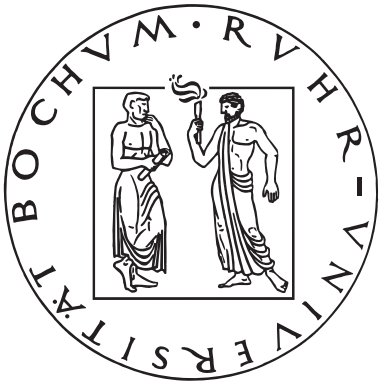
Cryptography and Cryptanalysis on Reconfigurable Devices

Security Implementations for Hardware and Reprogrammable Devices



Dissertation

---

for the degree "Doktor-Ingenieur".
Ruhr-University Bochum, Germany,
Faculty of Electrical Engineering and Information Technology.

Tim Erhan Güneysu
Bochum, February 2009

Cryptography and Cryptanalysis on Reconfigurable Devices
DOI:10.000/XXX

This thesis is dedicated to Sindy for her love and support throughout the course of this thesis.

In loving memory of my parents.

Author's contact information:
tim@gueneysu.de
www.gueneysu.de

Thesis Advisor:	**Prof. Dr. Christof Paar** Ruhr-University Bochum, Germany
Secondary Referee:	**Prof. Dr. Daniel J. Bernstein** University of Illinois at Chicago

"Aber für was ist das gut?"
(Ingenieur vom Advanced Computing Systems Division of IBM, 1968, zum Mikrochip)

Abstract

With the rise of the Internet, the number of information processing systems has significantly increased in many fields of daily life. To enable commodity products to communicate, so-called *embedded computing systems* are integrated into these products. However, many of these small systems need to satisfy strict application requirements with respect to cost-efficiency and performance. In some cases, such a system also needs to drive cryptographic algorithms for maintaining data security – but without significantly impacting the overall system performance. With these constraints, most small microprocessors, which are typically employed in embedded systems, cannot provide the necessary number of cryptographic computations. Dedicated hardware is required to handle such computationally challenging cryptography. This thesis presents novel hardware implementations for use in cryptography and cryptanalysis.

The first contribution of this work is the development of novel high-performance implementations for symmetric and asymmetric cryptosystems on reconfigurable hardware. More precisely, most presented architectures target hardware devices known as *Field Programmable Gate Arrays* (FPGAs) which consist of a large number of generic logic elements that can be dynamically configured and interconnected to build arbitrary circuits. The novelty of this work is the usage of dedicated arithmetic function cores – available in some modern FPGA devices – for cryptographic hardware implementations. These arithmetic functions cores (also denoted as DSP blocks) were originally designed to improve filtering functions in Digital Signal Processing (DSP) applications. The thesis at hand investigates how these embedded function cores can be used to significantly accelerate the operation of symmetric block ciphers such as AES (FIPS 197 standard) as well as asymmetric cryptography, e.g., Elliptic Curve Cryptography (ECC) over NIST primes (FIPS 186-2/3 standard).
Graphics Processing Units (GPU) on modern graphics cards provide computational power exceeding that of most recent CPU generations. In addition to FPGAs, this work also demonstrates how graphics cards can be used for high performance asymmetric cryptography. For the first time in open literature, the standardized asymmetric cryptosystem RSA (PKCS #1) and ECC over the NIST prime P-224 are implemented on an NVIDIA 8800 GTS graphics card, making use of the Compute Uniform Device Architecture (CUDA) programming model.

A second aspect of this thesis is cryptanalysis based on FPGA-based hardware architectures. All cryptographic methods involve an essential trade-off between efficiency and security margin, i.e., a higher security requires more (and more complex) computations leading to degraded performance of the cryptosystem. Hence, to maintain efficiency, the designer of a cryptosystem

must carefully adapt the security margin according to the computational power of a potential attacker with high but limited computing resources. It is therefore essential to determine the cost performing an attack on a cryptosystem as precisely as possible - using a concrete metric like the required financial costs to attack a specific cryptographic setup. In this context, another contribution of this thesis is the design and enhancement of an FPGA-based cluster platform (COPACOBANA) which was developed to provide a computational platform with optimal cost-performance ratio for cryptanalytic applications. COPACOBANA is used to mount brute-force and advanced attacks on the weak DES cryptosystem, which was the worldwide and long-lasting standard for block ciphers (FIPS 46-3 standard) until superseded by AES. Due to its popularity for many years, various legacy and recent products still rely on the security of DES. As an example, a class of recent one-time password token generators are broken in this work. Furthermore, this thesis discusses attacks on the Elliptic Curve Discrete Logarithm Problem (ECDLP) used in context with ECC cryptosystems as well as Factorization Problem (FP), which is the basis for the well-known RSA system.

A third and last contribution of this thesis considers the protection of reconfigurable systems themselves and contained security-related components. Typically, logical functions in FPGAs are dynamically configured from SRAM cells and lookup tables used as function generators. Since the configuration is loaded at startup and also can be modified during runtime, an attacker can easily compromise the functionality of the hardware circuit. This is particularly critical for security related functions in the logical elements of an FPGA, e.g., the attacker could be able to extract secret information stored in the FPGA just by manipulating its configuration. As a countermeasure, FPGA vendors already allow the use of encrypted configuration files with some devices to prevent unauthorized tampering of circuit components. However, in practical scenarios the secure installation of secret keys required for configuration decryption by the FPGA is an issue left to the user to solve. This work presents an efficient solution for this problem which hardly requires any changes to the architecture of recent FPGA devices. Finally, this thesis presents a solution on how to install a trustworthy security kernel – also known as Trusted Platform Module (TPM) – within the dynamic configuration of an FPGA. A major advantage of this approach with respect to the PC domain is the prevention of bus eavesdropping between TPM and application since all functionality is encapsulated in a System-on-a-Chip (SoC) architecture. Additionally, the functionality of the TPM can easily be extended or updated in case a security component has been compromised without need to replace the entire chip or product.

Keywords

Cryptography, Cryptanalysis, High-Performance Implementations, Hardware, FPGA

Kurzfassung

Seit Durchbruch des Internets ist die Zahl an informationsverarbeitenden Systemen in vielen Bereichen des täglichen Lebens stark gewachsen. Dabei kommen bei der Kommunikation und Verarbeitung von Daten in den verschiedensten Gegenständen des Alltags eingebettete Systeme zum Einsatz, die oft harten Anforderungen, wie beispielsweise hohe Leistung bei optimaler Kosteneffizienz, gerecht werden müssen. Zusätzlich müssen diese – je nach Anwendungsfall – weitere Kriterien, wie z.B. Sicherheitsaspekte durch kryptografische Verfahren, ohne nennenswerte Einbußen bezüglich der Datenverarbeitungsgeschwindigkeit erfüllen. In diesem Zusammenhang sind kleine Mikrocontroller, wie sie typischerweise in diesen Systemen verwendet werden, schnell überfordert, so dass für kryptografische Funktionen in eingebetteten Hochleistungssystemen fast immer dedizierte Hardwarechips zum Einsatz kommen.

Ein erster Kernaspekt dieser Dissertation beschäftigt sich mit Hochleistungsimplementierungen von symmetrischen wie asymmetrischen Kryptosystemen auf rekonfigurierbarer Hardware (*Field Programmable Gate Arrays* oder kurz *FGPAs*). Ein Herausstellungsmerkmal der Arbeit ist hierbei die Implementierung der standardisierten AES-Blockchiffre (FIPS 197) sowie von Elliptischen Kurven Kryptosystemen (ECC) über Primkörpern (FIPS 186-2/3) unter Nutzung von dedizierten Arithmetikfunktionskernen moderner FPGAs, die primär für Filteroperationen der klassischen digitalen Signalverarbeitung entwickelt wurden. Neben dem Einsatz von FPGAs wird weiterhin die Eignung von modernen, handelsüblichen Grafikkarten als Koprozessorsystem für asymmetrische Kryptosysteme untersucht, die durch hohe parallele Rechenleistung sowie günstige Anschaffungskosten eine weitere Option für effiziente kryptografische Hochgeschwindigkeitslösungen darstellen. Basierend auf einer NVIDIA 8800 GTS Grafikkarte werden im Rahmen dieser Arbeit neuartige Implementierungen für das RSA sowie ECC Kryptosystem vorgestellt.

Ein zweiter Aspekt dieser Arbeit ist die Kryptanalyse mit Hilfe von FPGA-basierten Spezialhardwarearchitekturen. Alle praktikablen, kryptografischen Verfahren sind grundsätzlich der Abwägung zwischen Effizienz und dem gewünschten Maß an Sicherheit unterworfen; desto höher die Sicherheitsanforderungen sind, desto langsamer ist im Allgemeinen das Kryptosystem. Die Sicherheitsparameter eines Kryptosystems werden daher aus Effizienzgründen an die besten zu Verfügung stehenden Angriffsmöglichkeiten angepasst, wobei einem Angreifer ein hohes, aber beschränktes Maß an Rechenleistung zugesprochen wird, das dem gewünschten Sicherheitsniveau entsprechen soll. Aus diesem Grund muss die Komplexität eines Angriffs genau untersucht werden, damit eine präzise Angabe der durch das Kryptosystem *tatsächlich* erreich-

ten Sicherheit in praktikabler Weise gemacht werden kann. Im Rahmen dieser Arbeit wurde maßgeblich der FPGA-basierte Parallelcluster *COPACOBANA* mit- und weiterentwickelt. Dieser speziell auf eine optimale Kosten-Leistungseffizienz ausgelegte Cluster ermöglicht genaue Aufwandsabschätzungen von Angriffen auf verschiedenen Kryptosystemen, u.a. auf Basis einer finanziellen Metrik. Mit Hilfe dieser Clusterplattform können sowohl schwache oder ältere Kryptosysteme gebrochen, wie auch Angriffe auf aktuell als sicher geltende kryptografische Verfahren abgeschätzt werden. Neben der erfolgreichen Kryptanalyse der symmetrischen DES-Blockchiffre, sind ein weiterer Teil dieser Arbeit neuartige Hardwareimplementierungen von (unterstützenden) Angriffen auf asymmetrische Kryptosysteme, die auf dem Elliptischen Kurven Diskreten Logarithmus Problem (ECDLP) oder dem Faktorisierungsproblem (FP) basieren.

Ein dritter und letzter Bereich dieser Dissertation betrifft den Schutz der rekonfigurierbaren Hardware und seinen logischen Komponenten selbst. Es handelt sich bei typischen FPGAs zumeist um dynamische SRAM-basierte Logikschaltungen, die zur Laufzeit (um-)konfiguriert werden können. Deshalb muss insbesondere bei sicherheitskritischen Funktionen darauf geachtet werden, dass die Konfiguration des FPGA durch einen Angreifer nicht manipuliert werden kann, um beispielsweise ein Auslesen des geheimen Schlüssels oder die Kompromittierung eines eingesetzten Sicherheitsprotokolls zu verhindern. Manchen FPGA hat der Hersteller bereits mit der Funktion ausgestattet, symmetrisch verschlüsselte Konfigurationsdateien zu verwenden. Jedoch besteht gerade bei komplizierteren Geschäftsmodellen in der Praxis das klassische Problem der Schlüsselverteilung, d.h. wie kann der Hersteller von FPGA-Konfigurationsdateien den vom FPGA zur Entschlüsselung der Konfiguration benötigten Schlüssel im Chip installieren, ohne dabei physischen Zugriff auf den FPGA zu haben? In dieser Dissertation wird hierfür ein sicheres Protokoll vorgestellt, welches auf dem Diffie-Hellman Schlüsselaustauschverfahren basiert und dieses Schlüsselverteilungsproblem löst.
Weiterhin werden FPGAs auf ihre Fähigkeit untersucht, einen dynamisch konfigurierbaren Sicherheitskern, ein so genanntes *Trusted Platform Module* (TPM), in einem dedizierten, dynamischen Bereich einzurichten, der einer Applikation vertrauenswürdige Sicherheitsfunktionen zu Verfügung stellen kann. Der große Vorteil dieses Systems in Bezug auf klassischen TPM-Architekturen im PC-Umfeld ist dabei die erschwerte Abhörbarkeit sicherheitsrelevanter Busleitungen, da hier ein vollständiger System-on-a-Chip (SoC)-Architektur zum Einsatz kommt. Weiterhin können durch die dynamische Erweiter- und Aktualisierbarkeit der Sicherheitsfunktionen im rekonfigurierbaren System schwache oder gebrochene Sicherheitskomponenten jederzeit ausgetauscht werden, ohne dafür das gesamte System ersetzen zu müssen.

Schlagworte

Kryptographie, Kryptanalyse, Hochgeschwindigkeitsimplementierungen, Hardware, FPGA

Acknowledgements

This thesis is the result of nearly three years of cryptographic research in which I have been accompanied and supported by many people during this time. Now I'd like to say thank you. First, I would like to express my deep and sincere gratitude to my supervisor Prof. Christof Paar for his continuous inspiration. I am grateful and glad that he gave me advice in professional and personal matters and also shared many of his of his research experiences with me. And, without doubt, he has the most outstanding talent to motivate people!

Furthermore, I like to thank my thesis committee, especially Prof. Daniel J. Bernstein for his very valuable council as external referee.

Next, I want to thank my wife Sindy and my family, in particular Ludgera, Suzan, Maria and Denis, for all their great support and encouragement during the course of preparing for my PhD. Thank you!

Very important for my research career at the university was the joint work accomplished with Jan Pelzl. It was him who introduced me to the scientific community and also showed me how to efficiently write research contributions. I also want to thank Saar Drimer for all the research projects on which we collaborated. I thoroughly enjoyed the time we shared during his stay in Bochum. Many thanks go to my colleagues and friends Thomas Eisenbarth, Markus Kasper, Timo Kasper, Kerstin Lemke-Rust, Martin Novotný, Axel Poschmann, Andy Rupp and Marko Wolf for discussions, publications and projects in all aspects of cryptography and, of course, also the great time and activities beyond work! Moreover, I should not forget the COPACOBANA team led by Gerd Pfeiffer and Stefan Baumgart who always did outstanding work to support me in all low-level hardware questions with respect to our joint work on FPGA-based cluster architectures. Also, I like to thank Christa Holden for all her efforts on final corrections in my thesis. Last but not least, a special "thank you" is due to our team assistant Irmgard Kühn for contributing to the outstanding atmosphere in our group and all her support with any administrative task. I like to thank all the hard-working students I supervised, in particular Hans-Christian Röpke, Sven Schäge, Christian Schleiffer, Stefan Spitz and Robert Szerwinski.

And if you're now done reading these lines with any, yet unsatisfied expectations, I'd like to let you know that I certainly also intend to thank you. Thanks a lot!

Table of Contents

Chapter 1

Introduction

This chapter introduces the aspects of cryptography and cryptanalysis for reprogrammable devices and summarizes the research contributions of this thesis.

Contents of this Chapter

1.1 Motivation

Since many recent commodity products integrate electronic components to provide more functionality, the market for *embedded systems* has grown expansively. Likewise, the availability of new communication channels and data sources, like mobile telephony, wireless networking and global navigation systems, has created a demand for a various mobile devices and handheld computers. Along with the new features for data processing and communication, the need for various security features on all of these devices has arisen. Examples for such security requirements are the installation and protection of vendor secrets inside a device to enable gradual feature activation, secure firmware updates, and also aspects of user privacy. Some applications even demand a complex set of interlaced security functions involving all fields of cryptography. Additionally, these applications often put a demand on the necessary data throughput or define a minimum number of operations per second. Since most embedded systems are based on small microprocessors with limited computing power, execution of computationally costly cryptographic operations on these platforms are extremely difficult without severely impacting performance. This is where special-purpose hardware implementations for the cryptographic components come into play.

Compared to microprocessor-based platforms, specifically designed hardware implementations can be designed optimally with respect to time and area complexity for most applications. Currently, the only options to build such hardware chips for a specific application are the *Application Specific Integrated Circuit* (ASIC) implementing the application as a static circuit

and the *Field Programmable Gate Array* (FPGA) which allows mapping the application circuitry dynamically into a two-dimensional array of generic and reconfigurable logic elements.

Though an ASIC provides best possible performance and lowest cost per unit, its development process is expensive due to the required setup of complex production steps and the manpower involved. Furthermore, the circuit of an ASIC is inherently static and cannot be modified afterwards so that design changes require complete redevelopment. This does not only affect system prototypes during development: it is especially crucial for later upgrades of cryptosystems which have been reported compromised or insecure, but have already been delivered to the customer. With classic ASIC technology, such a modification requires an expensive rollback and in most cases the exchange of the entire device.

Since the mid eighties, the FPGA technology has provided *reconfigurable logic* on a chip [Xil08a]. Instead of using fixed combinatorial paths and fine-grain logic made up from standard-cell libraries as with ASICs, these reconfigurable devices provide *Configurable Logic Blocks* (CLB) capable of providing logical functions that can be reconfigured during runtime. As a result of their dynamic configuration feature, FPGA allow for rapid prototyping of systems with minimal development time and costs. However, FPGAs come as a complete package with a specific amount of reconfigurable logic making the use of FPGAs for a specific hardware application more coarse-grain and thus more costly than ASICs (post development). Besides FPGAs, so called *Complex Programmable Logic Devices* (CPLD) are an alternative and cheaper variant of reconfigurable devices. Note that CPLDs consist of large configurable macro cells with fixed and static interconnects and are thus used for simple hardware applications like bus arbitration or low-latency signal processing. On the contrary, FPGAs have a finer grain architecture and freely allow the connection of a large number of logic elements via a programmable switch matrix. This makes FPGAs the best choice for complex systems such as cryptographic and cryptanalytic algorithms. In this context such algorithms can be integrated in FPGAs either as a holistic approach together with the main application and deployed as a *System-on-a-Chip* (SoC) or as coprocessor unit extending the feature set of a separate microprocessor. In this thesis, we focus mainly on crypto implementations for FPGAs, since they provide sufficient logic resources for complex implementations *and* the feature of reconfigurability to update implemented security functions when necessary.

This thesis focuses on hardware implementations both in the fields of *cryptography* and *cryptanalysis*. In general, cryptography is considered the constructive science of securing information, by means of mathematical techniques and known hard problems. Cryptanalysis on the other hand denotes the destructive art of revealing the secured information from an attacker's perspective without the knowledge of any secret. Cryptanalysis is an essential concept maintaining the effectiveness of cryptography – cryptographers should carefully review their cryptosystems with the (known) tools given by cryptanalysis to assess the threat and possibilities of potential attackers.

The field of cryptography is divided into *public-key* (*asymmetric*) and *private-key* (*symmetric*) cryptography. In symmetric cryptography, all trusted parties share a common secret key, e.g., to establish confidential communication. This symmetric approach to secure communication channels has been used throughout history. As an example, first monoalphabetic shift ciphers were already employed by Julius Caesar around 70 BC [TraAD]. In contrast, asymmetric cryptography is rather new and was first introduced in open literature by Diffie and Hellman [DH76] in the mid 1970s. In this approach, each party is provided with a key pair consisting of a secret and public key. Encryption of data can be performed by everyone who has knowledge of the public key, but only the owner of the secret key can decrypt information. Besides encryption, public-key cryptography can also be used to efficiently achieve other security goals, such as mutual key agreement and digital signatures.

In the past, symmetric and asymmetric cryptosystems are both essential in practical systems. By nature, the computational complexity of asymmetric cryptography is much higher than with symmetric cryptography. This is due to the necessity of hard mathematical problems which are converted to *one-way functions with trapdoors* to support the complex principle of a secret with a public and private component. Common choices of hard problems for these one-way functions are the *Factorization Problem* (FP), which is the foundation of the security of the popular RSA [RSA78] system, and the *Discrete Logarithm Problem* for finite fields (DLP) or elliptic curve groups (ECDLP). Public-key cryptography is thus only employed for applications with demand for the advanced security properties of the asymmetric key approach. For all other needs, like bulk data encryption, the symmetric cryptography is the more efficient choice, e.g., using the legacy *Data Encryption Standard* (DES) or the *Advanced Encryption Standard* (AES) block ciphers. In many cases, hybrid cryptography comprising symmetric *and* asymmetric cryptography is required (e.g., to provide symmetric data encryption with fresh keys which are obtained from an asymmetric key agreement scheme).

This thesis provides new insights into the field of asymmetric and symmetric cryptography as well as the cryptanalysis of established cryptosystems (and related problems) by use of reconfigurable devices. In addition to that, this work also presents novel measures and protocols to protect reconfigurable devices against manipulation, theft of *Intellectual Property* (IP) and secret extraction.

1.2 Summary of Research Contributions

Most of the presented design strategies and implementations of cryptographic and cryptanalytic applications in this thesis target Xilinx FPGAs. Xilinx Inc is the current market leader in FPGA technology, hence, the presented results can be widely applied where FPGA technology comes into play. All presented cryptographic architectures in this contribution aim at applications with

high demands for data throughput and performance. For these designs, we[1] primarily employ powerful Xilinx Virtex-4 and Virtex-5 FPGAs which include embedded functional elements that can accelerate the arithmetic operations of many cryptosystems.

Implementations for cryptanalytic applications are usually designed to achieve an optimal cost-performance ratio. More precisely, the challenge is to select an (FPGA) device which is available at minimal cost but can provide a maximum number of cryptanalytic operations. Hence, we mainly tailor our architectures for cryptanalytic applications specifically for clusters consisting of cost-efficient Xilinx Spartan-3 FPGAs.

Finally, we present strategies to protect the configuration and security-related components on FPGAs. Our protection and trust models are designed for use with arbitrary FPGAs satisfying a specific set of minimum requirements (e.g., on-chip configuration decryption).

Summarizing, the following topics have been investigated in this thesis:

- High-performance implementations of the symmetric AES block cipher on FPGAs
- High-performance implementations of Elliptic Curve Cryptosystems (ECC) over NIST-primes on FPGAs
- Implementations of RSA and ECC public-key cryptosystems on modern graphics cards
- FPGA architectures for advanced cryptanalysis of the DES block cipher and DES-related systems
- Implementations to solve the Elliptic Curve Discrete Logarithm Problem on FPGAs
- Improvements to the hardware-based Elliptic Curve Method (ECM)
- Protection methods of Intellectual Property (IP) contained in FPGA configuration bitstreams
- Establishing a chain of trust and trustworthy security functions on FPGAs

1.2.1 High-Performance Cryptography on Programmable Devices

This first part presents novel high-performance solutions for standardized symmetric and asymmetric cryptosystems for FPGAs and graphics cards. We propose new design strategies for the symmetric AES block cipher (FIPS-197) on Virtex-5 FPGAs and asymmetric ECC over NIST primes P-224 and P-256 according to FIPS 186-2/3 on Virtex-4 FPGAs. Moreover, we will discuss implementations of asymmetric cryptosystems on graphics cards and develop solutions for RSA-1024, RSA-2048 and ECC based on the special NIST prime P-224 on these devices.

[1] Though this thesis represents my own work, some parts result from joint research projects with other contributors. Therefore, I prefer to use "we" rather than "I" throughout this thesis.

Optimal AES Architectures for High-Performance FPGAs

The Advanced Encryption Standard is the most popular block cipher due to its standardization by NIST in 2002. We developed an AES cipher implementation that is almost exclusively based on embedded memory and arithmetic units embedded of Xilinx Virtex-5 FPGAs. It is designed to match specifically the features of this modern FPGA class – yielding one of the smallest and fastest FPGA-based AES implementation reported up to now – with minimal requirements on the (generic) configurable logic of the device. A small AES module based on this approach returns a 32 bit column of an AES round each clock cycle, with a throughput of 1.76 Gbit/s when processing two 128 bit input streams in parallel or using a counter mode of operation. Moreover, this basic module can be replicated to provide a 128 bit data path for an AES round and a fully unrolled design yielding throughputs of over 6 and 55 Gbit/s, respectively.

Optimal ECC Architectures for High-Performance FPGAs

Elliptic curve cryptosystems provide lower computational complexity compared to other traditional cryptosystems like RSA [RSA78]. Therefore, ECCs are preferable when high performance is required. Despite a wealth of research regarding high-speed implementation of ECC since the mid 1990s [AMV93, WBV+96], providing truly high-performance ECC on reconfigurable hardware platforms is still an open challenge. This applies especially to ECCs over prime fields, which are often selected instead of binary fields due to standards in Europe and the US. In this thesis, we present a new design strategy for an FPGA-based, high performance ECC implementation over prime fields. Our architecture makes intensive use of embedded arithmetic units in FPGAs originally designed to accelerate digital signal processing algorithms. Based on this technique, we propose a novel architecture to create ECC arithmetic and describe the actual implementation of standard compliant ECC based on the NIST primes.

High-Performance Asymmetric Cryptography with Graphics Cards

Modern Graphics Processing Units (GPU) have reached a dimension that far exceeds conventional CPUs with respect to performance and gate count. Since many computers already include such powerful GPUs as stand-alone graphics card or chipset extension, it seems reasonable to employ these devices as coprocessing units for general purpose applications and computations to reduce the computational burden of the main CPU. This contribution presents novel implementations using GPUs as accelerators for asymmetric cryptosystems like RSA and ECC. With our design, an NVIDIA Geforce 8800 GTS can compute 813 modular exponentiations per second for RSA with 1024 bit parameters (or, alternatively, for the Digital Signature Standard (DSA)). In addition to that, we describe an ECC implementation on the same platform which is capable to compute 1412 point multiplications per second over the prime field $P - 224$.

Extracts of the contributions presented in this part were also published in [DGP08, GP08, SG08].

1.2.2 Cryptanalysis with Reconfigurable Hardware Clusters

In this part, we investigate scalable and reconfigurable architectures to support the field of cryptanalysis. For this purpose, we develop and enhance a parallel computing cluster based on cost-efficient Xilinx Spartan-3 FPGAs. Besides actual attacks on weak block ciphers like the DES, we also discuss how to employ this computing platform for attacks on the security assumptions of asymmetric cryptosystems like RSA and ECC.

Cryptanalysis of DES-based Systems with Special Purpose Hardware

Cryptanalysis of symmetric (and asymmetric) ciphers is a challenging task due to the enormous amount of computations involved. The security parameters of cryptographic algorithms are commonly chosen so that attacks are infeasible with available computing resources. Thus, in the absence of mathematical breakthroughs to a cryptanalytical problem, a promising way for tackling the computations involved is to build special-purpose hardware which provide a better performance-cost ratio than off-the-shelf computers in many cases. We have developed a massively parallel cluster system (COPACOBANA) based on low-cost FPGAs as a cost-efficient platform primarily targeting cryptanalytical operations with these high computational but low communication and memory requirements [KPP+06b]. Based on this machine, we investigate here various attacks on the weak DES cryptosystem which was the long-lasting standard block cipher according to FIPS 46-3 since 1977 – and is still used in many legacy (and even recent) systems. Besides simple brute-force attack on DES, we also evaluate time-memory trade-off attacks for DES keys on COPACOBANA as well as the breaking of more advanced modes of operations of the DES block cipher, e.g., some one-time password generators.

Parallelized Pollard-Rho Method Hardware Implementations for Solving the ECLDP

As already mentioned, the utilization of Elliptic Curves (EC) in cryptography is very promising for embedded systems due to small parameter sizes. This directly results from their resistance against powerful index-calculus attacks meaning only generic, exponential-time attacks like the Pollard-Rho method are available. We present here a first concrete hardware implementation of this attack against ECC over prime fields and describe an FPGA-based multi-processing hardware architecture for the Pollard-Rho method. With the implementation at hand and given a machine like COPACOBANA, a fairly accurate estimate about the cost of an FPGA-based attack can be generated. We will extrapolate the results on actual ECC key lengths (128 bits and above) and estimate the expected runtimes for a successful attack. Since FPGA-based attacks are out of reach for key lengths exceeding 128 bits, we also provide additional estimates based on ASICs.

Improving the Elliptic Curve Method in Hardware

The factorization problem is a well-known mathematical issue that mathematicians have already attempted to tackle since the beginning. Due to the lack of factorization algorithms

with better than subexponential complexity, cryptosystems like the well-established asymmetric RSA system remain state-of-the-art. Since the best known attacks like the Number Field Sieve (NFS) are too complex to be (efficiently) handled solely by (simple) FPGA systems, we focus on improvements of hardware architectures of the Elliptic Curve Method (ECM) which is preferably also used in substeps of the NFS. Previous implementations of ECM on FPGAs were reported by Pelzl *et al.* [ŠPK+05] and Gaj *et al.* [GKB+06a]. In this work we will optimize the low-level arithmetic of their proposals by employing the DSP blocks of modern FPGAs and also discuss also high-level decisions as the choice of alternative elliptic curve representation like Edwards curves.

Parts of the presented research contributions were also published by the author in [GKN+08, GRS07, GPP+07b, GPP07a, GPP08, GPPS08].

1.2.3 Trust and Protection Models for Reconfigurable Devices

This part investigates trust and protection models for reconfigurable devices. This comprises the authenticity and integrity of security functions implemented in the configurable logic as well as prevention mechanisms against theft of the IP contained in the configuration of FPGAs.

Intellectual Property Protection for FPGA Bitstreams

The distinct advantage of SRAM-based FPGAs is their flexibility for configuration changes. However, this opens up the threat of IP theft since the system configuration is usually stored in easy-to-access external Flash memory. To prevent this, high-end FPGAs have already been fitted with symmetric-key decryption engines used to load an encrypted version of the configuration that cannot easily be copied and used without knowledge of the secret key. However, such protection systems based on straightforward use of symmetric cryptography are not well-suited with respect to business and licensing processes, since they are lacking a convenient scheme for key transport and installation. We propose a new protection scheme for the IP of circuits in configuration files that provides a significant improvement to the current unsatisfactory situation. It uses both public-key and symmetric cryptography, but does not burden FPGAs with the usual overhead of public-key cryptography: While it needs hardwired symmetric cryptography, the public-key functionality is moved into a temporary configuration file for a one-time setup procedure. Therefore, our proposal requires only very few modifications to current FPGA technology.

Trusted Computing in Reconfigurable Hardware

Trusted Computing (TC) is an emerging technology used to build trustworthy computing platforms which can provide reliable and untampered security functions to upper layers of an application. The Trusted Computing Group (TCG) has proposed several specifications to implement TC functionalities by a hardware extension available for common computing platforms,

the Trusted Platform Module (TPM). We propose a reconfigurable (hardware) architecture with TC functionalities where we focus on security functionality as proposed by the TCG for TPMs [Tru06], however specifically designed for embedded platforms. Our approach allows for an efficient design and update of security functionalities for hardware-based crypto engines and accelerators. We discuss a possible implementation based on current FPGA architectures and point out the associated challenges, in particular the protection of the internal, security-relevant state which should not be subject to manipulation, replay, and cloning.

Extracts of the research contributions in this part are published in [GMP07a, GMP07b, EGP+07a, EGP+07b]

Part I

High-Performance Cryptosystems on Reprogrammable Devices

Chapter 2

Optimal AES Architectures for High-Performance FPGAs

This chapter presents an AES cipher implementation that is based on memory blocks and DSP units embedded within Xilinx Virtex-5 FPGAs. It is designed to match specifically the features of these modern FPGA devices – yielding the fastest FPGA-based AES implementation reported in open literature with minimal requirements on the configurable logic of the device.

Contents of this Chapter

2.1 Motivation

Since its standardization in 2001 the Advanced Encryption Standard (AES) [Nat01] has become the most popular block cipher for many applications with requirements for symmetric security. Therefore, by now there exist a multitude of implementations and literature discussing how to optimize AES in software and hardware. In this chapter we will focus on AES implementations in reconfigurable hardware, in particular on Xilinx Virtex-5 FPGAs.

Analyzing existing solutions, these AES implementations are mostly based on traditional configurable logic to maintain platform independence and thus do not exploit the full potential of modern FPGA devices. Thus, we present a novel way to implement AES based on the 32-bit T-Table method [DR02, Section 4.2] by taking advantage of new embedded functions located inside of the Xilinx Virtex-5 FPGA [Xil06], such as large dual-ported RAMs and Digital Signal Processing (DSP) blocks [Xil07] with the goal of minimizing the use of registers and look-up

tables that could otherwise be used for other functions. Unlike conventional AES design approaches for these FPGAs [BSQ+08], our design is especially suitable for applications where user logic is the limiting resource[1], yet not all embedded memory and DSP blocks are used. Several authors already proposed to employ embedded memory (Block RAM or BRAM) for AES [CG03, MM03] and there already exists work using the T-Table construction for FPGAs [FD01, CKVS06]. In contrast to these designs, our approach maps the *complete* AES data path onto embedded elements contained in Virtex-5 FPGAs. This strategy provides most savings in logic and routing resources and results in the highest data throughput on FPGAs reported in open literature.

More precisely, we demonstrate that an optimal AES module can be created from a combination of two 36 Kbit BlockRAM (BRAM) and four DSP slices in Virtex-5 FPGAs. This basic module comprises of eight pipeline stages and returns a single 32 bit column of an AES round each cycle. Since the output can be combined with the input in a feedback loop, this module is sufficient to compute the full AES output in iterative operation. Alternatively, the basic module can be replicated four times extending the data path to 128 bit to compute a full AES round resulting in a reduced number of iterations. This 128-bit design can be unrolled ten times for a fully pipelined operation of the AES block cipher. For reasons of comparability with other designs we do not directly include the key expansion function in these designs but instead, we provide a separate circuit for precomputing the required subkeys which can be combined with all three implementations. This project was done as joint work with Saar Drimer [DGP08] who did most of the implementations (except for the key schedule) as well as simulation of the entire design. Moreover, Saar also elaborated on suitable modes of operations and authentication methods (e.g., CMAC) for our design. See [Dri09] for further details.

2.2 Previous Work

Since the U.S. NIST adopted the Rijndael cipher as the AES in 2001, many hardware implementations have been proposed both for FPGAs and ASICs. Most AES designs are usually straightforward implementations of a single AES round or loop-unrolled, pipelined architectures for FPGAs utilizing a vast amount of user logic elements [EYCP01, JTS03, IKM00]. Particularly, the required 8×8 S-Boxes of the AES are mostly implemented in the Lookup Tables (LUT) of the user logic usually requiring large portions of the reconfigurable logic. For example, the authors of [SRQL03b] report 144 LUTs (4-input LUTs) to implement a single AES S-Box what accumulates to 2304 LUTs for a single AES round. More advanced approaches [MM01, SRQL03b, CG03, CKVS06] used the on-chip memory components of FPGAs, implementing the S-Box tables in separate RAM sections on the device. Since RAM capacities were limited in previous generations of FPGAs, the majority of implementations only mapped the 8×8 S-Box into the memory while all other AES operations like ShiftRows, MixColumns and the AddRoundKey are realized using traditional user logic, and proved costly in terms of

[1] Note that a very large percentage of all FPGA designs are restricted either by lack of logic or routing resources.

flip-flops and LUTs.
Since it is not in the scope of this thesis to review all available AES implementations for FPGAs and ASICs (see, for example [Jär08], for a survey of AES implementations), we will only review few designs with relevance to our work. We will now discuss and categorize published AES implementations according to their performance and resource consumption (and implicitly, if a small 8 bit or wide 32 bit data-path is used).

- *AES optimized for constrained resources:* AES implementations designed for area efficiency are mostly based on an 8 bit data path and use shared resources for key expansion and round computations. Such as design is presented by Good and Benaissa [GB05] which requires 124 slices and 2 BRAMs of a Xilinx Spartan-II XC2S15(-6) yielding an encryption throughput of 2.2 MBit/s. Small implementations with a 32 bit data path exist as well: the AES implementation by Chodowiec and Gaj [CG03] on a Xilinx Spartan-II 30(-6) consumes 222 slices and 3 embedded memories and provides an encryption rate of 166 Mbit/s. A similar concept was implemented in [RSQL04] where AES was realized on a more recent Xilinx Spartan-3 50(-4) with 163 slices and a throughput of 208 Mbit/s. Fischer and Drutarovský [FD01] proposed an economic AES implementation on an Altera ACEX 1K100(-1) device FPGAs using the 32-bit T-table technique. Their encryptor/decryptor provided a throughput of 212 Mbit/s using 12 embedded memory blocks and 2,923 logical elements.

- *Balanced Designs:* Balanced designs denote implementations which focus on area-time efficiency. In most cases, hardware for handling a single round of AES with a 32 or 128 bit data path is iteratively used to compute the required total number of AES rounds (depending on the key size). In the same work as mentioned above, Fischer and Drutarovský proposed a faster T-table implementation for a single round based on an Altera APEX 1K400(-1) taking 86 embedded memory blocks and 845 logical elements which provides a throughput of 750 Mbit/s. Standaert *et al.* [SRQL03b] present an even faster AES round design solely implemented in user logic: they report their design on an Xilinx Virtex-E 3200(-8) to achieve a throughput of 2.008 GBit/s with 2257 slices. Recently, Bulens *et al.* [BSQ+08] presented an AES design that takes advantage of the slice structure and 6-input LUTs of the Virtex-5 but it does not use any BRAM or DSP blocks. Further designs for Virtex-5 FPGAs can only be obtained from commercial companies, e.g., we will here refer to implementations by Algotronix [Alg07] and Heliontech [Hel07, v2.3.3].

- *Designs targeting High Performance:* Architecture with the goal to achieve maximum performance usually make thorough use of pipelining techniques, i.e., all AES rounds are unrolled in hardware and can be processed in parallel. McLoone *et al.* [MM03] discuss an AES-128 implementation based on the Xilinx Virtex-E 812(-8) device using 2,457 CLBs and 226 block memories providing an overall encryption rate of 12 Gbit/s. Hodjat and Verbauwhede [HV04] report an AES-128 implementation with 21.54 Gbit/s throughput using 5,177 slices and 84 BRAMs on a Xilinx Virtex-II Pro 20(-7) FPGA. Järvinen *et al.* [JTS03]

shows how to achieve a high throughput even without use of any BRAMs on a Xilinx Virtex-II 2000(-5) at the cost of additional CLBs: their design takes 10750 slices and provides an encryption rate of 17.8 GBit/s. Finally, Chaves *et al.* [CKVS06] also use the memory-based T-Table implementation on a Virtex-II Pro 20(-7) and provide a design of a single iteration and a loop unrolled AES based on a similar strategy as ours.

To our knowledge, only few implementations [FD01, RSQL04, CKVS06] have transferred the software architecture based on the T-table to FPGAs. Due to the large tables and the restricted memory capacities on those devices, certain functionality must be still encoded in user logic up to now (e.g., the multiplication elimination required in the last AES round, see 2.3). The new features of Virtex-5 devices provide wider memories and more advanced logic resources. Our contribution is the first T-table-based AES-implementation that efficiently uses mostly device-specific features minimizing the need for generic logic elements. We will provide three individual solutions that address each of the design categories mentioned above – minimal resource usage, area-time efficiency and high-throughput.

2.3 Mathematical Background

We will now briefly review the operation of the AES block cipher. AES was designed as a Substitution-Permutation Network (SPN) and uses between 10, 12 or 14 rounds (depending on the key length with 128, 192 and 256 bit, respectively) for encryption and decryption of one 128 bit block. In a single round, the AES operates on all 128 input bits. Fundamental operations of the AES are performed based on byte-level field arithmetic over the Galois Field $GF(2^8)$ so that operands can be represented in 8 bit vectors. Processing these 8 bit vectors serially allows implementations on very small processing units, while 128 bit data paths allow for maximum throughput. The output of such a round, or state, can be represented as a 4×4 matrix of bytes. For the remainder of this chapter, A denotes the input block consisting of bytes $a_{i,j}$ in columns C_j and rows R_i, where $i, j = 0..3$.

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

Four basic operations process the AES state A in each round::

(1) *SubBytes*: all input bytes of A are substituted with values from a non-linear 8×8 bit S-Box.

(2) *ShiftRows*: the bytes of rows R_i are cyclically shifted to the left by 0, 1, 2 or 3 positions.

(3) *MixColumns*: columns $C_j = (a_{0,j}, a_{1,j}, a_{2,j}, a_{3,j})$ are matrix-vector-multiplied by a matrix of constants in $GF(2^8)$.

(4) *AddRoundKey*: a round key K_i is added to the input using $GF(2^8)$ arithmetic.

The sequence of these four operations defines an AES round, and they are iteratively applied for a full encryption or decryption of a single 128 bit input block. Since some of the operations above rely on $GF(2^8)$ arithmetic we are able to combine them into a single complex operation. In addition to the Advanced Encryption Standard, an alternative representation of the AES operation for software implementations on 32 bit processors was proposed in [DR02, Section 4.2] based on the use of large lookup tables. This approach requires four lookup tables with 8 bit input and 32 bit output for the four round transformations, each the size of 8 Kbit. According to [DR02], these transformation tables T_i with $i = 0..3$ can be computed as follows:

$$T_0[x] = \begin{bmatrix} S[x] \times \texttt{02} \\ S[x] \\ S[x] \\ S[x] \times \texttt{03} \end{bmatrix} \qquad T_1[x] = \begin{bmatrix} S[x] \times \texttt{03} \\ S[x] \times \texttt{02} \\ S[x] \\ S[x] \end{bmatrix}$$

$$T_2[x] = \begin{bmatrix} S[x] \\ S[x] \times \texttt{03} \\ S[x] \times \texttt{02} \\ S[x] \end{bmatrix} \qquad T_3[x] = \begin{bmatrix} S[x] \\ S[x] \\ S[x] \times \texttt{03} \\ S[x] \times \texttt{02} \end{bmatrix}$$

In this notation, $S[x]$ denotes a table lookup in the original 8×8 bit AES S-Box (for a more detailed description of this AES optimization see NIST's FIPS-197 [Nat01]). The last round, however, is unique since it omits the MixColumns operation, so we need to give it special consideration. There are two ways for computing the last round, either by "reversing" the MixColumns operation from the output of a regular round by another multiplication in $GF(2^8)$, or creating dedicated T-tables for the last round. The latter approach will allow us to maintain the same data path for all rounds, so – since Virtex-5 devices provide larger memory blocks than former devices – we chose this method and denote these T-tables as $T_{[j]'}$. With all T-tables at hand, we can redefine all transformation steps of a single AES round as

$$E_j = K_{r[j]} \oplus T_0[a_{0,j}] \oplus T_1[a_{1,(j+1 \bmod 4)}] \oplus T_2[a_{2,(j+2 \bmod 4)}] \oplus T_3[a_{3,(j+3 \bmod 4)}] \tag{2.1}$$

where $K_{r[j]}$ is a corresponding 32 bit subkey and E_j denotes one of four encrypted output *columns* of a full round. We now see that based on only four T-table lookups and four XOR operations, a 32 bit column E_j can be computed. To obtain the result of a full round, Equation (2.1) must be performed four times with all 16 bytes.

Input data to an AES encryption can be defined as four 32 bit column vectors $C_j = (a_{0,j}, a_{1,j}, a_{2,j}, a_{3,j})$ with the output similarly formatted in column vectors. According to Equation (2.1), these input column vectors need to be split into individual bytes since all bytes are required for the computation steps for different E_j. For example, for column $C_0 = (a_{0,0}, a_{1,0}, a_{2,0}, a_{3,0})$ the first byte $a_{0,0}$ is part of the computation of E_0, the second byte $a_{1,0}$ is used in E_3, etc. Since fixed (and thus simple) data paths are preferable in hardware

implementations, we have rearranged the operands of the equation to align the bytes according to the input columns C_j when feeding them to the T-table lookup. In this way, we can implement a unified data path for computing all four E_j for a full AES round. Thus, Equation (2.1) transforms into

$$\begin{aligned}
E_0 &= K_{r[0]} \oplus T_0(a_{0,0}) \oplus T_1(a_{1,1}) \oplus T_2(a_{2,2}) \oplus T_3(a_{3,3}) = (a'_{0,0}, a'_{1,0}, a'_{2,0}, a'_{3,0}) \\
E_1 &= K_{r[1]} \oplus T_3(a_{3,0}) \oplus T_0(a_{0,1}) \oplus T_1(a_{1,2}) \oplus T_2(a_{2,3}) = (a'_{0,1}, a'_{1,1}, a'_{2,1}, a'_{3,1}) \\
E_2 &= K_{r[2]} \oplus T_2(a_{2,0}) \oplus T_3(a_{3,1}) \oplus T_0(a_{0,2}) \oplus T_1(a_{1,3}) = (a'_{0,2}, a'_{1,2}, a'_{2,2}, a'_{3,2}) \\
E_3 &= K_{r[3]} \oplus T_1(a_{1,0}) \oplus T_2(a_{2,1}) \oplus T_3(a_{3,2}) \oplus T_0(a_{0,3}) = (a'_{0,3}, a'_{1,3}, a'_{2,3}, a'_{3,3})
\end{aligned}$$

where $a_{i,j}$ denotes an input byte, and $a'_{i,j}$ the corresponding output byte after the round transformation. However, the unified input data path still requires a look-up to all of the four T-tables for the second operand of each XOR operation. For example, the XOR component at the first position of the sequential operations E_0 to E_3 and thus requires the lookups $T_0(a_{0,0}), T_3(a_{3,0}), T_2(a_{2,0})$ and $T_1(a_{1,0})$ (in this order) and the corresponding round key $K_{r[j]}$. Though operations are aligned for the same input column now, it becomes apparent that the bytes of the input column are not processed in canonical order, i.e., bytes need to be swapped for each column $C_j = (a_{0,j}, a_{1,j}, a_{2,j}, a_{3,j})$ first before being fed as input to the next AES round. The required byte transposition is reflected in the following equations:

$$\begin{aligned}
C_0 &= (a'_{0,0}, a'_{3,0}, a'_{2,0}, a'_{1,0}) \\
C_1 &= (a'_{1,1}, a'_{0,1}, a'_{3,1}, a'_{2,1}) \\
C_2 &= (a'_{2,2}, a'_{1,2}, a'_{0,2}, a'_{3,2}) \\
C_3 &= (a'_{3,3}, a'_{2,3}, a'_{1,3}, a'_{0,3})
\end{aligned} \tag{2.2}$$

Note that the given transpositions are static so that they can be efficiently hardwired in our implementation.

Finally, we need to consider the XOR operation of the input key and the input 128 bit block which is done prior to the round processing. Initially, we will omit this operation when reporting our results for the round function. However, adding the XOR to the data path is simple, either by modifying the AES module to perform a sole XOR operation in a preceding cycle, or – more efficiently – by just adding an appropriate 32-bit XOR which processes the input columns prior being fed to the round function.

2.3.1 Decryption

Although data encryption and decryption semantically only reverses the basic AES operations, the basic operations itself require different treatment so typically separate hardware components and significant logic overhead is necessary to support both. With our approach, all primitive operations are encoded into T-tables for encryption, so that we can apply a similar strategy for decryption by creating tables representing the inverse cipher transformation. Hence, we can basically support an encryptor and decryptor engine with the same circuit by only swapping the

values of the transformation tables and slightly modifying the input. As with Equation (2.1), decryption of columns D_j can be expressed by the following set of equations:

$$
\begin{aligned}
D_0 &= K_{r[0]} \oplus I_0(a_{0,0}) \oplus I_1(a_{1,3}) \oplus I_2(a_{2,2}) \oplus I_3(a_{3,1}) = (a'_{0,0}, a'_{1,0}, a'_{2,0}, a'_{3,0}) \\
D_3 &= K_{r[3]} \oplus I_3(a_{3,0}) \oplus I_0(a_{0,3}) \oplus I_1(a_{1,2}) \oplus I_2(a_{2,1}) = (a'_{0,3}, a'_{1,3}, a'_{2,3}, a'_{3,3}) \\
D_2 &= K_{r[2]} \oplus I_2(a_{2,0}) \oplus I_3(a_{3,3}) \oplus I_0(a_{0,2}) \oplus I_1(a_{1,1}) = (a'_{0,2}, a'_{1,2}, a'_{2,2}, a'_{3,2}) \\
D_1 &= K_{r[1]} \oplus I_1(a_{1,0}) \oplus I_2(a_{2,3}) \oplus I_3(a_{3,2}) \oplus I_0(a_{0,1}) = (a'_{0,1}, a'_{1,1}, a'_{2,1}, a'_{3,1})
\end{aligned}
$$

This requires the following inversion tables (I-Tables), where S^{-1} denotes the inverse 8×8 S-Box for the AES decryption:

$$
I_0[x] = \begin{bmatrix} S^{-1}[x] \times \texttt{0E} \\ S^{-1}[x] \times \texttt{09} \\ S^{-1}[x] \times \texttt{0D} \\ S^{-1}[x] \times \texttt{0B} \end{bmatrix} \qquad I_1[x] = \begin{bmatrix} S^{-1}[x] \times \texttt{0B} \\ S^{-1}[x] \times \texttt{0E} \\ S^{-1}[x] \times \texttt{09} \\ S^{-1}[x] \times \texttt{0D} \end{bmatrix}
$$

$$
I_2[x] = \begin{bmatrix} S^{-1}[x] \times \texttt{0D} \\ S^{-1}[x] \times \texttt{0B} \\ S^{-1}[x] \times \texttt{0E} \\ S^{-1}[x] \times \texttt{09} \end{bmatrix} \qquad I_3[x] = \begin{bmatrix} S^{-1}[x] \times \texttt{09} \\ S^{-1}[x] \times \texttt{0D} \\ S^{-1}[x] \times \texttt{0B} \\ S^{-1}[x] \times \texttt{0E} \end{bmatrix}
$$

Obviously, compared to encryption, the input to the decryption equations is different at two positions for each decrypted column D_j. But, instead of changing the datapath from the encryption function, we can change the order in which the columns D_j are computed so that instead of computing E_0, E_1, E_2, E_3 for encryption, we determine the decryption output in the column sequence D_0, D_3, D_2, D_1. Preserving the data path by only changing the content of the tables will allow us to use (nearly) the same circuit for both functions, as we shall see in Section 2.5.

2.3.2 Key Schedule

The AES uses a key expansion operation to derive ten subkeys K_r (12 and 14 for AES-192 and AES-256, respectively) from the main key, where r denotes the corresponding round number, to avoid simple related-key attacks. There are two different ways to implement the key schedule: first using a precomputation phase which is more common and expands all subkeys prior encryption. Alternatively, it is possible to perform the key schedule on-the-fly, i.e., simultaneously to the round encryption/decryption. However, during decryption all subkeys must be provided in reverse order, i.e., the main key needs to be completely expanded first so that the decryption process is able to start with the last subkey to invert the last round's encryption (what has previously been encrypted with exactly this last key). Obviously, this process is particularly expensive when a key derivation scheme is used which generates the keys simultaneously to the round processing. Thus, precomputing keys and storing them in an individual memory is the preferred way for a design supporting both encryption *and* decryption within the same circuit.

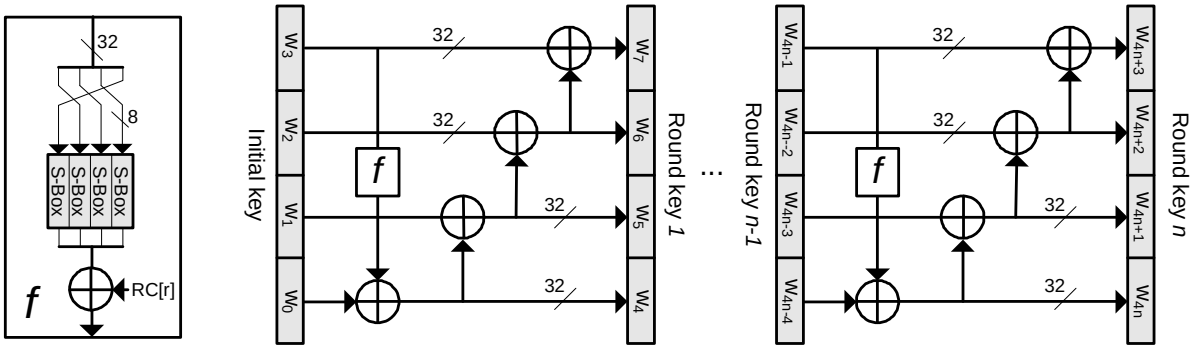


Figure 2.1: The key schedule derives subkeys for the round computations from a main key.

The first operation of AES is a 128 bit XOR of the main key K_0 with the 128 bit initial plaintext block. During expansion, each subkey is split into four individual 32 bit words $K_r[j]$ for $j = 0 \ldots 3$. The first word $K_r[0]$ of each round subkey is extensively transformed using byte-wise rotations and mappings along the same non-linear AES S-Box already used for encryption. All subsequent words for $j = 1 \ldots 3$ are determined by an exclusive-or operation with the previous subkey words $K_r[j-1] \oplus K_{(r-1)}[j]$. Figure 2.1 depicts the full key schedule.

2.4 Embedded Elements of Modern FPGAs

In this section, we will introduce the functionalities of embedded elements which come with (most) modern FPGAs. Note that we will make use of the embedded elements in several parts of this thesis (cf. also to Chapter 3 and Chapter 7). Since their invention in 1985 [Xil08a], FPGAs came up providing a sea of generic, reconfigurable logic. Although devices grew larger and larger, there are still function blocks which should be placed externally in separate peripheral devices since it is inefficient to implement them with generic logic. Examples of thesis functions blocks are large , hard microprocessors, and fast serial transceivers. Thus, FPGA manufacturers integrate more and more of these dedicated function blocks into modern devices to avoid the necessity of extensions on the board. Figure 2.2 depicts the simplified structure of recent Xilinx Virtex-5 FPGAs including separate columns of additional function blocks for memory (BRAM) and arithmetic operations (DSP blocks). Note that other FPGA classes, like Spartan-3 or Virtex-4 have a similar architecture despite variations in dimensions and features of the embedded elements. In Virtex-4 and Virtex-5 devices, the DSP blocks are grouped in pairs that span the height of four or five configurable logic blocks (CLB), respectively. The dual-ported BRAM matches the height of the pair of DSP blocks and supports a fast datapath between memory and the DSP elements.

In particular interest of this thesis is the use of these memory elements and DSP blocks for efficient boolean and integer arithmetic operations with low signal propagation time. More precisely, large devices of Xilinx's Virtex-4 and Virtex-5 class are equipped with up to thousand individual function blocks of these dedicated memory and arithmetic units. Originally, the

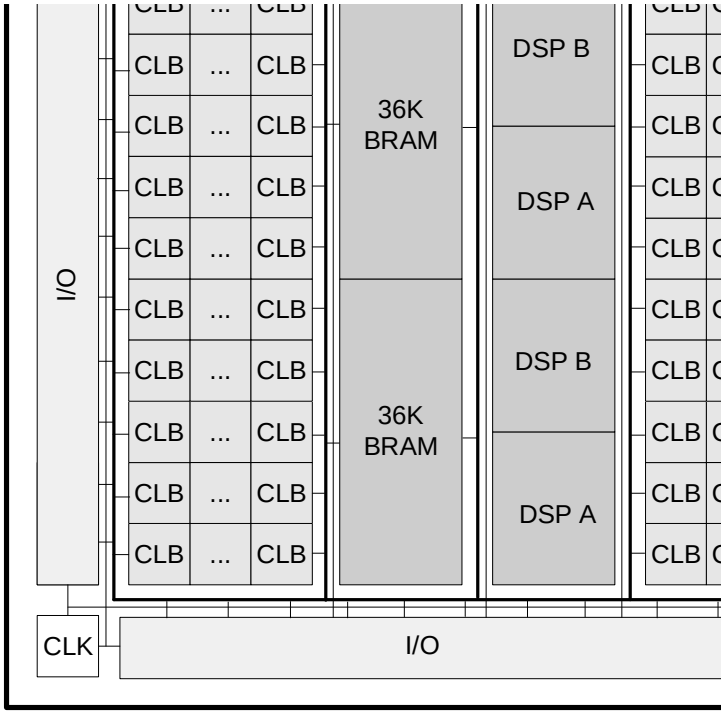


Figure 2.2: Simplified structure of Xilinx Virtex-5 FPGAs.

integrated DSP blocks – as indicated by their name – were designed to accelerate Digital Signal Processing (DSP) applications, e.g., Finite Impulse Response (FIR) filters, etc. However, these arithmetic units can be programmed to perform universal arithmetic functions not limited to the scope of DSP filters; they support generic multiplication, addition and subtraction of (un)signed integers. Dependant on the FPGA class common DSP component comprises an l_M-bit signed integer multiplier coupled with an l_A-bit signed adder where the adder supports a larger data path to allow accumulation of multiple subsequent products. Exactly, Xilinx Virtex-4 FPGAs support 18 bit unsigned integer multiplication (yielding 36 bit products) and three-input addition, subtraction or accumulation of unsigned 48 bit integers. Virtex-5 devices offer support for even wider 25×18 bit multiplications. Since DSP blocks are designed as an embedded element in FPGAs, there are several design constraints which need to be obeyed for maximum performance with the remaining logic, e.g., the multiplier and adder block should be surrounded by pipeline registers to reduce signal propagation delays between components. Furthermore, since they support different input paths, DSP blocks can operate either on external inputs A, B, C or on internal feedback values from accumulation or result P_{j-1} from a neighboring DSP block. Figure 2.3 shows the generic DSP-block and a small selection of possible modes of operations available in recent Xilinx Virtex-4/5 FPGA devices [Xil08b] and used in this thesis.

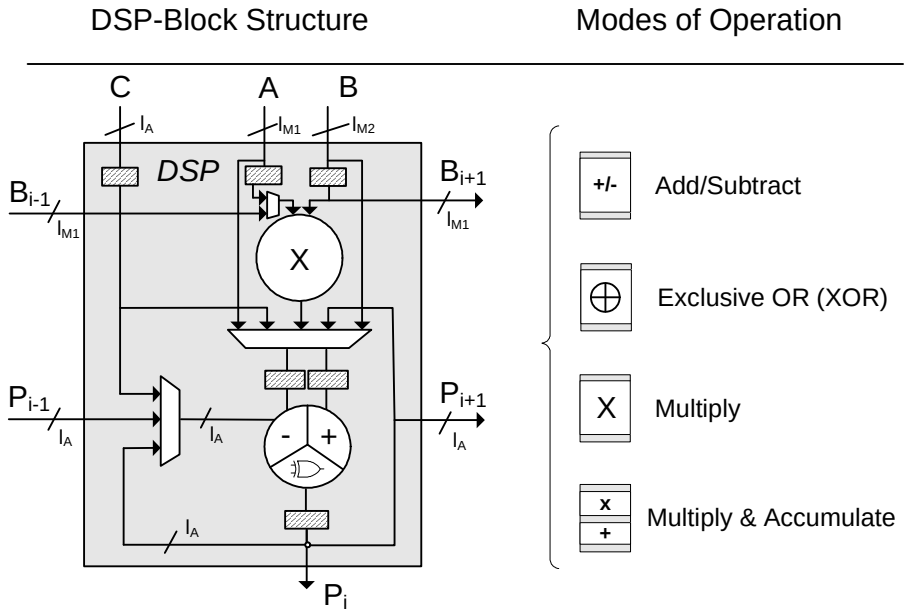


Figure 2.3: Generic and simplified structure of DSP-blocks of advanced FPGA devices.

2.5 Implementation

In Section 2.3, we have introduced the T-table method for implementing the AES round most suitable for 32 bit microprocessors. Now, we will demonstrate how to adapt this technique into modern reconfigurable hardware devices in order to achieve high throughput for modest amounts of resources. For our implementations, we use Xilinx Virtex-5 FPGAs and make intensive use of the embedded elements to achieve a design beyond traditional LUTs and registers. Our architecture relies on dual ported 36 Kbit BlockRAMs (BRAM) (with independent address and data buses for the same stored content) and DSP blocks. The fundamental idea of this work is that the 8 to 32 bit lookup followed by a 32 bit XOR AES operation perfectly matched this architectural alignment of Virtex-5 FPGAs. Based on these primitives, we developed a basic AES module that performs a quarter (one column) of an AES round transformation given by Equation (2.1). Figure 2.4 depicts such a mapping of Equation (2.1) into embedded functions blocks of a Virtex-5 FPGA. The chosen design is optimal for Virtex-5 so that it allows efficient placing and routing of components such that it can operate at the maximum device frequency of 550 MHz. Furthermore, our basic module is designed such that it can be replicated for higher throughput.

2.5.1 Basic Module

Figure 2.4 shows a first design idea which does not yet take any input transformations for different columns or rounds into account. More precisely, we yet need to consider alignment of

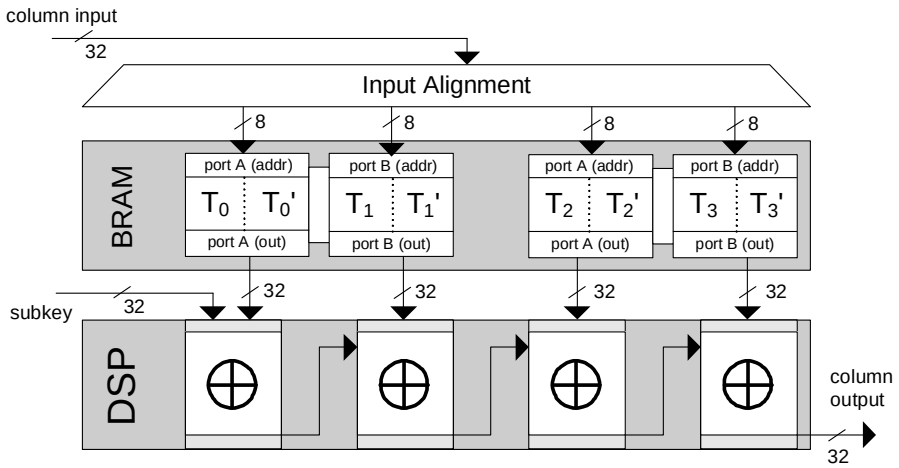


Figure 2.4: The mapping of AES column operations onto functional components of modern Virtex-5 devices. Each dual ported BRAM contains four T-tables, including separate tables for the last round. Each DSP block performs a 32 bit bit-wise XOR operation.

the inputs: here, four bytes $a_{i,j}$ are selected from the current state A at a time and passed to the BRAMs for the T-table lookup. Since the order of bytes $a_{i,j}$ vary for each column computation E_j, this requires a careful design of the input logic since it need to support selection from all four possible byte positions of each 32-bit column input. Hence, instead of implementing a complex input logic, we modified the order of operations according to Equations (2.2) exploiting that addition in $GF(2^m)$(i.e., XOR) is a commutative operation. When changing the order of operations dynamically for each computation of E_j, this requires that all four T-table lookups with their last-round T-table counterparts are stored in each BRAM. However, that would require to fit a total of eight 8 Kbit T-tables in a single 36 Kbit dual-port RAM. As discussed in Section 2.3, for performance and resource efficiency reasons we opted against adding out the MixColumn operations from the stored T-tables and preferred a solution so that all BRAM can provide all eight required tables. Utilizing the fact that all T-tables are byte-wise transpositions of each other, we can produce the output of T_1, T_2 and T_3 by cyclically byte-shifting of the BRAM's output for T-table T_0. Using this observation, we only store T_0 and T_2 and their last-round counterparts $T_{0'}$ and $T_{2'}$ in a single BRAM. Using a single byte circular right rotation $(a, b, c, d) \rightarrow (d, a, b, c)$, T_0 becomes T_1, and T_2 becomes T_3 and the same for the last round's T-tables. In hardware, this only requires a 32 bit 2:1 multiplexer at the output of each BRAM with a select signal from the control logic. For the last round, a control bit is connected to a high order address bit of the BRAM to switch from the regular T-table to the last round's T-table. The adapted design can be seen in Figure 2.5. A dual-port 32 Kbit BRAM with three control bits, and a 2:1 32 bit mux allows us to output all T-table combinations. Using two such BRAMs with identical content, we get the necessary lookups for four columns, each capable of performing all four T-table lookups in parallel.

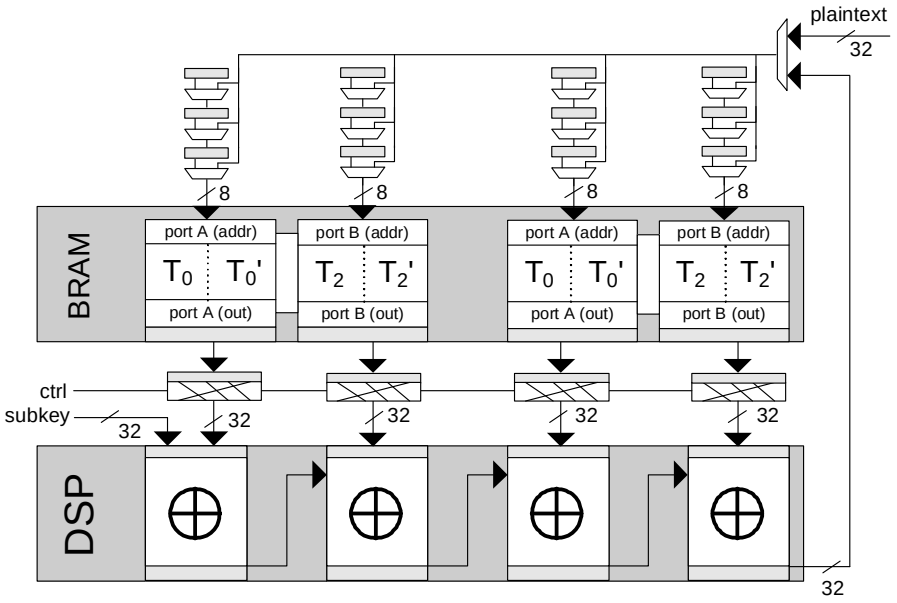


Figure 2.5: The complete basic AES module consisting of 4 DSP slices and 2 dual-ported Block Memories. Tables T_1 and T_3 are constructed on-the-fly using byte shifting from tables T_0 and T_2 in the block memory, respectively.

Note that both the BRAMs and DSP blocks provide internal input and output registers for pipelining along the data path so that we include these registers without occupation of any flip-flops in the fabric. At this point, we already had six pipeline stages that could not have been easily removed if our goal was high throughput. Instead of trying to reduce pipeline stages for lower latency, we opted to add two more so that we are able to process two input blocks at the same time, doubling the throughput for separate input streams. One of these added stages is the 32 bit register after the 2:1 multiplexer that shifts the T-tables at the output of the BRAM.

A full AES operation is implemented by operating the basic construct with an added feedback scheduling in the data path.

Figure 2.6 shows the eight pipeline stages where $K_{r[i]}$ denotes the ith subkey of round r and D_j the 32 bit table output produced by the four BRAM ports. The first column output E_0 becomes available after the eighth clock cycle and is fed back as input for the second round. For the second round, the control logic switches the 2:1 input multiplexer for the feedback path rather than the external input. The exact data flow is given in detail in Table A.1 which can be found in the appendix. In the eight pipeline stages we can process two separate AES blocks, since we only need 4 stages to process the 128 bit of one block. This allows us to feed two consecutive 128 bit blocks one after another, in effect doubling our throughout without any additional complexity.

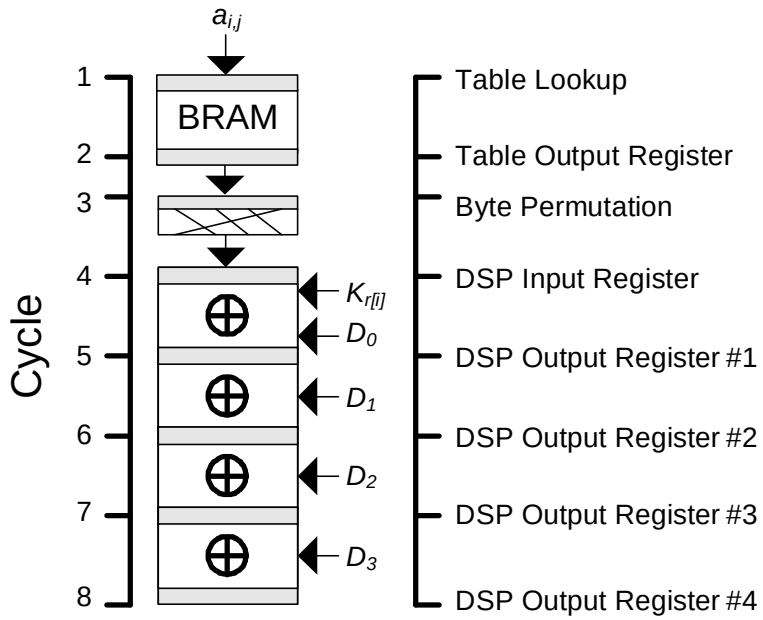


Figure 2.6: Pipeline stages to compute the column output of an AES round.

We also investigated on an alternative design approach for the basic AES module. Instead of cascading several DSP units to use and create a data path with eight pipeline stages, we chose to process each column E_j with the j-th DSP slice only by selecting an operation mode for the DSP slice which accumulates all input values using an internal feedback path (i.e., accumulation in $GF(2^m)$). We found, however, that this requires the input of a key to each DSP block, extra control logic, different operating modes for the DSP (e.g., for restarting accumulation), and a 32 bit 4:1 mux to choose between the output of each DSP for feeding the input to the next round. Due to higher resource cost and worse routing results, we prefer to stick to the original design.

Up to now we focused on the encryption process, though decryption is quite simply achieved with minor modifications to the circuit. As the T-tables are different for encryption and decryption, storing them all would require double the amount of storage what is not desirable. Recall, however, that any T_i can be converted into T_j simply by shifting the appropriate amount of bytes. The most straightforward modification to the design is to replace the 32 bit 2:1 mux at the output of the BRAM with a 4:1 mux such that all byte transpositions can be created. Then, we load the BRAMs with T_i^E, $T_{i'}^E$, T_i^D and $T_{i'}^D$, where T^E and T^D denote encryption and decryption T-tables, respectively, with their corresponding last round counterparts. Note, that this does not necessarily increase the data path due to the 6-input LUTs in the CLBs of a Virtex-5 device. Based on 6-input LUTs, a 4:1 multiplexer can be as efficiently implemented as a 2:1 multiplexer with only a single stage of logic. An alternative is to dynamically reconfigure the content of the BRAMs with the decryption T-tables; this can be done from an external source,

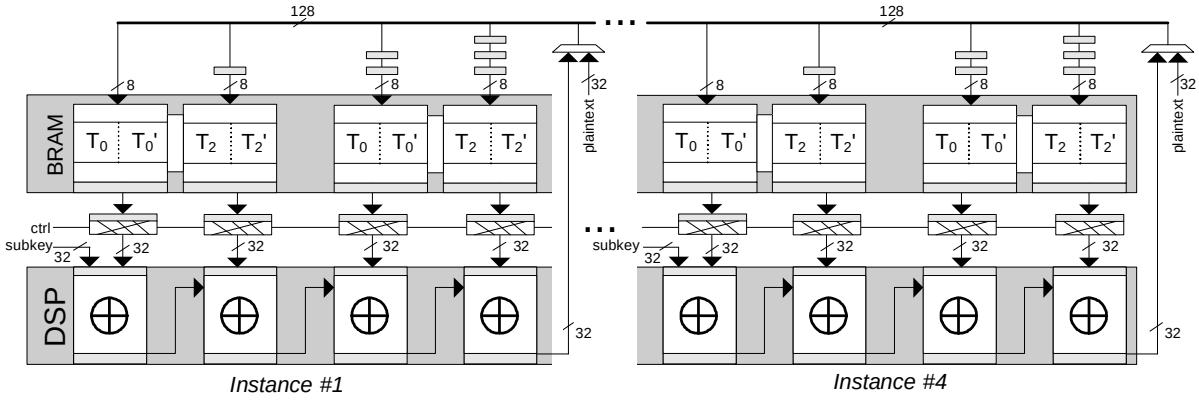


Figure 2.7: Four instances of the basic structure in hardware allow all AES columns being processed in parallel (128 bit data path).

or even from within the FPGA using the internal configuration access port (ICAP) [Xil06] with a storage BRAM for reloading content through the T-table BRAMs' data input port.

Finally, the AES specification requires an initial key addition of the input with the main key which has not covered by the AES module so far. Most straightforward, this can be done by adding one to four DSP blocks (alternatively, the XOR elements can be implemented in CLB logic) as a prestage to the round operation.

2.5.2 Round and Loop-Unrolled Modules

Since the single AES round requires the computation of four 32 bit columns, we can replicate the basic construct four times and add 8, 16, and 24 bit registers at the inputs of the columns. This is shown in Figure 2.7 where all instances are connected to a 128 bit bus (32 bits per instance) of which selected bytes are routed to corresponding instances by fixed wires. Note that only one byte per 32 bit column output remains within the same instance, the other three bytes will be processed by the other instances in the next round. The latency of this construct is still 80 clock cycles as before, but allows us to interleave eight 128 bit inputs instead of two. In contrast to the basic module, however, the input byte arrangements allow that the T-tables be static so the 32 bit 2:1 multiplexers are no longer required. This simplifies the data paths between the BRAMs and DSP blocks since the shifting can be fixed in routing. The control logic is simple as well, comprising of a 3 bit counter and a 1 bit control signal for choosing the last round's T-tables.

Finally, we implemented a fully unrolled AES design for achieving maximum throughput by connecting ten instances of the round design presented above. We yield an architecture with an 80-stage pipeline, producing a 128 bit output every clock cycle at a resource consumption of 80 BRAMs and 160 DSP blocks. One advantage of this approach is the savings for control signals since the full process is unrolled and thus completely hardwired in logic.

2.5.3 Key Schedule Implementation

Considering the key schedule, many designers (e.g., [BSQ+08]) prefer a shared S-Box and/or datapath for deriving subkeys and the AES round function. This approach needs additional multiplexing and control signals to switch the central data path between subkey computations and data encryption which may lead to decreased performance in practice. Furthermore, key precomputation is mostly preferred over on-the-fly key expansion because the first relaxes the constraints on data dependencies, i.e., the computation is only dependent on the availability of the previous state (plaintext) and not additionally on completion of key computations.

In case that high throughput is not required but the key schedule needs to be precomputed on chip without adversely increasing logic resource utilization, our basic AES module can be modified to support the key generation. Remember that we already store T-tables $T_{[0..3]'}$ for the last round in the BRAMs without the MixColumns operation so that the values of these tables are basically a byte-rotated 8 bit S-Box value. These values are perfectly suited for generating a 32 bit round key from S-Box lookups and our data path has been specifically designed for 32 bit XOR operations based on the DSP unit. Hence, with additional input multiplexers, control logic and a separate BRAM as key-store, we can integrate a key scheduler in our existing design. However, although this is possible, the additional overhead (i.e., additional multiplexers) will potentially degrade the performance of the AES rounds.

The second approach for the key schedule is a dedicated circuit to preserve the regularity of the basic module and the option to operate the design at maximum device frequency. For a minimal footprint, we propose to add another dual-ported BRAM to the design used for storing the expanded 32 bit subkeys (44 words for AES-128), the round constants (10 32 bit values) and S-Box entries with 8 bit each. The design of our key schedule implementation is shown in Figure 2.8: port A of the BRAM is 32 bit wide which feeds the subkeys to the AES module, while port B is configured for 8 bit I/O enabling a minimal data path for the key expansion function. With an 8 bit multiplexer, register and XOR connected to port B data output, we can construct a minimal and byte-oriented key schedule that can compute the full key expansion.

The sequential and byte-wise nature of this approach for loading and storing the appropriate bytes from and to the BRAM requires a complex state machine. Recall that the BRAM provides 36 Kbits of memory of which 1408 to 1920 bits are required for subkeys (for AES-128 and AES-256, respectively), 2048 bits for S-Box entries and 80 bits for round constants, so the BRAM can still be used to store further data. Thus, we have decided that the most area economic approach is to encode all the required memory addresses as well as control signals for multiplexers and registers as 32 bit instructions, and store these instruction words in the yet unused portion of the BRAM. This method also ensures a constant and uniform signal propagation in all control signals since they do not need to be generated by combinatorial logic but loaded (and hardwired) from the BRAM. In particular, complex state machines and the latency within their combinatorial circuitry are usually the bottleneck of high-performance implementations since nested levels of logic to generate dozens of control signals are likely to emerge as the critical path. By encoding this complexity into the BRAM, we could avoid this performance degrade.

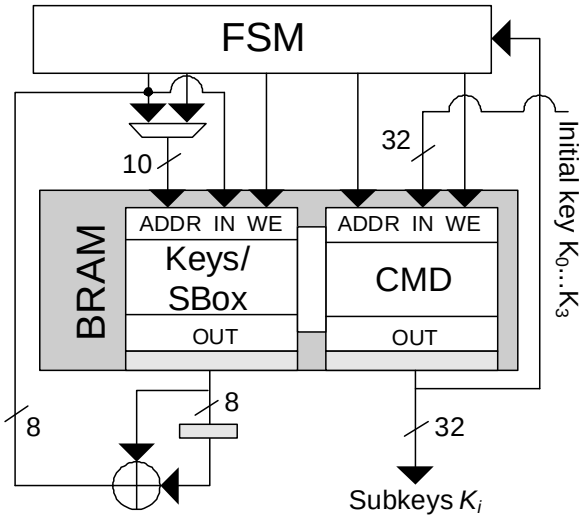


Figure 2.8: Block diagram of the key schedule implementation. Complex instructions of the finite state maschine, S-boxes, round constants and 32-bit subkeys are stored in the dual-port BRAM.

Like the basic AES module it can be operated at full device frequency of 550 MHz and with the complete key expansion function requiring 524 clock cycles for AES-128.

2.6 Results

Our designs target a Virtex-5 LX30 and SX95T devices at their fastest speed grade (-3) using Xilinx Synthesis Technology (XST) and the ISE 9.2 implementation flow. For simulation we used Mentor's ModelSim 6.2g for both behavioral and post place-and-route stages. In addition, the routes to input and output ports were ignored for timing ("TIG" constraint) during synthesis, as we consider the cores as a stand-alone function.

The basic AES module as shown in Figure 2.5 passed timing (post place-and-route) for a frequency just over 550 MHz, the maximum frequency rating of the device. The design requires the following resources: 247 flip-flops, 96 ($8 \cdot 3 \cdot 4$) for the input shift registers plus 128 ($4 \cdot 32$) for the pipeline stages in between the BRAMs and DSPs, with the rest used for control logic; 275 look-up tables, mostly functioning as multiplexers; and finally, two 36 Kbit dual-port BRAM (32 Kbit used in each) and four DSP blocks. We calculate throughput as follows: given that there are 80 processing cycles operating at 550 MHz and we maintain state of 256 bits in the pipeline stages, we achieve $550 \cdot 10^6 \cdot 256/80 = 1.76$ Gbit/s of throughput. This assumes that the pipeline stages are always full, meaning that the module is processing two 128 bit inputs at any given time; if only one input is processed, the throughput is halved. As we have mentioned, the eight pipeline stages were implemented for the purpose of interleaving two inputs or using a parallel mode of operation like Counter (CTR) mode, though the designer can remove pipeline

stages to reduce resources. Removing pipeline stages reduces latency, though it may also reduce the maximum frequency, so there is a trade-off that needs to be assessed according to the application.

In the round module the basic construct is used four times for a 128 bit-width interface. The maximum frequency reported by the tools post place-and-route was over 485 MHz, and it uses 621 flip-flops, 204 look-up tables, 8 36 Kbit BRAMs (32 Kbit used in each), and 16 DSP blocks. Notice that we expect at least $4 \cdot 48 + 4 \cdot 128 = 704$ registers but the tools report only 621. This is because the synthesizer tries to achieve a balanced FF-LUT ratio for better packing into slices so the 2- and 3-stage input shift registers for each basic cells are implemented in eight LUTs each. The latency of 80 clock cycles is the same as the previous design, though now we can maintain state of $128 \cdot 8 = 1024$ bits, thus giving us a throughput of $485 \cdot 10^6 \cdot 8 \cdot 128/80 = 6.21$ Gbit/s when processing eight input blocks. We can see that the complexity of this design reduces the maximum frequency and throughput, though hand placement of DSPs and BRAMs, along with matching the bit ordering to the routing can improve on this performance. As with the basic module, pipeline stages can be removed to minimize the use of logic resources if they are required for other functions and the highest throughput is not required.

Finally, the unrolled implementation produces 128 bits of output every clock cycle once the initial latency is complete. We have experimented with eliminating the pipeline stage between the BRAM and DSP to see if it adversely affects performance; this will save us 5,120 registers. We found that the performance degradation is minimal, with the added benefit of having an initial latency of only 70 clock cycles instead of 80. The resulting throughput is $430 \cdot 10^6 \cdot 128 = 55$ Gbit/s. This design operates at a maximum frequency of over 430 MHz and uses 992 flip-flops, 672 look-up tables, 80 36 Kbit BRAMs (only 16 Kbit in each for dec/enc or 32 Kbit for both), and 160 DSP blocks; the same balancing act of FF-LUT ratio by the synthesizer occurs here as well. There are very few flip-flops and LUTs compared to what is available in the large SX95T device: 1.68% and 1.14%, respectively, though we use 32% of BRAMs and 25% of DSP blocks.

Our results are summarized in Table 2.1. We extended the list of our findings with previous result available in the literature. However, please be aware of the unfairness of direct comparison. Due to the different architectures of Spartan-2/3 (S2/S3) and Virtex-2/E Pro (V2/V2P/VE) and Virtex-5 (V5) FPGAs we cannot directly compare soft metrics like "'slices"'. This is due to the different concepts of contained LUTs (6-input LUTs in Virtex-5 and 4-input LUTs in all others) as well as the different number of LUTs and flip-flops per slice (e.g., a slice in Spartan and Virtex-2 FPGAs consists a combination of 2 LUTs/FF but 4 LUTs/FF in Virtex-5 devices). Even the amount of memory contained in BRAMs is different: Virtex-5 FPGAs provide block memories capable to store 36 KBits of data, twice as much as with Virtex-2 devices. Beside device-specific differences, the implementations also target different applications and requirements: some can operate in more complex modes of operations, others include a key schedule in the data path or support natively encryption and decryption with the same circuit. This all leads to the conclusion that comparisons with other publications based on different FPGAs and application goals are mostly misleading, e.g., meaningful comparisons are only possible when

	Design	Dec/ Key	FPGA	Resources					f	Perf.
				slices	LUT	FF	BRAM	DSP	MHz	Gbit/s
Ours	**Basic**[a]	○/○	V5	93	274	245	2	4	550	1.76
	Round[a]	○/○	V5	277	204	601	8	16	485	6.21
	Unrolled	•/○	V5	428	672	992	80	160	430	55
Basic	Good *et al.*	•/•	S2	67	*n/a*	*n/a*	2	0	67	0.002
	Chodowiec *et al.*	•/•	S2	222	*n/a*	*n/a*	3	0	60	0.166
	Rouvroy *et al.*	•/•	S3	163	293	126	3	0	71	0.208
	Algotronix	○/○	V5	161	*n/a*	*n/a*	2	0	250	0.8
Round	Standaert *et al.*	○/•	VE	2257	3846	2517	2	0	169	2.008
	Helion	○/•	V5	349	*n/a*	*n/a*	0	0	350	4.07
	Bulens *et al.*	○/•	V5	400	*n/a*	*n/a*	0	0	350	4.1
	Chaves *et al.*	•/○	V2P	515	*n/a*	*n/a*	12	0	182	2.33
Unrolled	Kotturi *et al.*	•/○	V2P	10816	*n/a*	*n/a*	400	0	126	16
	Järvinen *et al.*	○/•	V2	10750	*n/a*	*n/a*	0	0	139	17.8
	Hodjat *et al.*	○/○	V2P	5177	*n/a*	*n/a*	84	0	168	21.5
	Chaves *et al.*	•/○	V2P	3513	*n/a*	*n/a*	80	0	272	34.7

[a] For "basic" and "round" implementations, decryption can be achieved by adding 32 bit muxes in the datapath between BRAM and DSP.

Table 2.1: Our results along with recent academic and commercial implementations. Decryption (Dec.) and Key expansion (Key) are included when denoted by •, by ○ otherwise. Note the structural differences between the FPGA types: Virtex-5 (V5) has 4 FF and 4 6-LUT per slice and a 36 Kbit BRAM, while Spartan-3 (S3), Virtex-E (VE), Virtex-II (PRO) (V2/V2P) has 2 FF and 2 4-LUT per slice and an 18 Kbit BRAM. Spartan-II (S2) devices only provide 4 KBit BRAMs.

the same device/technology is used and the compared cryptographic implementations comply to a predefined application setup or framework. Note that all these constraints on comparisons between FPGA implementations also apply to other results reported in the remainder of this thesis.

Note that our results for the AES modules are all based on the assumption that the set of subkeys are externally provided. In case that all subkeys should be generated on the same device, these modules can be augmented with the key schedule precomputing all subkeys and storing them in a dedicated BRAM. As shown in Section 2.5.3, our key schedule is optimized for a minimal footprint and allows operation at maximum device frequency of 550 MHz. The complexity of the state machine, which is the most expensive part in terms of logic, is mostly hidden within the encoded 32 bit instructions stored in the BRAM. Hence, since only a small stub of the state machine in the user logic is required to address the individual instructions words, the overall resources consumption of the full key schedule is only 1 BRAM, 55 LUTs

Key schedule	**Resources** slices	LUT	FF	BRAM	DSPs	f (MHz)	**Cycles**
AES-128							524
AES-192	37	55	41	1	0	550	628
AES-256							732

Table 2.2: Implementation results for the AES key schedule. Most state machine encoding and control logic has been incorporated into the BRAM to save on logic resources.

and 41 flip-flops. All key schedule related data is presented in Table 2.2 supporting key sizes of 128, 192 and 256 bits.

2.7 Conclusions and Future Work

In this chapter we built a novel design to perform AES operations at a high throughput using on-chip RAM and DSP blocks with minimal use of traditional user logic such as flip-flops and look-up tables.

The AES block cipher can be used with many different modes of operations. Due to the pipelined architecture, our design is suited best to operate in CTR mode running an incrementing counter that is encrypted and XORed with the plaintext. With direct support of parallelism in CTR mode and careful design of the counter[2], we can achieve the high performance reported in Section 2.6 also for more complex modes of operation. Furthermore, the Electronic Codebook Mode (ECB) as well as authenticated encryption by CMAC can be naturally supported by running parallel streams with the presented design. Please confer to [Dri09] for more details about using the presented architectures with different modes of operation.

Further investigation (that is not in the scope of this thesis) should target the evaluation of our designs against side channel attacks, such as power analysis. Since we use the same basic construct for all designs, it is interesting to evaluate how pipelining and unrolling affect power signatures (cf. [SÖP04]).

[2] A wide 128 bit integer increment counter can introduce long signal propagation paths due to carry propagation degrading the overall system performance. Here, the use of three cascaded DSP slices for multi-precision addition or a Linear Feedback Shift Registers (LFSR) can be used to avoid any performance penalty.

Chapter 3

Optimal ECC Architectures for High-Performance FPGAs

Elliptic Curve Cryptosystems (ECC) have gained increasing acceptance in practice due to their significantly smaller bit size of the operands compared to other public-key cryptosystems, particularly when high-performance is required. This chapter presents a new architecture for an FPGA-based high-performance ECC implementation over prime fields. Our architecture makes intensive use of the embedded DSP function blocks in Virtex-4 FPGAs to accelerate the low-level modular arithmetic required in ECC. Based on this approach we describe the implementation of standard ECC over prime fields compliant to FIPS 186-2/3.

Contents of this Chapter

3.1 Motivation

Asymmetric cryptographic algorithms are known to be extremely arithmetic intensive since their security assumptions rely on computational problems which are considered hard in combination with parameters of significant bit sizes.

Neal Koblitz and Victor Miller proposed independently in 1985 [Mil86, Kob87] the use of Elliptic Curve Cryptography providing similar security compared to classical cryptosystems but using smaller keys. This benefit allows for greater efficiency when using ECC (160–256 bit) compared to RSA or discrete logarithm schemes over finite fields (1024–4096 bit) while providing an equivalent level of security [LV01]. Due to this, ECC has become the most promising

candidate for many new applications, especially in the embedded domain, which is also reflected by several standards by IEEE, ANSI and SECG [P1300, ANS05, Cer00a, Cer00b].

In addition to many new "lightweight" applications (e.g., digital signatures on RFID-like devices), there are also many new applications which call for high-performance asymmetric primitives. Even though very fast public-key algorithms can be provided for PC and server applications by accelerator cards equipped with ASICs, providing very high speed solutions in embedded devices is still a major challenge. Somewhat surprisingly, there appears to be extremely few, if any, commercially available ASICs or chip sets that provide high speed ECC and which are readily available for integration in general embedded systems. A potential alternative is provided by FPGAs. However, despite a wealth of research regarding high-speed FPGA (and high-speed software) implementation of ECC since the mid 1990s, providing truly high-performance ECC (i.e., to reach less than 100μs per point multiplication) on readily available platforms remains an open challenge. This holds especially for ECC over prime fields, which are often preferred over binary fields due to standards in Europe and the US, and a somewhat clearer patent situation.

In this work, we propose a novel hardware architecture based on reconfigurable FPGAs supporting ECC cryptography over prime fields $\mathbb{F}_p$ offering the highest single-chip performance reported in literature up to now. Usually, known ECC implementations for reconfigurable logic implement the computationally expensive low-level arithmetic in configurable logic elements, allowing for greatest flexibility but offering only moderate performance. Some implementations have attempted to address this problem by using dedicated arithmetic hardware in the reconfigurable device for specific parts of the computations, like built-in 18×18 multipliers [MMM04]. But other components of the circuitry for field addition, subtraction and inversion have been still implemented in the FPGA's fabric which usually leads to a decrease in performance.
The central idea of this contribution is to relocate the arithmetic intensive operations of ECC over prime fields *entirely* in dedicated arithmetic function blocks on the FPGA actually targeting DSP filter applications. As introduced in Section 2.4, these DSP accelerating functions are built-in components in the static logic of modern FPGA devices capable to perform integer multiplication, addition and subtraction as well as a multiply-accumulate operation.

3.2 Previous Work

We briefly summarize previously published results of relevance to this chapter. There is a wealth of publication addressing ECC hardware architectures, and a good overview can be found in [dDQ07]. In the case of high speed architectures for ECC, most implementation primarily address elliptic curves over binary fields $GF(2^m)$ since the arithmetic is more hardware-friendly due to carry-free computations [OP00, EGCS03]. Our work, however, focuses on the prime field $\mathbb{F}_p$. First implementations for ECC over prime fields $\mathbb{F}_p$ have been proposed by [OP01, ST03a] demonstrating ECC processors built completely in reconfigurable logic. The contribution by [MMM04] proposes a high-speed ECC crypto core for arbitrary moduli with up to 256 bit length designed on a large number of built-in multiplier blocks of FPGA devices pro-

viding a significant speedup for modular multiplications. However, other field operations have been implemented in the FPGA fabric, resulting in a very large design (15,755 slices and 256 multiplier blocks) on a large Xilinx XC2VP125 device. The architecture presented in [DMKP04] was designed to achieve a better trade-off between performance and resource consumption. According to the contribution, an area consumption of only 1,854 slices and a maximum clock speed of 40 MHz can be achieved on a Xilinx Virtex-2 XC2V2000 FPGA for a parameter bit length of 160 bit.

Our approach to implementing an FPGA-based ECC engines was to shift *all* field operations into the integrated DSP building blocks available on modern FPGAs. This strategy frees most configurable logic elements on the FPGA for other applications and requires less power compared to a conventional design. In addition to that, this architecture offers the fastest performance for ECC computations over prime fields with up to 256 bit security in reconfigurable logic.

3.3 Mathematical Background

In this section, we will briefly introduce to the mathematical background relevant for this work. We will start with a short review of the Elliptic Curve Cryptosystems (ECC). Please note that only ECC over prime fields $\mathbb{F}_p$ will be subject of this work since binary extensions fields $GF(2^m)$ require binary arithmetic which is not (yet) natively supported by DSP blocks.

3.3.1 Elliptic Curve Cryptography

Let p be a prime with $p > 3$ and $\mathbb{F}_p = \mathbb{F}_p$ the Galois Field over p. Given the Weierstrass equation of an elliptic curve

$$\mathcal{E} : y^2 = x^3 + ax + b,$$

with $a, b \in \mathbb{F}_p$ and $4a^3 + 27b^2 \neq 0$, points $\mathcal{P}_i \in \mathcal{E}$, we can compute tuples (x, y) also considered as points on this elliptic curve $\mathcal{E}$. Based on a group of points defined over this curve, ECC arithmetic defines the addition $\mathcal{R} = \mathcal{P} + \mathcal{Q}$ of two points $\mathcal{P}, \mathcal{Q}$ using the *tangent-and-chord* rule as the primary group operation. This group operation distinguishes the case for $\mathcal{P} = \mathcal{Q}$ (*point doubling*) and $\mathcal{P} \neq \mathcal{Q}$ (*point addition*). Furthermore, formulas for these operations vary for affine and projective coordinate representations. Since affine coordinates require the availability of fast modular inversion, we will focus on projective point representation to avoid the implementation of a costly inversion circuit. Given two points $\mathcal{P}_1, \mathcal{P}_2$ with $\mathcal{P}_i = (X_i, Y_i, Z_i)$ and $\mathcal{P}_1 \neq \mathcal{P}_2$, the sum $\mathcal{P}_3 = \mathcal{P}_1 + \mathcal{P}_2$ is defined by

$$\begin{aligned} A &= Y_2Z_1 - Y_1Z_2 & B &= A^2Z_1Z_2 - C^3 - 2C^2X_1Z_2 & C &= X_2Z_1 - X_1Z_2 \\ X_3 &= BC & Y_3 &= A(C^2X_1Z_2 - B) - C^3Y_1Z_2 & Z_3 &= C^3Z_1Z_2, \end{aligned} \tag{3.1}$$

where A, B, C are auxiliary variables and $\mathcal{P}_3 = (X_3, Y_3, Z_3)$ is the resulting point in projective coordinates. Similarly, for $\mathcal{P}_1 = \mathcal{P}_2$ the point doubling $\mathcal{P}_3 = 2\mathcal{P}_1$ is defined by

$$\begin{aligned} A &= aZ^2 + 3X^2 & B &= YZ \quad C = XYB & D &= A^2 - 8C \\ X_3 &= 2BD & Y_3 &= A(4C - D) - 8B^2Y^2 & Z_3 &= 8B^3. \end{aligned} \tag{3.2}$$

Most ECC-based cryptosystems rely on the Elliptic Curve Discrete Logarithm Problem (ECDLP) and thus employ the technique of point multiplication $k \cdot \mathcal{P}$ as cryptographic primitive, i.e., a k times repeated point addition of a base point $\mathcal{P}$. More precisely, the ECDLP (cf. Chapter 6) is the fundamental cryptographic problem used in protocols and crypto schemes like the Elliptic Curve Diffie-Hellman key exchange [DH76], the ElGamal encryption scheme [Elg85] and the Elliptic Curve Digital Signature Algorithm (ECDSA) [ANS05].

3.3.2 Standardized General Mersenne Primes

The arithmetic for ECC point multiplication is based on modular computations over a prime field $\mathbb{F}_p$. These computations always include a subsequent step to reduce the result to the domain of the underlying field. Since the reduction is very costly for general primes due to the demand for a multi-precision division, special primes have been proposed by Solinas [Sol99] which have been finally standardized in [Nat99]. These primes provide efficient reduction algorithms based on a sequence of multi-precision addition and subtractions only and eliminate the need for the costly division. Special primes P-l with bit sizes $l = \{192, 224, 256, 384, 521\}$ are part of the standard. But we believe that the primes P-224 and P-256 are the most relevant bit sizes for future implementations of the next decades.

Algorithm 3.1 NIST Reduction with P-224 $= 2^{224} - 2^{96} + 1$

Input: Double-sized integer $c = (c_{13}, \ldots, c_2, c_1, c_0)$ in base 2^{32} and $0 \leq c <$ P-224^2

Output: Single-sized integer c mod P-224.

1: Concatenate c_i to following 224-bit integers z_j:

$$\begin{aligned} z_1 &= (c_6, c_5, c_4, c_3, c_2, c_1, c_0),\ z_2 = (c_{10}, c_9, c_8, c_7, 0, 0, 0), \\ z_3 &= (0, c_{13}, c_{12}, c_{11}, 0, 0, 0),\ z_4 = (0, 0, 0, 0, c_{13}, c_{12}, c_{11}), \\ z_5 &= (c_{13}, c_{12}, c_{11}, c_{10}, c_9, c_8, c_7) \end{aligned}$$

2: Compute $c = (z_1 + z_2 + z_3 - z_4 - z_5 \bmod \text{P-224})$

According to Algorithm 3.1 the modular reduction for P-224 can be performed with two 224-bit subtractions and additions. However, these four consecutive operations can lead to a potential over- and underflow in step 2. With $Z = z_1 + z_2 + z_3 - z_4 - z_5$, we can determine the bounds $-2p < Z < 3p$ reducing the number of final correction steps to two additions or subtractions to compute the correctly bounded c mod P-224.

Algorithm 3.2 NIST Reduction with P-256 $= 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$

Input: Double-sized integer $c = (c_{15}, \ldots, c_2, c_1, c_0)$ in base 2^{32} and $0 \leq c <$ P-256^2

Output: Single-sized integer c mod P-256.

1: Concatenate c_i to following 256-bit integers z_j:

$$\begin{aligned}
z_1 &= (c_7, c_6, c_5, c_4, c_3, c_2, c_1, c_0),\ z_2 = (c_{15}, c_{14}, c_{13}, c_{12}, c_{11}, 0, 0, 0),\\
z_3 &= (0, c_{15}, c_{14}, c_{13}, c_{12}, 0, 0, 0),\ z_4 = (c_{15}, c_{14}, 0, 0, 0, c_{10}, c_9, c_8),\\
z_5 &= (c_8, c_{13}, c_{15}, c_{14}, c_{13}, c_{11}, c_{10}, c_9),\ z_6 = (c_{10}, c_8, 0, 0, 0, c_{13}, c_{12}, c_{11}),\\
z_7 &= (c_{11}, c_9, 0, 0, c_{15}, c_{14}, c_{13}, c_{12}),\ z_8 = (c_{12}, 0, c_{10}, c_9, c_8, c_{15}, c_{14}, c_{13}),\\
z_9 &= (c_{13}, 0, c_{11}, c_{10}, c_9, 0, c_{15}, c_{14})
\end{aligned}$$

2: Compute $c = (z_1 + 2z_2 + 2z_3 + z_4 + z_5 - z_6 - z_7 - z_8 - z_9 \bmod \text{P-256})$

Algorithm 3.2 presents the modular reduction for P-256 requiring two doublings, four 256-bit subtractions and four 256-bit additions. Based on the computation $Z = z_1 + 2z_2 + 2z_3 + z_4 + z_5 - z_6 - z_7 - z_8 - z_9$, the range of the result to be corrected is $-4p < Z < 5p$.

3.4 An Efficient ECC Architecture Using DSP Cores

In this section we demonstrate how to implement ECC over NIST primes P-224 and P-256 using available DSP blocks of Xilinx Virtex-4 FPGAs.

3.4.1 ECC Engine Design Criteria

When using DSP blocks as presented in Section 2.4 to develop a high-speed ECC design, there are several criteria which should be obeyed to exploit their full performance for complex arithmetic. Note that the following aspects have been designed to target the requirements of Xilinx Virtex-4 FPGAs:

(1) *Build DSP cascades:* Neighboring DSP blocks can be cascaded to widen or extend their atomic operand width (e.g., from 18 bit to 256 bit).

(2) *Use DSP routing paths:* DSPs have been provided with inner routing paths connecting two adjacent blocks. It is advantageous in terms of performance to use these paths as frequently as possible instead of using FPGA's general switching matrix for connecting logic blocks.

(3) *Consider DSP columns:* Within a Xilinx FPGA, DSPs are aligned in columns, i.e., routing paths between DSPs within the same column are efficient while a switch in columns can lead to degraded performance. Hence, DSP cascades should not exceed the column width (typically 32/48/64 DSPs per column).

(4) *Use DSP pipeline registers:* DSP blocks feature pipeline stages which should be used to achieve the maximum clock frequency supported by the device (up to 500 MHz).

(5) *Use different clock domains:* Optimally, DSP blocks can be operated at maximum device frequency. This is not necessarily true for the remainder of the design so that separate clock domains should be introduced (e.g. by halving the clock frequency for control signals) to address the critical paths in each domain individually.

3.4.2 Arithmetic Units

According to the EC arithmetic introduced in Section 3.3.1, an ECC engine over $\mathbb{F}_p$ based on projective coordinates requires functionality for modular addition, subtraction and multiplication. Since modular addition and subtraction is very similar, both operation are combined. In the following description we will assume a Virtex-4 FPGA as reference device and corresponding DSP block arithmetic with word sizes $l_A = 32$ and $l_M = 16$ for unsigned addition and multiplication, respectively. Recall that native support by the DSP blocks on a Virtex-4 device is available for up to 48-bit signed addition and 18-bit signed multiplication.

Modular Addition/Subtraction

Let $A, B \in GF(P)$ be two multi-precision operands with lengths $|A|, |B| \leq l$ and $l = \lfloor \log_2 P \rfloor + 1$. Modular addition $C = A + B \bmod P$ and subtraction $C = A - B \bmod P$ can be efficiently computed according to Algorithm 3.3:

Algorithm 3.3 Modular addition and subtraction

Input: A, B, P with $0 \leq A, B < P$;
Flag $f \in \{0, 1\}$ denotes a subtraction when $f = 1$ and addition otherwise

Output: $C = A \pm B \bmod P$

1: $(C_{\mathsf{IN0}}, S_0) = A + (-1)^f B$;
2: $(C_{\mathsf{IN1}}, S_1) = S_0 + (-1)^{1-f} P$;
3: Return $S_{|f - C_{\mathsf{IN}f}|}$;

For using DSP blocks, we need to divide the l-bits operands into multiple words each having a maximum size of l_A bits due to the limited width of the DSP input port. Thus, all inputs A, B and P to the DSP blocks can be represented in the form $X = \sum_{i=0}^{n_A - 1} x_i \cdot 2^{i \cdot l_A}$, where $n_A = \lceil l / l_A \rceil$ denotes the number of words of an operand. According to Algorithm 3.3, we employ two cascaded DSP blocks, one for computing $s_{(0,i)} = a_i \pm (b_i + C_{\mathsf{IN0}})$ and a second for $s_{(1,i)} = s_{(0,i)} \mp (p_i + C_{\mathsf{IN1}})$. The resulting values $s_{(0,i)}$ and $s_{(1,i)}$ each of size $|s_{(j,i)}| \leq l_A + 1$ are temporarily stored and recombined to S_0 and S_1 using shift registers (SR). Finally, a 2-to-1 l-bit output multiplexer selects the appropriate value $C = S_i$. Figure 3.1 presents a schematic overview of a combined modular addition and subtraction based on two DSP blocks. Note that DSP blocks on Virtex-4 FPGAs provide a dedicated carry input c_{IN} but *no* carry output c_{OUT}. Particularly, this fact requires extra logic to compensate for duplicate carry propagation to the

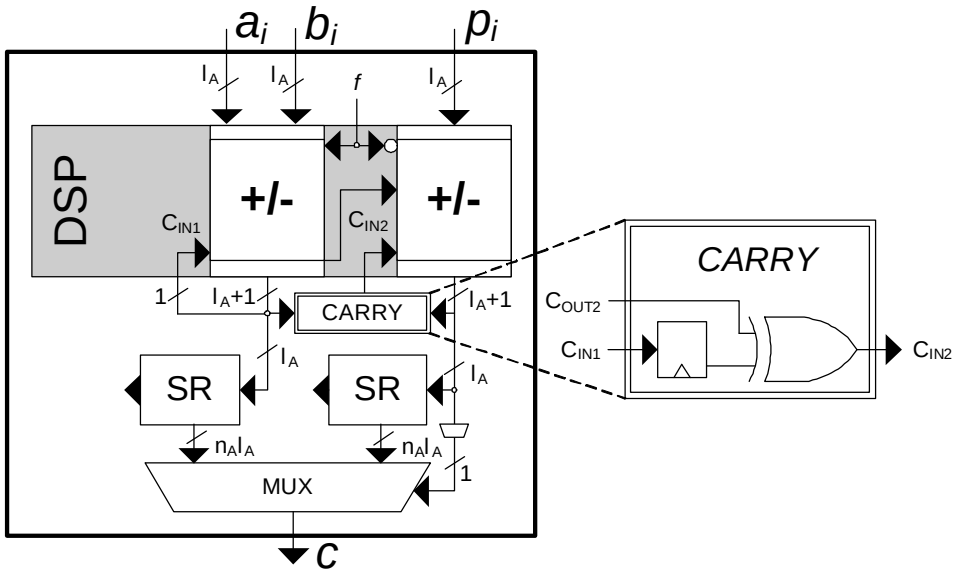


Figure 3.1: Modular addition/subtraction based on DSP-blocks.

second DSP which is due to the fixed cascaded routing path between the DSP blocks. In this architecture, each carry is considered twice, namely in $s_{0,i+1}$ and $s_{1,i}$ what needs to be corrected. This special carry treatment requires a wait cycle to be introduced so that one l_A-bit word can be processed each two clock cycles. However, this is no restriction for our architecture since we design for *parallel* addition and multiplication so that the (shorter) runtime of an addition is completely hidden in the duration of a concurrent multiplication operation.

Modular Multiplication

The most straightforward multiplication algorithm to implement the multiplication with subsequent NIST prime reduction (cf. Section 3.3.2) is the schoolbook multiplication method with a time complexity of $\mathcal{O}(n^2)$ for n-bit inputs. Other methods, like the Karatsuba algorithm [KO63], trade multiplications for additions using a divide-and-conquer approach. However Karatsuba computing the product $C = A \times B$ for $A = a_1 2^n + a_0$ and $B = b_1 2^n + b_0$ requires to store the intermediate results for $a_1 a_0$ and $b_1 b_2$ for later reuse in the algorithm. Although this is certainly possible, this requires a much more complex data and memory handling and cannot be solely done within DSP blocks. Since many parts of the Karatsuba multiplier would require generic logic of CLBs, we are likely to lose the gain of performance of the fast arithmetic in DSP blocks. We thus use a variant of the schoolbook multiplication, known as Comba multiplication [Com90] which combines carry handling and reduces write accesses to the memory. These optimizations result in improved performance with respect to the original pen-and-paper method. Let $A, B \in GF(P)$ be two multi-precision integers with bit length $l \leq \lfloor \log_2 P \rfloor + 1$. According to the limited input size l_M of DSP blocks, we split now the values A, B in $n_M = \lceil l/l_M \rceil$ words

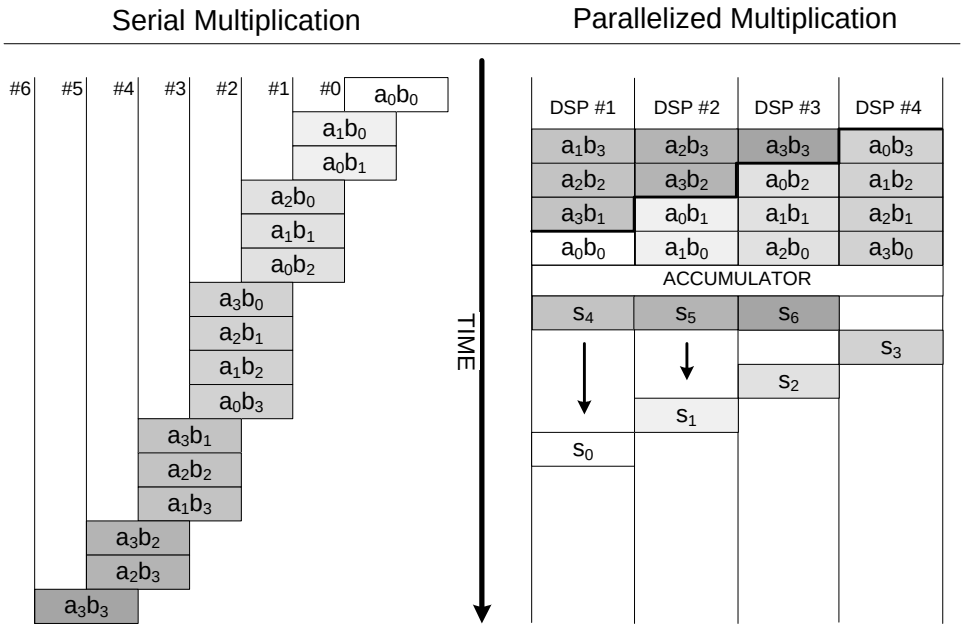


Figure 3.2: Parallelizing Comba's multiplication method for efficient DSP-based computation.

represented as $X = \sum_{i=0}^{n_M-1} x_i \cdot 2^{il_M}$. Straightforward multiplication computes $C = A \cdot B$ based on accumulation of $(n_M)^2$ products $C = \sum_{i=0}^{2n_M} 2^{i \cdot n_M} \sum_{j=0}^{i} a_j b_{i-j}$ providing a result C of size $|C| \leq 2n_M$. For parallel execution on n_M DSP units, we compacted the order of inner product computations used for Comba's algorithm as shown in Figure 3.2. All n_M DSP blocks operate in a loadable *Multiply-and-Accumulate* mode (MACC) so that intermediate results remain in the corresponding DSP block until an inner product $s_i = \sum_{j=0}^{i} a_j b_{i-j}$ is fully computed. Note that s_i returned from the n_M DSP blocks are not aligned and can vary in size up to $|s_i| \leq 2l_M + \log_2(n_M) = l_{ACC} = 36$ bits. Thus, all s_i need to be converted to non-redundant representation to finally form the final product of words c_i with maximum size $2l_M$ each. Hence, we feed all values into a subsequent accumulator to combine each s_i with the corresponding bits of s_{i-1} and s_{i+1}. Considering the special input constraints, timing conventions and carry transitions of DSP blocks, we developed Algorithm 3.4 to address the accumulation of inner products based on two DSP blocks performing l_{ACC}-bit additions.

Figure 3.3 gives a schematic overview of the multiplication circuit returning the full-size product C. This result has to be reduced using the fast NIST prime reduction scheme discussed in the next section.

Modular Reduction

At this point we will discuss the subsequent modular reduction of the $2n_M$-bit multiplication result C using the NIST reduction scheme. All fast NIST reduction algorithms rely on a reduction step (1) defined as a series multi-precision additions and subtractions followed by a

Algorithm 3.4 Accumulation of partial product c_i

Input: Partial products s_i with $|s_i| \leq l_{ACC}$ bits for $i = 0 \ldots 2n_M - 1$ and $l_{ACC} = 2l_M + \log_2(n_M)$
Output: Product $C = (c_{2n_M}, \ldots, c_0)$ with $|C| \leq 2l$ bits

1: $s_{(-1)} \rightarrow 0$; $c_{(-1)} \rightarrow 0$
2: **for** $i = 0$ to $2n_M - 2$ by 2 **do**
3: $\quad d_i \rightarrow \text{ADD}(s_{i-1}[l_{ACC} - 1 \ldots l_M], s_i[l_{ACC} \ldots 0])$
4: $\quad c_i \rightarrow \text{ADD}(d_i[l_{ACC} \ldots l_M], (s_{i+1}[l_M \ldots 0] | c_{i-1}[3l_M \ldots 2l_M]))$
5: **end for**
6: **return** $c = (c_{2n_M - 1}, \ldots, c_0)$

correction step (2) to achieve a final value in the interval $[0, \ldots, P-1]$ (cf. Algorithms 3.1 and 3.2). To implement (1), we decided to use one DSP-block for each individual addition or subtraction, e.g., for the P-256 reduction we reserved a cascade of 8 DSP blocks. Each DSP performs one addition or subtraction and stores the result in a register whose output is taken as input to the neighboring block (data pipeline).

For the correction step (2), we need to determine *in advance* the possible overflow or underflow of the result returned by (1) to avoid wait or idle cycles in the pipeline. Hence, we introduced a Look-Ahead Logic (LAL) consisting of a separate DSP block which exclusively computes the expected overflow or underflow. Then, the output of the LAL is used to select a corresponding reduction value which are stored as multiple $\{0, \ldots, 5P\}$ in a ROM table. The ROM values are added or subtracted to the result of (1) by a sequence of two DSP blocks ensuring that the final result is always in $\{0, \ldots, P-1\}$. Figure 3.4 depicts the general structure of the reduction circuit which is applicable for both primes P-224 and P-256.

3.4.3 ECC Core Architecture

With the basic field operations for $l - bit$ computations at hand supporting NIST primes P-224 and P-256, we have combined a modular multiplier and a modular subtraction/addition component with dual-port RAM modules (BRAM) and a state machine to build an ECC core. We have implemented an asymmetric data path supporting two different operand lengths: the first operand provides full l-bit of data whereas the second operand is limited to 32-bit words so that several words need to be transferred serially to generate the full l-bit input. This approach allows for direct memory accesses of our *serial-to-parallel* multiplier architecture. Note further that we introduced different clock domains for the core arithmetic based on the DSP blocks and the state machines for upper layers (running at half clock frequency only). An overview of the entire ECC core is shown in Figure 3.5. We implemented ECC group operations based on projective Chudnowsky coordinates[1] since the implementation should support to compute

[1] ECC operations for elliptic curves in Weierstrass form based on mixed affine-Jacobian coordinates are more efficient but more complex in hardware. This is due to the required conversion of precomputed points from Jacobian to affine coordinates which is necessary when computing $k \cdot \mathcal{P} + r \cdot \mathcal{Q}$ for a ECDSA signature verification. In that case modular inversion is required.

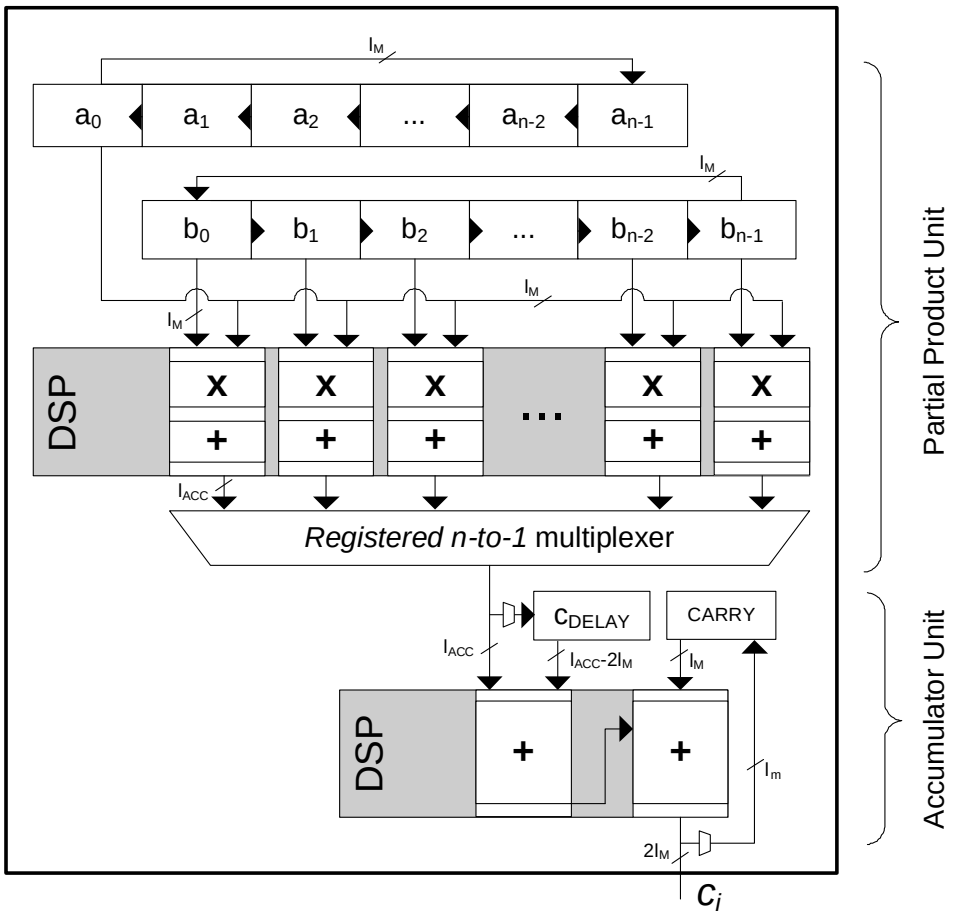


Figure 3.3: An l-bit multiplication circuit employing a cascade of parallelly operating DSP blocks.

a point multiplication $k \cdot \mathcal{P}$ *as well as* a corresponding linear combination $k \cdot \mathcal{P} + r \cdot \mathcal{Q}$ based on a fixed base point $\mathcal{P} \in \mathcal{E}$ where $k, r \in \{1, \ldots, ord(\mathcal{P}) - 1\}$ and $\mathcal{Q} \in \langle \mathcal{P} \rangle$. Both operations can be considered as basic ECC primitives, e.g., used for ECDSA signature generation and verification [ANS05]. The computation of $k \cdot \mathcal{P} + r \cdot \mathcal{Q}$ can make use of *Shamir's trick* to efficiently compute several point products simultaneously [Elg85]. For this first implementation of the point multiplication and the sake of simplicity, we used a standard double-and-add (binary method) algorithm [HMV04], but more efficient windowing methods [ACD+05] can also be implemented without significantly increasing the resource consumption.

3.4.4 ECC Core Parallelism

Due the intensive use of DSP blocks to implement the core functionality of ECC, the resulting implementation requires only few reconfigurable logic elements on the FPGA. This allows for efficient multiple-core implementations on a single FPGA improving the overall system throughput

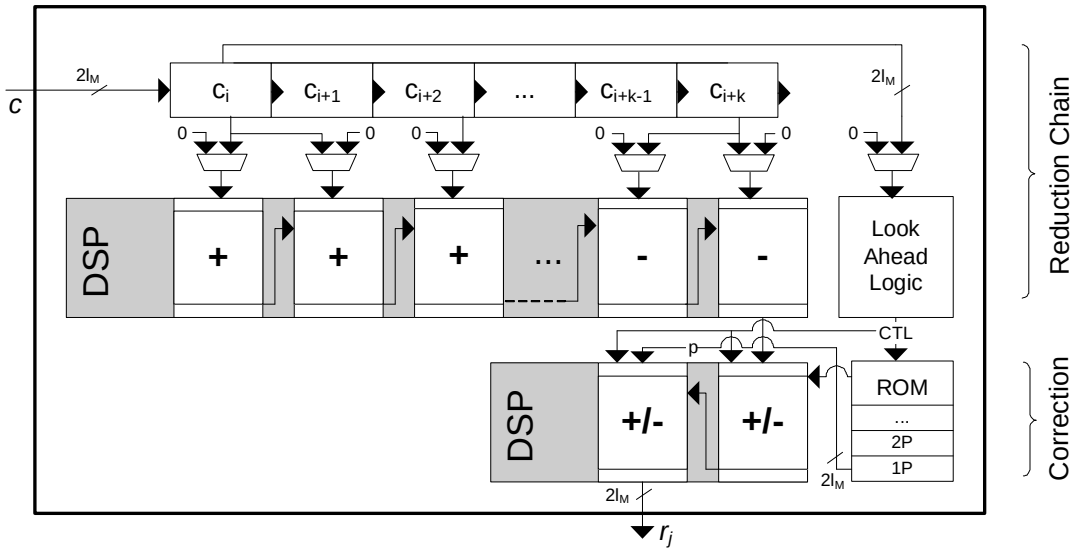


Figure 3.4: Modular reduction for NIST-P-224 and P-256 using DSP blocks.

by a linear factor n dependent on the number of cores. Note that most other high-performance implementations occupy the full FPGA due to their immense resource consumption so that these cannot easily be instantiated several times.
Based on our synthesis results, the limiting factor of our architecture is the number of available DSP blocks of a specific FPGA device (cf. Section 3.5).

3.5 Implementation

The proposed architecture has been synthesized and implemented for the smallest available Xilinx Virtex-4 device (XC4VFX12-12SF363) and the corresponding results are presented in Subsection 3.5.1. This FPGA offers 5,472 slices (12,288 4-input LUTs and flip-flops) of reconfigurable logic, 32 DSP blocks and can be operated at a maximum clock frequency of 500 MHz. Furthermore, to demonstrate how many ECC computations can be performed using ECC core parallelism, we take a second device, the large Xilinx Virtex-4 XC4VSX55-12FF1148 providing the maximum number of 512 DSP blocks and 24,576 slices (49,152 4-input LUTs and flip-flops) as a reference for a multi-core architecture.

3.5.1 Implementation Results

Based on the Post-Place and Route (PAR) results using Xilinx ISE 9.1 we can present the following performance and area details for ECC cores for primes P-224 and P-256 on the small XC4VFX12 device as shown in Table 3.1. Note that the implementation for P-224 is not yet fully verified in functionality since we focused on the development of the core for P-256 that is already available for use in real-world products.

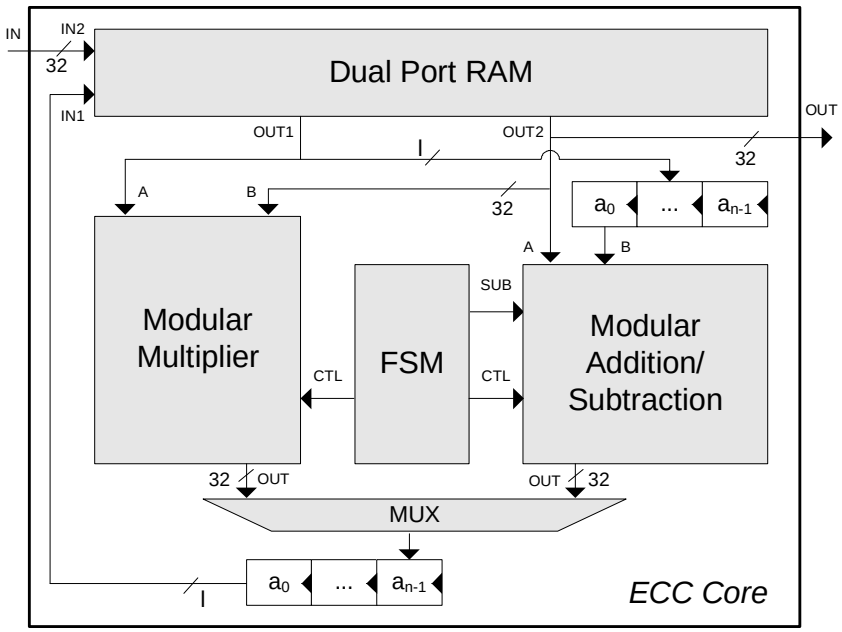


Figure 3.5: Schematic overview of a single ECC core.

3.5.2 Throughput of a Single ECC Core

Given an ECC core with a separate adder/subtracter and multiplier unit, we can perform a field multiplication and field addition simultaneously. By optimizing the execution order of the basic field operations, it is possible to perform all additions/subtraction required for the ECC group operation in parallel to a multiplication. Based on the runtimes of a single field multiplication, we can determine the number of required clock cycles for the operations $k \cdot \mathcal{P}$ and $k \cdot \mathcal{P} + r \cdot \mathcal{Q}$ using the implemented Double-and-Add algorithm. Moreover, we also give

Aspect	**Core**P-224	**Core** P-256
Slices occupied	1,580 (29%)	1,715 (31%)
4-input LUTs	1,825	2,589
Flip-flops	1,892	2,028
DSP blocks	26	32
BRAMs	11	11
Frequency (DSP)/Max. delay	487 MHz/2.050 ns	490 MHz/2.040 ns
Frequency (control)	243.5 MHz	245 MHz

Table 3.1: Resource requirements of a single ECC core on a Virtex-4 FX 12 after PAR. Note the different clock domains for arithmetic (DSP) and control logic.

Aspect	**Core** P-224	**Core** P-256
Cycles per MUL in $\mathbb{F}_p$	58	70
Cycles per ADD/SUB in $\mathbb{F}_p$	16	18
Cycles per ECC Addition (Chudnovsky)	812	980
Cycles per ECC Doubling (Chudnovsky)	580	700
Cycles $k \cdot \mathcal{P}$ (Double&Add)	219,878	303,450
Cycles $k \cdot \mathcal{P}$ (Window)	178,000*	243,000*
Cycles $k \cdot \mathcal{P} + r \cdot \mathcal{Q}$ (Double&Add)	265,959	366,905
Cycles $k \cdot \mathcal{P} + r \cdot \mathcal{Q}$ (Window)	194,000*	264,000*
Time and OP/s for $k \cdot \mathcal{P}$ (Double&Add)	452 μs/2214	620 μs/1614
Time and OP/s for $k \cdot \mathcal{P}$ (Window)	365 μs*/2740*	495 μs*/2020*
Time and OP/s for $k \cdot \mathcal{P} + r \cdot \mathcal{Q}$ (Double&Add)	546 μs/1831	749 μs/1335
Time and OP/s for $k \cdot \mathcal{P} + r \cdot \mathcal{Q}$ (Window)	398 μs*/2510*	540 μs*/1850*

Table 3.2: Performance of ECC operations based on a single ECC core using projective Chudnowsky coordinates on a Virtex-4 XC4VFX12 (Figures with asterisk are estimates).

estimates concerning their performance when using a window-based method [ACD+05] based on a window size $w = 4$.

Note that the specified timing considers signal propagation after complete PAR excluding the timing constraints from I/O pins ("TIG" constraint) since no underlying data communication layer was implemented. Hence, when being combined with an I/O protocol of a real-world application, the clock frequency can be slightly lower than specified in Table 3.1 and Table 3.3.

3.5.3 Multi-Core Architecture

Since a single ECC core has obviously moderate resource requirements, it is possible to place multiple instances of the core on a larger FPGA. On a *single* XC4VSX55 device, we can implement, depending on the underlying prime field, between 16 to 18 ECC cores in parallel (cf. Table 3.3). Due the small amount of LUTs and flip-flops required for a single core, the number of available DSP blocks (and routing resources) on the FPGA is here the limiting factor.

3.5.4 Comparison

Based on our architecture, we can estimate a throughput of more than 37,000 point multiplications on the standardized elliptic curve P-224 per second which exceeds the throughput of all *single-chip* hardware implementation known to the authors by far. A detailed comparison with other implementations is presented in Table 3.5.4.

At this point we like to point out that the field of highly efficient *prime field* arithmetic is believed to be predominated by implementations on general purpose microprocessors rather than

Aspect	**Core** P-224	**Core** P-256
Number of Cores	18	16
Slices occupied	24,452 (99%)	24,574 (99%)
4-input LUTs	32,688	34,896
Flip-flops	34,166	32,430
DSP blocks	468	512
BRAMs	198	176
Frequency (DSP)/Max. delay	372 MHz/2.685 ns	375 MHz/2.665 ns
Frequency (control)	186 MHz	187.5 MHz
OP/s $k \cdot P$ (Double&Add)	30,438	19,760
OP/s $k \cdot P$ (Window)	37,700*	24,700*
OP/s $k \cdot P + r \cdot Q$ (Double&Add)	25,164	16,352
OP/s $k \cdot P + r \cdot Q$ (Window)	34,500*	22,700*

Table 3.3: Results of a multi-core architecture on a Virtex-4 XC4VSX55 device for ECC over prime fields P-224 and P-256 (Figures with an asterisk are estimates).

on FPGAs. Hence, we will also compare our hardware implementation against the performance of software solutions on recent microprocessors. Since most performance figures for software implementations are given in cycles rather than absolute times, we assumed for comparing throughputs that uninterrupted, repeated computations can be performed simultaneously on *all* available cores of a modern microprocessor with no further cycles spent, e.g., on scheduling or other administrative tasks. Note that this is indeed a very optimistic assumption possibly overrating the performance of software implementations with respect to actual applications.
For example, a point multiplication using the highly efficient software implementation by Bernstein based on floating point arithmetic for ECC over P-224 requires 839.000 cycles on an (outdated) Intel Pentium 4 [Ber01] at 1.4GHz. According to our assumption for cycle count interpretation, this correlates to 1670 point multiplication per second.
We also compare our design to more recent results, e.g., obtained from ECRYPT's eBATS project. According to the report from March 2007 [ECR07], an Intel Core2 Duo running at 2.13 GHz is able to generate 1868 and 1494 ECDSA signatures based on the OpenSSL implementation for P-224 and P-256, respectively. Taking latest Intel Core2 Quad microprocessors into account, these performance figures might even double. We also compare our work to the very fast software implementation by [GT07] using an Intel Core2 Duo system at 2.66 GHz. However, in this contribution the special Montgomery and non-standard curve over $\mathbb{F}_{2^{255}-19}$ is used instead of a standardized NIST prime. Despite of that, for the design based on this curve the authors report the impressive throughput of 6700 point multiplications per second.
For a fair comparison with software solutions it should be considered that a single Virtex-4 SX 55 costs about US$ 1,170[2]. Recent microprocessors like the Intel Core2 Duo, however, are

[2]Market price for a single device in Jan 2009.

Scheme	Device	Design	Logic	Clock	Time
This work	XC4VFX12-12	ECC NIST P-224	1580 LS/26 DSP	487 MHz	365 μs
	XC4VFX12-12	ECC NIST P-256	1715 LS/32 DSP	490 MHz	495 μs
	XC4VSX55-12	ECC NIST P-224	24452 LS/468 DSP	372 MHz	26.5 μs
	XC4VSX55-12	ECC NIST P-256	24574 LS/512 DSP	375 MHz	40.5 μs
[OP01]	XCV1000E	ECC NIST P-192	5708 LS	40 MHz	3 ms
[MMM04]	XC2VP125-7	ECC P-256 all	15755 LS/256 MUL	39.5 MHz	3.84 ms
[ST03a]	0.13 μm CMOS	ECC P-160 all	117500 GE	137 MHz	1.21 ms
[Ber01]	Intel Pentium4	ECC NIST P-224	32 bit μP	1.4 GHz	599 μs
[ECR07]	Intel Core2 Duo	ECC NIST P-256	64 bit μP	2.13 GHz	669[a] μs
[GT07]	Intel Core2 Duo	ECC GF($2^{255} - 19$)	64 bit μP	2.66 GHz	145 μs
[BP01]	XC40250XV	RSA-1024	6826 CLB	45.2 MHz	3.1 ms
[Suz07]	XC4VFX12-10	RSA-1024	3937 LS/17 DSP	400 MHz	1.71 ms
[SFCK04]	0.5 μm CMOS	RSA-1024	28,000 GE	64 MHz	46 ms

[a]Note that this figure reflects a full ECDSA signature generation rather than a point multiplication.

Table 3.4: Selected high-performance implementations of public-key cryptosystems.

available at only about a quarter of that price. With this in mind, we might not be able to beat all software implementation in terms of the cost-performance ratio, but we still like to point out that our FPGA-based design - as the fastest reported hardware implementation so far - definitely closes the performance gap between software and hardware implementations for ECC over prime fields. Furthermore, we like to emphasize again that all software related performance figures are based on very optimistic assumptions.

3.6 Conclusions

In this chapter, we presented novel ECC implementations in reconfigurable hardware for fields over the NIST primes P-224 and P-256. Due to the exhaustive utilization of DSP blocks, which are contained as hardcores in modern FPGA devices, we are able to run the critical components computing low-level integer arithmetic operations nearly at maximum device frequency. Furthermore, considering a multi-core architecture on a Virtex-4 XC4VSX55 FPGA, we can achieve a throughput of more than 24,000 and 37,000 point multiplications per second for P-256 and P-224, respectively, what significantly exceeds the performance of all other hardware implementation known to the authors and comes close to the cost-performance ratio provided by the fastest available software implementations in the open literature.

Chapter 4

High-Performance Asymmetric Cryptography with Graphics Cards

Modern Graphics Processing Units (GPU) have reached a dimension with respect to performance and gate count exceeding conventional Central Processing Units (CPU) by far. Besides the CPU, many today's computer systems include already such a powerful GPU which runs idle most of the time. Thus, it might be used as cheap and instantly available co-processor for general-purpose applications.
In this chapter, we focus on the efficient processing of computationally expensive operations in asymmetric cryptosystems on the off-the-shelf NVIDIA 8800 GTS graphics card by use of NVIDIA's CUDA programming model. We present improved and novel implementations employing GPUs as accelerator for RSA and DSA cryptosystems as well as for ECC.

Contents of this Chapter

4.1 Motivation

For the last twenty years graphics hardware manufacturers have focused on producing fast Graphics Processing Units (GPUs), specifically for the gaming community. This has more recently led to devices which outperform general purpose Central Processing Units (CPUs) for specific applications, particularly when comparing the MIPS (million instructions per second) benchmarks. Hence, a research community has been established to use the immense power of GPUs for general purpose computations (GPGPU). In the last two years, prior limitations of the graphics application programming interfaces (API) have been removed by GPU manufacturers

by introducing unified processing units in graphics cards. They support a general purpose instruction set by a native driver interface and framework.

In the field of asymmetric cryptography, the security of all practical cryptosystems rely on hard computational problems strongly dependent on the choice of parameters. But with rising parameter sizes (often in the range of 1024–4096 bits), however, computations become more and more challenging for the underlying processor. For modern hardware, the computation of a *single* cryptographic operation is not critical, however in a many-to-one communication scenario, like a central server in a company's data processing center, it may be confronted with hundreds or thousands of simultaneous connections and corresponding cryptographic operations. As a result, the most common current solution are cryptographic accelerator cards. Due to the limited market, their price tags are often in the range of several thousands euros or US dollars. The question at hand is whether commodity GPUs can be used as high-performance public-key accelerators.

In this chapter, we will present novel implementations of cryptosystems based on modular exponentiations and elliptic curve operations on recent graphics hardware. To the best of our knowledge, this is the first publication making use of the CUDA framework for GPGPU processing of asymmetric cryptosystems. We will start with implementing the extremely widespread RSA cryptosystem [RSA78]. The same implementation based on modular exponentiation for large integers can be used to implement the Digital Signature Algorithm (DSA) [Nat00] which also has been adopted to elliptic curve groups in the ANSI X9.62 standard [ANS05]. The implementation of this DSA variant for elliptic curves (ECDSA) is our second goal.
All presented results in this chapter originated from joint work with Robert Szerwinski [SG08].

4.2 Previous Work

Recently, the research community has started to explore techniques to accelerate cryptographic algorithms using the GPU. For example, various authors looked at the feasibility of the current industry standard for *symmetric* cryptography, the Advanced Encryption Standard (AES) [Man07, Ros07, HW07, CIKL05]. Two groups, Moss *et al.* [MPS07] and Fleissner, have aimed for the efficient implementation of modular exponentiation on former generations of GPU [Fle07]. Their results were not promising, as they were limited by the legacy GPU architecture and interface. To the best of our knowledge, we here present the first implementation of RSA and ECC on modern hardware using the CUDA programming model.

We will implement the core operations for both systems efficiently on modern graphics hardware, creating the foundation for the use of GPUs as accelerators for public key cryptography. For our work we use NVIDIA's G80 generation, together with its new GPGPU interface CUDA.

4.3 General-Purpose Applications on GPUs

The following section will give an overview over traditional GPU computing, followed by a more in-depth introduction to NVIDIA's general purpose interface CUDA.

4.3.1 Traditional GPU Computing

Roughly, the pipeline for processing graphical elements in GPUs consist of the stages *transform & light*, *assemble primitives*, *rasterize* and *shade*. First GPUs implemented a static graphics pipeline, but over time more and more stages became *programmable* by introducing specialized processors, e.g., vertex and fragment processors that made the transform & light and shading stages, respectively, more flexible.

When processing power of such devices increased while prices kept falling, the research community thought of ways to use these resources for computationally-intense tasks other than graphics processing. However, as the processors' capabilities were very limited and the API of the graphics driver was specifically built to implement the graphics pipeline, a lot of overhead needed to be taken into account. For example, all data had to be encoded in textures which are two dimensional arrays of pixels storing color values for red, green, blue and an additional alpha channel used for transparency. Additionally, textures are *read-only* objects, which forced the programmers to compute one step of an algorithm, store the result in the frame buffer, and start the next step using a texture reference to the generated pixels. This technique is known as *ping-ponging*. Most GPUs did only provide instructions to manipulate floating point numbers, forcing GPGPU programmers to map integers onto the available mantissa and find ways to emulate bit-logical functions, e.g., by using look-up tables.

These limitations have been the main motivation for the key GPU manufacturers ATI/AMD and NVIDIA to create APIs specifically for the GPGPU community and modify their hardware for better support: ATI's solution is called Close To the Metal (CTM) [ATI06], while NVIDIA presented the Compute Unified Device Architecture (CUDA), a radically new design that makes GPU programming and GPGPU switch places: The underlying hardware of the G80 series is an accumulation of *scalar* common purpose processing units ("unified" design) and quite a bit of "glue" hardware to efficiently map the graphics pipeline to this new design. GPGPU applications however directly map to the target hardware and thus graphics hardware can be programmed without any graphics API whatsoever.

4.3.2 Programming GPUs using NVIDIA's CUDA Framework

In general, the GPU's immense computation power mainly relies on its inherent parallel architecture. For this, the CUDA framework introduces the thread as smallest unit of parallelism, i.e., a small piece of concurrent code with associated state. However, when compared to threads on microprocessors, GPU threads have much lower resource usage and lower creation and switching cost. Note that GPUs are only effective when running a *high number* of such threads. A group of threads that is executed *physically* in parallel is called a warp. All threads in one warp are executed in *Single Instruction Multiple Data* (SIMD) fashion. If one or more thread(s) in the *same* warp need to execute different instructions, e.g., in case of a data-dependent jump, their execution will be serialized and the threads are called *divergent*. As the next level of parallelism, a (thread) block is a group of threads that can communicate with each other and synchronize their execution. The maximum number of threads per block is limited by the hardware. Finally,

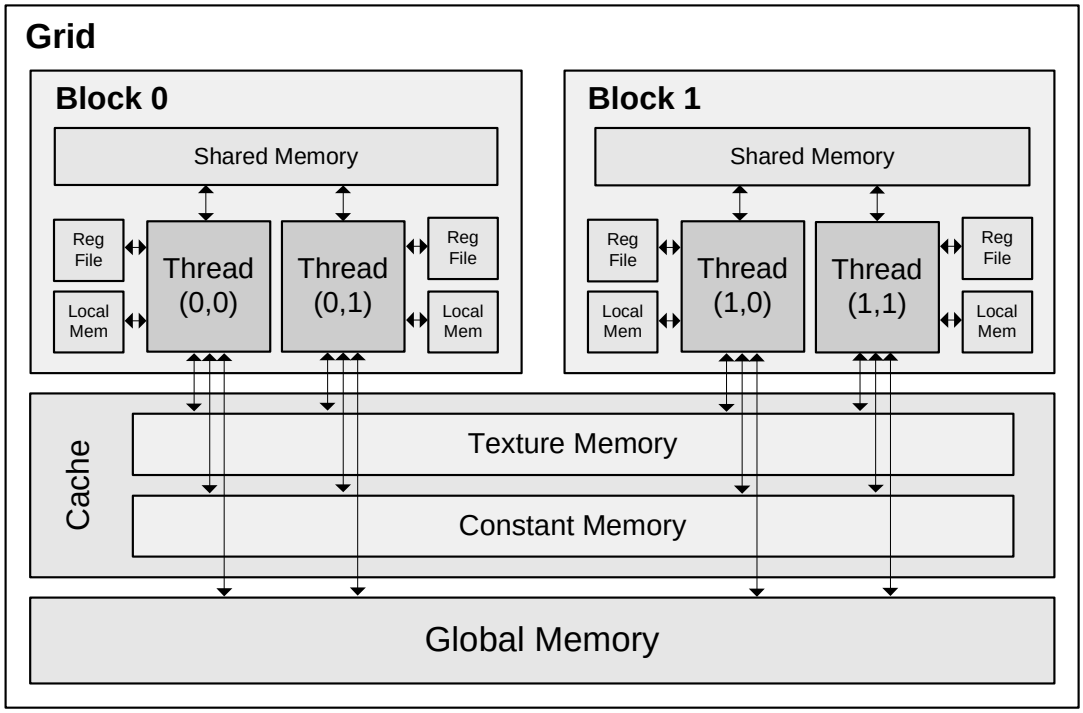


Figure 4.1: The memory and programming model for CUDA based applications.

a group of blocks that have same dimensionality and execute the same CUDA program *logically* in parallel is called grid.

To allow optimal performance for different access patterns, CUDA implements a hierarchical memory model, contrasting to the flat model normally found in PCs. Host (PC) and device (GPU) have their own memory areas, called *host memory* and *device memory*, respectively. CUDA supplies optimized functions to transfer data between these separate spaces.

Each thread possesses its own register file, which can be read and written. Additionally, it can access its own copy of so-called local memory. All threads in the same *grid* can access the same on-chip read- and writable shared memory region. To prevent hazards resulting from concurrent execution of threads synchronization mechanisms must be used. Shared memory is organized in groups called banks that can be accessed in parallel. All threads can access a read- and writable memory space called global memory and read-only regions called constant memory and texture memory. The second last is optimized for one-dimensional locality of accesses, while the last is most effective when being used with two-dimensional arrays (matrices). Note that the texture and constant memories are the only regions that are cached. Thus, all accesses to the off-chip regions global and local memory have a high access latency, resulting in penalties when being used too frequently. The programming and memory model of CUDA is depicted in Figure 4.1.

The hardware consists of a number of so-called *multiprocessors* that are build from SIMD processors, on-chip memory and caches. Clearly, one processor executes a particular thread, the same warp being run on the multiprocessor at the same time. One or more blocks are

mapped to each multiprocessor, sharing its resources (registers and shared memory) and get executed on a time-sliced basis. When a particular block has finished its execution, the scheduler starts the next block of the grid until all blocks have been run.

Design Criteria for GPU Implementations

To achieve optimal performance using CUDA, algorithms must be designed to run in a multitude of parallel threads and take advantage of the presented hierarchical memory model. Now, we enumerate the key criteria necessary for gaining the most out of the GPU by following the CUDA programming guide [CUD07] and a talk given by Mark Harris of NVIDIA [Har07].

A. ***Maximize the use of available processing power***

- A1 **Maximize independent parallelism** in the algorithm to enable easy partitioning in threads and blocks.
- A2 **Keep resource usage low** to allow concurrent execution of as many threads as possible, i.e., use only a small number of registers per thread and shared memory per block.
- A3 **Maximize arithmetic intensity**, i.e., match the arithmetic to bandwidth ratio to the GPU design philosophy: GPUs spend their transistors on ALUs, not caches. Bearing this in mind allows to hide memory access latency by the use of independent computations (*latency hiding*). Examples include using arithmetic instructions with high throughput as well as re-computing values instead of saving them for later use.
- A4 **Avoid divergent threads** in the *same* warp.

B. ***Maximize use of available memory bandwidth***

- B1 **Avoid memory transfers between host and device** by shifting more computations from the host to the GPU.
- B2 **Use shared memory** instead of global memory **for variables**.
- B3 **Use constant or texture memory** instead of global memory **for constants**.
- B4 **Coalesce global memory accesses**, i.e., choose access patterns that allow to combine several accesses in the same warp to one, wider access.
- B5 **Avoid bank conflicts** when utilizing shared memory, i.e., choose patterns that result in the access of *different* banks per warp.
- B6 **Match access patterns** for constant and texture memory **to** the **cache design**.

CUDA Limitations

Although CUDA programs are written in the C language together with extensions to support the memory model, allow synchronization and special intrinsics to access faster assembler instructions, it also contains a number of limitations that negatively affect efficient implementation

of public key cryptography primitives. Examples are the lack for additions/subtractions with carry as well as the missing support for inline assembler instructions[1].

4.4 Modular Arithmetic on GPUs

In the following section we will give different ways do realize *modular arithmetic* on a GPU efficiently, keeping the mentioned criteria in mind. For the RSA cryptosystem we need to implement arithmetic modulo N, where N is the product of two large primes p and q: $N = p \cdot q$. The arithmetic of both DSA systems, however, is based on the prime field $GF(p)$ as the lowest-level building block. Note that the DSA systems both use a *fixed* – in terms of sessions or key generations – prime p, thus allowing to choose special primes at build time that have advantageous properties when reducing modulo p. As the RSA modulus N is the product of the two secret primes p and q that will be chosen secretly *for each* new key pair, we cannot apply specific optimizations for the modulus in this case.

Modular Addition and Subtraction

In general, addition $s \equiv a + b \bmod m$ of two operands a and b, where $0 \leq a, b < m$, is straightforward (cf. Section 3.4.2), as the result of the plain addition operation $a + b$ always satisfies $0 \leq a + b < 2m$ and therefore needs at maximum one subtraction of m to fulfil $0 \leq s < m$. Due to the SIMD design, we require algorithms that have a uniform control flow in all cases and compute both $a + b$ and $a + b - m$ and decide afterwards which is the correctly reduced result, cf. Criterion A4 and Section 3.4.2. Subtraction $d \equiv a - b \bmod m$ can be treated similarly: we compute both $a - b$ and $a - b + m$ and use a sign test at the end to derive the correctly reduced result.

Modular Multiplication

Multi-precision modular multiplication $r \equiv a \cdot b \bmod m$ is usually the most critical operation in common asymmetric cryptosystems. In a straightforward approach to compute r, we derive a double-sized product $r' = ab$ first and reduce afterwards by multi-precision division. Besides the quadratic complexity of standard multiplication, division is known to be very costly and should be avoided whenever possible. Thus, we will discuss several multiplication strategies to identify an optimal method for implementation on GPUs.

4.4.1 Montgomery Modular Multiplication

In 1985 Peter L. Montgomery proposed an algorithm [Mon85] to remove the costly division operation from the modular reduction. For a given modulus M, choose a radix $R > M$ which is co-prime to M and that allows computations modulo R to be computed efficiently, i.e., a

[1] NVIDIA published their own (abstract) assembler language PTX [PTX07], however as of CUDA version 1.0 one kernel *cannot* contain code both generated from the C language and PTX.

corresponding power of the radix used by the machine. A possible configuration could be $R = 2^{wn}$ for w bit registers and n words per multi-precision integer.

To use the reduction algorithm, we first need to convert the parameters (say $a \in \mathbb{Z}_M$) to Montgomery form by computing $A = aR \bmod M$. Furthermore, we need to determine an auxiliary value $M' = -M^{-1} \bmod R$, $0 \leq M' < R$ which is a precomputed constant for a given tuple (M, R). Based on these preliminaries, given an $X < RM$ in Montgomery form, we can use Algorithm 4.1 to compute $XR^{-1} \bmod M$ very efficiently by constructing an integer $v = X + fM$ that is a multiple of R. This, in turn, allows to simply divide by R to compute the desired result $x = XR^{-1} \bmod M$.

Algorithm 4.1 Montgomery Reduction $MonRed(X) = XR^{-1} \bmod M$

Input: A modulus M and a radix R so that $R > M$ and $\gcd(R, M) = 1$;
an auxiliary value $M' = -M^{-1} \bmod R$ and an unsigned integer X, $0 \leq X < RM$.
Output: The integer $x = XR^{-1} \bmod M$, $0 \leq x < M$.

1: $f \leftarrow X \cdot M' \bmod R$ $\{\Rightarrow 0 \leq f < R\}$
2: $v \leftarrow X + f \cdot M$
3: $w \leftarrow v/R$
4: **if** $w \geq M$ **then** {final reduction}
5: $\quad w \leftarrow w - M$
6: **end if**
7: **return** w

More precisely, Algorithm 4.1 first computes f which is a multiple of M that needs to be added to the input value X in order to determine a value v that is also a multiple of R, i.e., $X + fM \equiv 0 \bmod R$, while not changing the congruency modulo M ($X + fM \equiv X \bmod M$). Such a value can be found by computing $f = XM' \bmod R$ and leads to

$$v = X + fM = X + (XM')M = X + (-XM^{-1})M \equiv 0 \bmod R. \tag{4.1}$$

By construction, v is a multiple of R and thus v/R is indeed an integer. It remains to show that $0 \leq w < 2M$ so that a simple subtraction at the end of the algorithm suffices to ensure $0 \leq x < M$. Recall that the input is bound to $X < RM$ and the fact that $0 \leq f < R$ holds. Then the following inequality holds

$$w = \frac{X + fM}{R} \quad < \quad \frac{RM + RM}{R} = 2M, \tag{4.2}$$

so that we yield $x = XR^{-1} \bmod M$ with $0 \leq x < M$ as desired after final reduction. Algorithm 4.1 can also be used to transform parameters between standard and Montgomery representation. For a given integer x in standard representation, $X = MonRed(xR^2)$ converts it to Montgomery domain and $x = MonRed(X)$ back to standard domain.

Koç et al. [cKKAK96] provide a survey of different implementation options of this algorithm combined with multi-precision multiplication $X = A \times B$. As all these multi-precision algorithms feature no inherent parallelism except the possibility to pipeline, we do *not* consider them optimal for our platform and implement the method with the lowest temporary space requirement of $n+2$ words, Coarsely Integrated Operand Scanning (CIOS), as a reference solution only (cf. Algorithm 4.2).

Algorithm 4.2 Montgomery Multiplication for Multi-Precision Integers (CIOS Method) [cKKAK96]

Input: Modulus M and radix $R = 2^{wn}$ so that $R > M$ and $\gcd(R, M) = 1$; $M'_0 = (-M^{-1} \bmod R) \bmod 2^w$, two unsigned integers $0 \leq A, B < M$ in Montgomery form, i.e., $X = (X_{n-1}X_{n-2}\ldots X_0)_{2^w}$ for $X \in \{A, B, M\}$.

Output: The product $C = ABR^{-1} \bmod M$, $0 \leq C < M$, in Montgomery form.

1: $T \leftarrow 0$
2: **for** i **from** 0 **to** $n-1$ **do**
3: $\quad c \leftarrow 0$
4: $\quad$ **for** j **from** 0 **to** $n-1$ **do** {Multiplication}
5: $\quad\quad (c, T_j) \leftarrow A_j \cdot B_i + T_j + c$
6: $\quad$ **end for**
7: $\quad (T_{n+1}, T_n) \leftarrow T_n + c$
8: $\quad m \leftarrow T_0 \cdot M'_0 \bmod 2^w$ {Reduction}
9: $\quad (c, T_0) \leftarrow m \cdot M_0 + T_0$
10: $\quad$ **for** j **from** 1 **to** $n-1$ **do**
11: $\quad\quad (c, T_{j-1}) \leftarrow m \cdot M_j + T_j + c$
12: $\quad$ **end for**
13: $\quad T_{n-1} \leftarrow T_n + c$
14: $\quad T_n \leftarrow T_{n+1} + c$
15: **end for**
16: **return** $(T_{n-1}T_{n-2}\ldots T_0)_{2^w}$

4.4.2 Modular Multiplication in Residue Number Systems (RNS)

As an alternative approach to conventional base-2^w arithmetic, we can represent integers based on the idea of the Chinese Remainder Theorem, by encoding an integer x as a tuple formed from its residues x_i modulo n relatively prime w-bit moduli m_i, where $|x|_{m_i}$ denotes $x \bmod m_i$:

$$\langle x \rangle_{\mathcal{A}} = \langle x_0, x_1, \ldots, x_{n-1} \rangle_{\mathcal{A}} = \langle |x|_{m_0}, |x|_{m_1}, \ldots, |x|_{m_{n-1}} \rangle_{\mathcal{A}} \tag{4.3}$$

Here, the ordered set of relatively prime moduli $(m_0, m_1, \ldots, m_{n-1})$, $\gcd(m_i, m_j) = 1$ for all $i \neq j$, is called *base* and denoted by $\mathcal{A}$. The product of all moduli, $A = \prod_{i=0}^{n-1} m_i$ is called *dynamic range* of $\mathcal{A}$, i.e., the number of values that can be *uniquely* represented in $\mathcal{A}$. In other

words, all numbers in $\mathcal{A}$ get implicitly reduced modulo A. Such a representation in RNS has the advantage that addition, subtraction and multiplication can be computed *independently* for all residues:

$$\langle x\rangle_{\mathcal{A}} \circ \langle y\rangle_{\mathcal{A}} = \langle |x_0 \circ y_0|_{m_0}, |x_1 \circ y_1|_{m_1}, \ldots, |x_{n-1} \circ y_{n-1}|_{m_{n-1}}\rangle_{\mathcal{A}}, \qquad \circ \in \{+, -, \cdot\} \tag{4.4}$$

which allows carry-free computations[2] and multiplication without partial products. However, some information involving the whole number x cannot be easily computed. For instance, sign and overflow detection and comparison of magnitude are hard, resulting from the fact that residue number systems are no weighted representation. Furthermore, division and as a result reduction modulo an arbitrary modulus $M \neq A$ is *not* as easy as in other representations.

But similar to the basic idea of Montgomery multiplication, one can create a modular multiplication method for input values in RNS representation as shown in Algorithm 4.3, which involves a second base $\mathcal{B} = (\widetilde{m}_0, \widetilde{m}_1, \ldots, \widetilde{m}_{n-1})$ with corresponding dynamic range B. It computes a value $v = XY + fM$ that is equivalent to 0 mod A and XY mod M. Thus, we can safely divide by A, i.e., multiply by its inverse modulo B, to compute the output XYA^{-1} mod M. Note that the needed reduction modulo A to compute f is *free* in $\mathcal{A}$.

Algorithm 4.3 Modular Multiplication Algorithm for Residue Number Systems [KKSS00]

Input: Modulus M, two RNS bases $\mathcal{A}$ and $\mathcal{B}$ composed of n distinct moduli m_i each, $\gcd(A, B) = \gcd(A, M) = 1$ and $B > A > 4M$.
Two factors X and Y, $0 \leq X, Y < 2M$, encoded in both bases and in Montgomery form, i.e. $\langle X\rangle_{\mathcal{A}}, \langle X\rangle_{\mathcal{B}}$ and $\langle Y\rangle_{\mathcal{A}}, \langle Y\rangle_{\mathcal{B}}$, $X = xA$ mod M and $Y = yA$ mod M.

Output: The product $C = XYA^{-1}$ mod M, $0 \leq C < 2M$, in both bases and Montgomery form.

1: $\langle u\rangle_{\mathcal{A}} \leftarrow \langle X\rangle_{\mathcal{A}} \cdot \langle Y\rangle_{\mathcal{A}}$ and $\langle u\rangle_{\mathcal{B}} \leftarrow \langle X\rangle_{\mathcal{B}} \cdot \langle Y\rangle_{\mathcal{B}}$
2: $\langle f\rangle_{\mathcal{A}} \leftarrow \langle u\rangle_{\mathcal{A}} \cdot \langle -M^{-1}\rangle_{\mathcal{A}}$
3: $\langle f\rangle_{\mathcal{B}} \leftarrow \text{BaseExtend}(\langle f\rangle_{\mathcal{A}})$
4: $\langle v\rangle_{\mathcal{B}} \leftarrow \langle u\rangle_{\mathcal{B}} + \langle f\rangle_{\mathcal{B}} \cdot \langle M\rangle_{\mathcal{B}}$ $\{\langle v\rangle_{\mathcal{A}} = 0$ by construction$\}$
5: $\langle w\rangle_{\mathcal{B}} \leftarrow \langle v\rangle_{\mathcal{B}} \cdot \langle A^{-1}\rangle_{\mathcal{B}}$
6: $\langle w\rangle_{\mathcal{A}} \leftarrow \text{BaseExtend}(\langle w\rangle_{\mathcal{B}})$
7: **return** $\langle w\rangle_{\mathcal{A}}$ and $\langle w\rangle_{\mathcal{B}}$

All steps of the algorithm can be efficiently computed in parallel. However, a method to convert between both bases, a *base extension* mechanism, is required. We take four different options into account: the method based on a Mixed Radix System (MRS) according to Szabó and Tanaka [ST67], as well as CRT-based methods due to Shenoy and Kumaresan [SK89], Kawamura *et al.* [KKSS00] and Bajard *et al.* [BDK01]. We present a brief introduction of these methods, but for more detailed information about base extensions, please see the recent survey in [BP04].

[2]Note that inner-RNS operations still contain carries.

4.4.3 Base Extension Using a Mixed Radix System (MRS)

The classical way to compute base extensions is due to Szabó and Tanaka [ST67]. Let $(m_0, \ldots, m_{n-1})$ be the MRS base *associated* to $\mathcal{A}$. Then, each integer x can be represented in a *mixed radix system* as

$$x = x'_0 + x'_1 m_0 + x'_2 m_0 m_1 + \cdots + x'_{n-1} m_0 \ldots m_{n-2}. \tag{4.5}$$

The MRS digits x'_i can be derived from the residues x_i by a recursive strategy:

$$\begin{aligned} x'_0 &= x_0 \bmod m_0 \\ x'_1 &= (x_1 - x'_0) m^{-1}_{(1,0)} \bmod m_1 \\ &\vdots \\ x'_{n-1} &= (\cdots((x_n - x'_0) m^{-1}_{(n-1,0)} - x'_1) m^{-1}_{(n-1,1)} - \cdots - x'_{n-2}) m^{-1}_{(n-1,n-2)} \bmod m_{n-1} \end{aligned} \tag{4.6}$$

where $m^{-1}_{(i,j)}$ are the precomputed inverses of m_j modulo m_i. To convert x from this representation to a target RNS base, we could reduce Equation (4.5) by each target modulus $\widetilde{m}_k$, involving precomputed constants $\widetilde{c}_{(k,i)} = \left|\prod_{l=0}^{i-1} m_l\right|_{\widetilde{m}_k}$. But instead of creating a table for all $\widetilde{c}_k$, a recursive approach is more efficient in our situation, eliminating the need for table-lookups [BMP05], and allowing to compute all residues in the target base in parallel:

$$|x|_{\widetilde{m}_k} = \left|(\ldots((x'_{n-1} m_{n-2} + x'_{n-2}) m_{n-3} + x'_{n-3}) m_{n-4} + \cdots + x'_1) m_0 + x_0\right|_{\widetilde{m}_k} \tag{4.7}$$

4.4.4 Base Extension Using the Chinese Remainder Theorem (CRT)

We first present a brief introduction to the Chinese Remainder Theorem (CRT) before describing corresponding base extension techniques. Let $m_0, m_1, \ldots, m_{n-1}$ be n moduli that are relatively prime to each other, i.e. $\gcd(m_i, m_j) = 1$ for all $i \neq j$. Then there exists a unique solution modulo $M = \prod_{i=0}^{n-1} m_i$ solving the simultaneous congruences

$$\begin{aligned} x &\equiv x_0 \bmod m_0 \\ x &\equiv x_1 \bmod m_1 \\ &\vdots \\ x &\equiv x_{n-1} \bmod m_{n-1} \end{aligned}$$

that can be computed as follows:

$$x = \left(\sum_{i=0}^{n-1} \hat{M}_i \left|\frac{x_i}{\hat{M}_i}\right|_{m_i}\right) \bmod M \tag{4.8}$$

with $\hat{M}_i = \frac{M}{m_i}$. An equivalent version not including the reduction modulo M is

$$x = \sum_{i=0}^{n-1} \hat{M}_i \left|\frac{x_i}{\hat{M}_i}\right|_{m_i} - \alpha M \tag{4.9}$$

where α is an integer so that $0 \leq x < M$.

Now we take this definition and adapt it to the source base $\mathcal{A}$ with dynamic range A:

$$x = \sum_{k=0}^{n-1} \hat{A}_k \left| \frac{x_k}{\hat{A}_k} \right|_{m_k} - \alpha A, \qquad \alpha < n \tag{4.10}$$

where $\hat{A}_k = A/m_k$ and α is an integer so that $0 \leq x < A$. Note that α is strictly upper-bounded by n. When reducing this equation with an arbitrary target modulus, say $\widetilde{m}_i$, we obtain

$$|x|_{\widetilde{m}_i} = \left| \sum_{k=0}^{n-1} \left| \hat{A}_k \right|_{\widetilde{m}_i} \delta_k \; - |\alpha A|_{\widetilde{m}_i} \right|_{\widetilde{m}_i}, \qquad \delta_k = \left| x_k \cdot {\hat{A}_k}^{-1} \right|_{m_k} \tag{4.11}$$

where $\left|\hat{A}_k\right|_{\widetilde{m}_i}$, $\left|{\hat{A}_k}^{-1}\right|_{m_k}$ and $|A|_{\widetilde{m}_i}$ are precomputed constants. Note that the δ_k do *not* depend on the target modulus and can thus be reused in the computation of a different target residue.

This is an efficient way to compute all residues modulo the target base, provided we know the value of α. While involving a couple of look-ups for the constants as well, the instruction flow is highly uniform (cf. Criterion A4) and fits to our SIMD architecture, i.e., we can use n threads to compute the n residues of x in the target base in parallel (cf. to Criterion A1).

The first technique to compute such an α is due to Shenoy and Kumaresan [SK89] and requires a *redundant modulus* $m_r \geq n$ that is relatively prime to all other moduli m_j and $\widetilde{m}_i$, i.e., $\gcd(A, m_r) = \gcd(B, m_r) = 1$. Consider Equation 4.11, set $\widetilde{m}_i = m_r$ and rearrange it to the following:

$$|\alpha|_{m_r} \;=\; \left| |A^{-1}|_{m_r} \cdot \left(\sum_{k=0}^{n-1} \left| \hat{A}_k \right|_{m_r} \delta_k - |x|_{m_r} \right) \right|_{m_r}. \tag{4.12}$$

Since $\alpha < n \leq m_r$ it holds that $\alpha = |\alpha|_{m_r}$ and thus Equation 4.12 computes the exact value of α, involving the additional constant $|A^{-1}|_{m_r}$.

Kawamura *et al.* propose a different technique that approximates α using fixed-point computations [KKSS00]. Consider Equation 4.11, rearrange it and divide by A:

$$\alpha \;=\; \sum_{k=0}^{n-1} \frac{\delta_k}{m_k} - \frac{|x|_{\widetilde{m}_i}}{A} \;=\; \left\lfloor \sum_{k=0}^{n-1} \frac{\delta_k}{m_k} \right\rfloor. \tag{4.13}$$

Next, they approximate α by using $\mathrm{trunc}_r(\delta_k)$ as numerator and 2^w as denominator and adding a properly chosen offset σ, where $\mathrm{trunc}_r(\delta_k)$ sets the last $w - r$ bits of δ_k to zero:

$$\alpha' = \left\lfloor \sum_{k=0}^{n-1} \frac{\mathrm{trunc}_r(\delta_k)}{2^w} \;+\; \sigma \right\rfloor = \left\lfloor \frac{1}{2^r} \sum_{k=0}^{n-1} \left\lfloor \delta_k / 2^{w-r} \right\rfloor \;+\; \sigma \right\rfloor, \tag{4.14}$$

Thus, the approximate value α' can be computed in fixed-point arithmetic as integer part of the sum of the r most-significant bits of all δ_k. Provided σ is chosen correctly, Equation 4.14 will compute $\alpha' = \alpha$, and the resulting base extension will be exact.

Finally, Bajard *et al.* follow the most radical approach possible [BDK01]: they allow an offset of $\alpha A \leq (n-1)A$ to occur in Equation 4.11 and thus do not need to compute α at all. After the first base extension we have $f' = f + \alpha A$ and thus $w' = w + \alpha M$, i.e., the result w' will contain a maximum offset of $(n-1)M$, and thus be equivalent to $w \mod M$. However, this technique needs additional measures of precaution in the multiplication algorithm, which predominantly condense in the higher dynamic ranges needed.

4.5 Implementation

In this section we will describe the implementation of two primitive operations for a variety of cryptosystems: first, we realize modular exponentiation on the GPU for use with RSA, DSA and similar systems. Second, for ECC-based cryptosystems we present an efficient point multiplication method which is the fundamental operation, e.g., for ECDSA or ECDH [HMV04].

4.5.1 Modular Exponentiation Using the CIOS Method

We implemented the CIOS Method as introduced in Algorithm 4.2 for sequential execution since it does *not* include any inherent parallelism. Fan *et al.* describe efficient ways to pipeline such an algorithm for the use on multi-core systems [FSV07]. This would however need fairly complex coordination and memory techniques and thus will not be considered further for our implementation, cf. Criteria A4 and B4-B6.

As all modular exponentiations are independent, we let each thread compute exactly one modular exponentiation in parallel with all others. Resulting from that, this solution only profits from coarse-grained parallelism. We assume the computation of distinct exponentiations, each having the *same* exponent t – for example RSA signatures using the same key – and thus need to transfer only the messages P_i for each exponentiation to the device and the result $P_i^t \mod N$ back to the host. As a result, every thread executes the same control flow, satisfying Criterion A4. To accelerate memory transfers between host and device, we use page-locked host memory and pad each message to a fixed length that forces the starting address of each message to values that are eligible for global memory coalescing (cf. Criteria B1 and B4).

For modular exponentiation based on Algorithm 4.2, we applied the straightforward binary right-to-left method [Sti05]. During exponentiation, each *thread* needs three temporary values of $(n+2)$ words each that get used as input and output of Algorithm 4.2 in a round-robin fashion by pointer arithmetic. Thus, $3(n+2)$ words are required. This leads to 408 bytes and 792 bytes for 1024 bits and 2048 bit parameters, respectively. Each multiprocessor features 16384 bytes of shared memory, resulting in a maximum number of $\lfloor 16386/408 \rfloor = 40$ and $\lfloor 16386/792 \rfloor = 20$ threads per multiprocessor for 1024 and 2048 bits, respectively, if we use shared memory for temporary values. Clearly, both solutions are inefficient when considering that each multiprocessor is able to execute 768 threads per block (i.e., we favor Criterion A2 over B2).

Thus, we chose to store the temporary values in *global memory*. We have to store the values *interleaved* so that memory accesses of one word by all threads in a warp can be combined to

one global memory access. Hence, for a given set of values $(A, B, C, \ldots)$ consisting each of $n+2$ words $X = (x_0, x_1, \ldots, x_{n+1})$, we store all first words $(a_0, b_0, c_0, \ldots)$ for all threads in the same block, then all second words $(a_1, b_1, c_1, \ldots)$, and so on (cf. Criterion B4).

Moreover, we have to use *nailing* techniques, as CUDA does not yet include add-with-carry instructions. Roughly speaking, nailing reserves one or more of the high-order bits of each word for the carry that can occur when adding two numbers. To save register and memory space, however, we store the full word of w bits per register and use bit shifts and `and`-masking to extract two nibbles, each providing sufficient bits for the carry (cf. Criterion A3). We here decompose a 32 bit addition in two 16 bit additions, including some overhead for carry handling.

4.5.2 Modular Exponentiation Using Residue Number Systems

Computations in residue number systems yield the advantage of being inherently parallel. According to Algorithm 4.3 all steps are computed in *one* base only, except for the first multiplication. Thus, the optimal mapping of computations to threads is as follows: each thread determines values for one modulus in the two bases. As a result, we have coarse-grained (different exponentiations) and fine-grained parallelism (base size), satisfying Criterion A1. We call n' the number of residues that can be computed in parallel, i.e., the number of threads per encryption. The base extension by Shenoy *et al.* needs a *redundant* residue starting from the first base extension to be able to compute the second base extension. To reflect this fact, we use two RNS bases $\mathcal{A}$ and $\mathcal{B}$, having n moduli each, and an additional residue m_r resulting in $n' = n + 1$. For all other cases, it holds that $n' = n$.

Considering the optimal number of bits per modulus, we are faced with $w = 32$ bit integer registers on the target hardware. Thus, to avoid multi-precision techniques, we can use moduli that are smaller than 2^w. The hardware can compute 24 bit multiplications faster than full 32 bit multiplications. However, CUDA does *not* expose an intrinsic to compute the most-significant 16 bits of the result. Using 16 bit moduli would waste registers and memory and increase the *number of memory accesses* as well. Thus, we prefer *full* 32 bit moduli to save storage resources at the expense of higher computational cost (cf. Criteria A2 and A3).

For Algorithm 4.2 to work, the dynamic ranges A and B and the modulus M have to be related according to $B > A > 2^2M$, or $B > A > (2+n)^2M$ when using Bajard's method. For performance reasons, we consider *full warps* of 32 threads only, resulting in a slightly reduced size of M. The figures for all possible combinations can be found in Table A.2 in the appendix. For input and output values, we assume that all initial values will have been already converted to both bases (and possibly the redundant modulus m_r) and that output values will be returned in the same encoding. Note that it would be sufficient to transfer values in *one* base only and do a base extension for all input values (cf. Criterion B1, transferring values in both bases results in a more compact kernel together with a slightly higher latency). Different from the CIOS method, temporary values can be kept local for each thread, i.e., every thread stores its assigned residues in registers. Principally all operations can be performed *in parallel* on different residues and –

as a result – the plain multiplication algorithm does *not* need any synchronizations. However, both properties do *not* hold for the base extension algorithms.

Mixed Radix Conversion

Recall that the mixed radix conversion computes the mixed radix representation from all residues in the source base first and uses this value to compute the target residues. The second step involves the computation of n' residues and can be executed in parallel, i.e., each thread computes the residue for its corresponding modulus. As a result, we have to store the n MRS digits in shared memory to make them accessible to all threads (cf. Criteria A1 and B2). The first step, however, is the main caveat of this algorithm due to its highly divergent nature as each MRS digit is derived from the residue of a temporary variable in a *different* modulus (and thus thread) and depends on all previously computed digits. This clearly contradicts to Criterion A4 and results in serialization of executions. Additionally, note that threads having already computed an MRS digit do not generate any useful output anymore.

CRT-based Conversion

The first step for all CRT-based techniques is to compute δ_k for each source modulus what can be carried out by one thread for each value. Second, all n' threads compute a weighted sum involving δ_k and a modulus-dependent constant. Note that all threads need to access *all* δ_k and thus δ_k have to be stored in shared memory (cf. Criterion B2). Third, α has to be derived, whose computation is the main difference in the distinguished techniques. α is needed by *all* threads later and thus needs to be stored in shared memory as well. After computing α all threads can proceed with their independent computations.

Bajard's method does not compute α and consequently needs no further operations. For Shenoy's method, the second step above is needed for the redundant modulus m_r as well, which can be done in parallel with all other moduli. Then, a *single* thread computes α and writes it to shared memory. The redundant residue m_r comes at the price of an additional thread, however, the divergent part required to compute α does only contain one addition and one multiplication modulo m_r. Kawamura's method needs to compute the sum of the r most significant bits of all δ_k. While the right-shift of each δ_k can be done using *all* threads, the sum over all shifted values and the offset has to be computed using a *single* thread. A final right-shift results in the integer part of the sum, namely α.

Comparison and Selection

Clearly, Bajard's method is the fastest since it involves no computation of α. Shenoy's method only involves a small divergent part. However, we pay the price of an additional thread for the redundant modulus, or equivalently decrease the size of M. Kawamura's technique consists of a slightly larger divergent part, however it does neither include look-ups nor further reduces the size of M.

Method		$\mathcal{A} \rightarrow \mathcal{B}$			
		MRC (M)	**Shenoy (S)**	**Kawamura (K)**	**Bajard (B)**
$\mathcal{B} \rightarrow \mathcal{A}$	**MRC (M)**	•	○	○	•
	Shenoy (S)	•	○	○	•
	Kawamura (K)	•	○	○	•
	Bajard (B)	○	○	○	○

Table 4.1: Possible and efficient combinations of base extension algorithms.

Not all base extension mechanisms can be used for both directions required for Algorithm 4.3. For Bajard's method, consider the consequence of an offset in the second base extension: we would compute some w'' in base $\mathcal{A}$ that is *not equal* to the w' in $\mathcal{B}$. As a result, neither $\langle w' \rangle_{\mathcal{A}}$ nor $\langle w'' \rangle_{\mathcal{B}}$ could be computed leading to an invalid input for a subsequent execution of Algorithm 4.3. Thus, their method is only available for $\mathcal{A} \rightarrow \mathcal{B}$ conversions. Shenoy's method can only be used for the *second* base extension as there is no efficient way to carry the redundant residue through the computation of f modulo A. The technique by Kawamura *et al.* would in principle be available for both conversions. However, the sizes of both bases would be different to allow proper reduction in the $\mathcal{A} \rightarrow \mathcal{B}$ case, thus we exclude this option from our consideration. Table 4.1 shows the available and efficient combinations.

4.5.3 Point Multiplication Using Generalized Mersenne Primes

For implementation of the elliptic curve group operation, we chose mixed affine-Jacobian coordinates [ACD+05] to avoid costly inversions in the underlying field. We thus focused on efficient implementation of modular multiplication which is the remaining time-critical operation. For this, we used a straightforward schoolbook-type multiplication combined with the efficient reduction technique for the generalized Mersenne prime presented in Algorithm 3.1.

As for the CIOS method, there is no intrinsic parallelism except pipelining in this approach (cf. Criterion A1). Thus, we use one thread per point multiplication. We assume the use of the same base point P per point multiplication kP as defined in the NIST standard and *varying* scalars k. Thus, the only input that has to be transferred are the scalars. Secondly, we transfer the result in projective Jacobian coordinates back to the host. For efficiency reasons, we encode all coordinates interleaved for each threads in a block again.

We used shared memory to store all temporary values, nailed to 28 bits to allow schoolbook multiplication without carry propagation. Thus, we need 8 words per coordinate. Point addition and doubling algorithms were inspired by `libseccure` [Poe06]. With this approach shared memory turns out to be the limiting factor. Precisely, we require 111 words per point multiplication to store 7 temporary coordinates for point addition and modulo arithmetic, *two* points and each scalar. This results in 444 bytes of shared memory and a maximum of $\lfloor 16384/444 \rfloor = 36$ threads per multiprocessor. This leaves still room for improvements as Criterion A1 is *not* satisfied. However, due to internal problems within the toolchain, we were not (yet) able to compile

Base Ext. $\mathcal{A} \to \mathcal{B}$	Base Ext. $\mathcal{B} \to \mathcal{A}$	Throughput (1024 bits) [Enc/s] (rel.)	Throughput (2048 bits) [Enc/s] (rel.)
M	**M**	194 (46%)	28 (50%)
B	**M**	267 (63%)	38 (67%)
B	**K**	408 (97%)	55 (98%)
B	**S**	419 (100%)	56 (100%)

Table 4.2: Results for different Base Extension Techniques (RNS Method).

a solution that uses global memory for temporary values instead. Note that the left-to-right binary method for point multiplication demands only one temporary point. However, for the sake of a homogeneous flow of instructions we compute both possible solutions per scalar bit and use a small divergent section to decide which of them is the desired result (cf. Criterion A4).

4.6 Conclusions

With the previously discussed implementations on GPUs at hand, we finally need to identify the candidate providing the best performance for modular exponentiation.

4.6.1 Results and Applications

Before presenting the benchmark results of the best algorithm combinations we show our results regarding the different base extension options for the RNS method. The benchmarking scheme was the following: first, we did an exhaustive search for the number of registers per thread that can principally be generated by the tool chain. Then, we benchmarked all available execution configurations for these numbers of registers. To make the base extension algorithms comparable, we would have to repeat this for all possible combinations, as shown in Table 4.1. The results for the particular best configuration can be found in Table 4.2.

Clearly, the mixed radix based approach also used in [MPS07] cannot compete with CRT-based solutions. Kawamura *et al.* is slower than the method of Shenoy *et al.*, but performs only slightly worse for the 2048 bit range. Figure 4.2 shows the time over the number of encryptions for the four cases and the 1024 bit and 2048 bit ranges, respectively.

Both graphs show the characteristic behavior: Depending on the number of blocks that are started on the GPU and the respective execution configuration we get stair-like graphs. Only multiples of the number of warps per multiprocessor and the number of multiprocessors result in optimal configurations that fully utilize the GPU. However, depending on the number of registers per thread and the amount of shared memory used other configurations are possible and lead to smaller steps in between.

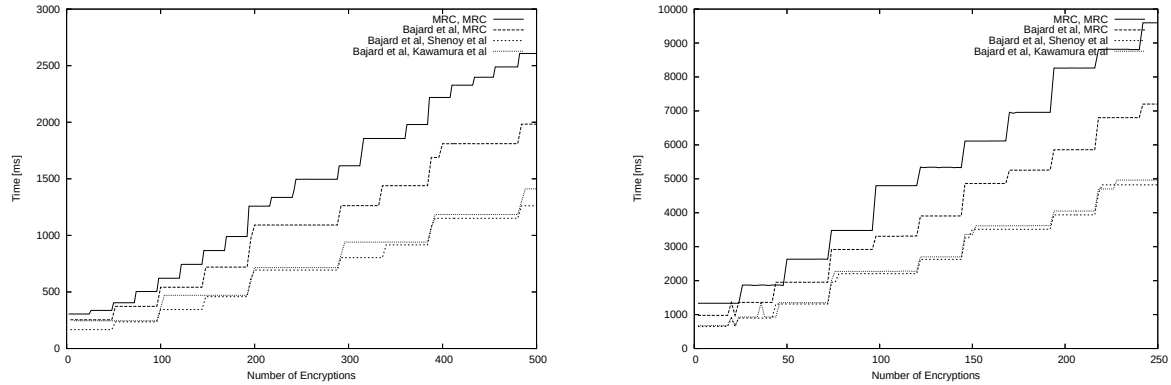


Figure 4.2: Results for modular exponentiation with about 1024 (left) and 2048 bit (right) moduli for different base extension methods, based on a NVIDIA 8800 GTS graphics card.

Optimized Implementations

Beside the reference implementation based on the CIOS algorithm, we selected as best choice the CRT-RNS method based on a combination of Bajard's and Shenoy's methods to compute the first and second base extension of Algorithm 4.3, respectively.

The selection of the implementation was primarily motivated to achieve high throughput rather than a small latency. Hence, due to the latency, not all implementations might be suitable for all practical applications. To reflect this, we present figures for data throughput as well as the initial latency t_{min} required at the beginning of a computation. Note that our results consider optimal configurations of warps per block and blocks per grid only. Table 4.3 shows the figures for modular exponentiation with 1024 and 2048 bit moduli and elliptic curve point multiplication using NIST's P-224 curve.

The throughput is determined from the number of encryptions divided by the elapsed time. Note that this *includes* the initial latency t_{min} at the beginning of the computations. The

Technique	Throughput [OPs/s]	Throughput [ms/OP]	Latency t_{min} [ms]	OPs at t_{min}
ModExp-1024 CIOS	813.0	1.2	6930	1024
ModExp-1024 RNS	439.8	2.3	144	4
ModExp-2048 CIOS	104.3	9.6	55184	1536
ModExp-2048 RNS	57.9	17.3	849	4
ECC PointMul-224	1412.6	0.7	305	36

Table 4.3: Results for throughput and minimum latency t_{min} on a NVIDIA 8800 GTS graphics card.

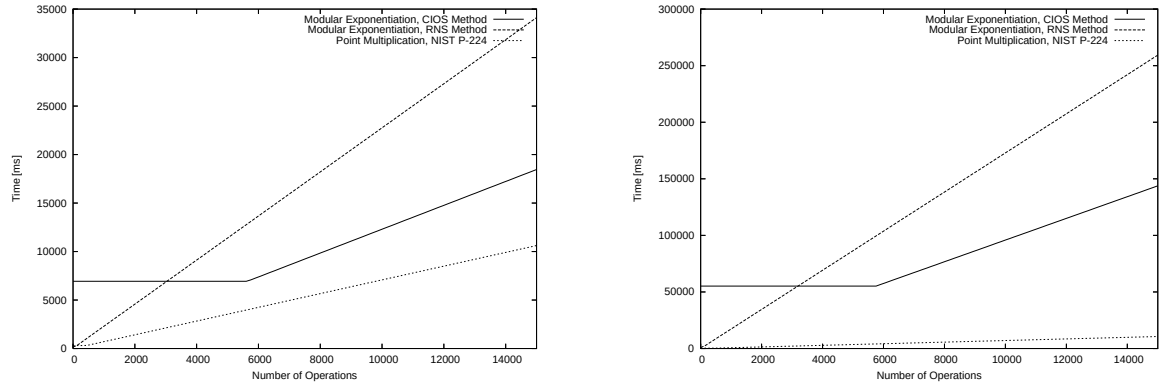


Figure 4.3: Results for modular exponentiation with about 1024 (left) and 2048 bit (right) moduli and elliptic curve point multiplication on NIST's P-224 curve, based on a NVIDIA 8800 GTS graphics card.

corresponding graphs are depicted in Figure 4.3. Note the relatively long plateau when using the CIOS technique. It is a direct result from having coarse-grained parallelism only: the smallest number of encryptions that can be processed is 128 times higher than for the RNS method. Its high offset is due to storing temporary values in global memory: memory access latency is hidden by scheduling independent computations, however the time needed to fetch/store the first value in each group cannot be hidden.

Clearly, the CIOS method delivers the highest throughput at the price of a high initial latency. For interactive applications such as online banking using TLS this will be a major obstacle. However, non-interactive applications like a *certificate authority* (CA) might benefit from the raw throughput. Note that both applications will share the *same* secret key for all digital signatures when using RSA. In case of ECC (ECDSA) however, *different* exponents were taken into account.

The residue number system based approach does only feature roughly half of the throughput but provides a more immediate data response. Thus, this method seems to be suitable even in interactive applications. Last but not least elliptic curve cryptography clearly outperforms modular exponentiation based techniques not only due to the much smaller parameters. With respect to other hardware and software implementations compared against our results in the next section, we present an ECC solution which outperforms most hardware devices and comes close the the performance of recent dual-core microprocessors.

4.6.2 Comparison with Previous Implementations

Due to the novelty of general purpose computations on GPUs and since directly comparable results are rare, we will take reference to recent hardware and software implementations in literature as well. To give a feeling for the different GPU generations, we provide Table 4.4.

Moss *et al.* implemented modular exponentiation for 1024 bit moduli on NVIDIA's 7800GTX GPU [MPS07], using the same RNS approach but picking different base extension mechanisms.

GPU	**Shader clock [MHz]**	**Shaders**	**Fill Rate [GPixels/s]**	**Mem Bandwidth** [GB/s]	CUDA
7800GTX			13.2	54.4	no
8800GTS	1200	92	24.0	64.0	yes
8800GTX	1350	128	36.8	86.4	yes
9800GX2	1500	$2 \cdot 128$	76.8	128.0	future

Table 4.4: Feature comparison of NVIDIA GPU platforms.

The authors present the *maximum* throughput only that has been achieved at the cost of an unspecified but high latency. Fleissner's recent analysis on modular exponentiation for GPUs is based on 192 bit moduli but relates the GPU performance solely to the CPU of his host system.

Costigan and Scott implemented modular exponentiation on IBM's Cell platform, i.e., a Sony Playstation 3 and an IBM MPM blade server, both running at 3.2 GHz [CS07]. We only quote the best figures for the Playstation 3 as they call the results for the MPM blade preliminary. The Playstation features one PowerPC core (PPU) and 6 *Synergistic Processing Elements* (SPUs). Software results have been attained from ECRYPT's eBATS project [ECR07]. Here, we picked a recent Intel Core2 Duo with 2.13 GHz clock frequency. Since mostly all figures for software relate to cycles, we assumed that repeated computations can be performed *without* interruption on *all* available cores so that no further cycles are spent, e.g., on scheduling or other administrative tasks of the operating system. Note that this might be an optimistic assumption possibly overrating the performance of microprocessors with respect to actual applications due to overhead for scheduling and interrupt handling of the operation system (which is not required on an FPGA). We also compare our work to the very fast software implementation by [GT07] on an Intel Core2 system at 2.66 GHz but which uses the special Montgomery and non-standard curve over $\mathbb{F}_{2^{255}-19}$. The comparison of our results with figures from the literature is shown in Table 4.5.

4.6.3 Further Work

Although our results are promising, we encountered several performance issues with the implementation of elliptic curves on GPUs, particularly with respect to conditions in the formulas and the exception handling for special points. As a remedy, elliptic curves in Edwards and Hessian form feature highly homogeneous formulas which can significantly reduce the divergent part of the threads for point operations [Edw07, HCD07, Sma01]. The study by Bernstein *et al.* [BCC$^+$09] for ECM on GPUs demonstrated that elliptic curves in Edwards form can be very beneficial for high performance. The authors reported that 2414 point multiplications with 280 bit parameters are possible on the same NVIDIA 8800 GTS device using inverted twisted Edwards coordinates and Montgomery multiplication. Their underlying multiplier is based on a computational block with 256 threads which can compute eight 280-bit multipli-

Reference	Platform & Technique	Throughput [ModExps/s] and [PointMuls/s]					
		ModExp-1024	ModExp-1024, CRT	ModExp-2048	ModExp-2048, CRT	ECC PointMul-224	ECC PointMul-256
Our Design	NVIDIA 8800GTS, CIOS	**813.0**		**104.3**			
	NVIDIA 8800GTS, RNS	439.8		57.9			
	NVIDIA 8800GTS, NIST-224					1412.6	
[MPS07]	NVIDIA 7800GTX, RNS	175.4					
[CS07]	Sony Playstation 3		909.2		**401.4**		
[Men07]	Xilinx XC2VP30 FPGA	471.7	1724.1		235.8		440.5
[Suz07]	Xilinx XC4FX12 FPGA	584.8		79.4			
[NMSK01]	0.25μm CMOS, 221k GE	238.1		34.2			
[ECR07]	Intel Core2 2.13 GHz		1447.5		300.4	**1868.5**[a]	1494.8[a]
[GT07]	Intel Core2 2.66 GHz						**6900**[b]

[a]Performance for ECDSA including additional modular inversion and multiplication operation.
[b]Special elliptic curve in Montgomery form, non-compliant to ECC standardized by NIST.

Table 4.5: Comparison of our designs to results from literature.

cations at a time. Note however, that the curves standardized by ANSI and NIST *cannot* be transformed to Edwards or Hessian form, hence this technique can unfortunately not exploited when targeting industrial or governmental applications with requirements for standardized cryptography.

A last aspect considers the maturity of tools: the CUDA tool chain was an early version that does not necessarily create optimal compilation results. Hence, the results could probably improved with a later versions of the tool chain.

Part II

Cryptanalysis with Reconfigurable Hardware Clusters

Chapter 5

Cryptanalysis of DES-based Systems with Special Purpose Hardware

Cryptanalysis of ciphers usually involves massive computations. In the absence of mathematical breakthroughs to a cryptanalytical problem, a promising way for tackling these computations is to build special-purpose hardware which provide a better performance-cost ratio than off-the-shelf computers for most applications, such as the ones shown in [KPP+06b]. This chapter introduces a reconfigurable machine called COPACOBANA (Cost-Optimized Parallel Code Breaker) machine which is a high-performance cluster consisting of 120 low-cost FPGAs. We use COPACOBANA for various attacks on the Data Encryption Standard (DES), the former world-wide standard block cipher (FIPS 46-3), and some related DES-based products like One-Time-Password (OTP) generating crypto tokens.

Contents of this Chapter

5.1 Motivation

The security of symmetric and asymmetric ciphers is usually determined by the size of their security parameters, in particular the key-length. Hence, when designing a cryptosystem, these parameters need to be chosen according to the assumed computational capabilities of an attacker. Depending on the chosen security margin, many cryptosystems are potentially vulnerable to attacks when the attacker's computational power increases unexpectedly. In real life,

the limiting factor of an attacker is often financial resources. Thus, it is quite crucial from a cryptographic point of view to not only investigate the complexity of an attack, but also to study possibilities to lower the cost-performance ratio of attack hardware. For instance, a cost-performance improvement of an attack machine by a factor of 1,000 effectively reduces the key length of a symmetric cipher by roughly 10 bit (since $1000 \approx 2^{10}$).

Cryptanalysis of modern cryptographic algorithms involves massive and parallel computations, usually requiring more than 2^{40} operations. Many cryptanalytical schemes spend their computations in independent operations, which allows for a high degree of parallelism. Such parallel functionality can be realized by individual hardware blocks that can be operated simultaneously, improving the time complexity of the overall computation by a perfect linear factor. At this point, it should be remarked that the high non-recurring engineering costs for ASICs – with can often consume more than US$ 100,000 for large projects – have put most projects for building special-purpose hardware for cryptanalysis out of reach for commercial or research institutions. However, with the recent advent of low-cost FPGAs which host vast amounts of logic resources, special-purpose cryptanalytical machines have now become a possibility outside government agencies.

There are several approaches to building powerful computing clusters for cryptanalysis. For instance, 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 is variable due the dynamically changing number of contributors. A second natural approach could rely on utilizing supercomputers like IBM's BlueGene [Int08] or other commercial machines, e.g., from Cray or SGI. Unfortunately, supercomputers tend to provide sophisticated options for high-speed communication and large portions of distributed memory which are mostly not required for simple cryptanalytical number crunching. Unfortunately, the availability of these features increases the costs of these systems significantly, resulting in a non-optimal cost-performance ratio for cryptanalytical applications. With the improvements in FPGA technology, reconfigurable computing has emerged as a cost effective alternative for certain supercomputer applications.

In this chapter, we will employ a hardware architecture called Cost-Optimized Parallel Code Breaker (COPACOBANA) for advanced cryptanalytic applications which was originally introduced in [KPP+06b]. The platform is optimal for computational problems which can be split among multiple, independent nodes with low communication and memory requirements. COPACOBANA consists of up to 120 FPGA nodes which are connected by a shared bus providing an aggregate bandwidth of 1.6 Gbps on the backplane of the machine. COPACOBANA is not equipped with additional memory modules, but offers a limited number of RAM blocks inside each FPGA. Even though breaking modern ciphers like AES (with keys of 128/192/256 bits), full-size RSA (1024 bit or more) or elliptic curves (ECC with 160 bit or more) is out of reach with COPACOBANA, we can use the machine to gather data for extrapolating attacks with realistic security parameters in terms of financial costs and attack time. Equally importantly,

there are numerous legacy systems (and not-so-legacy systems such as the electronic passport) which are still operating with key lengths that can be tackled with COPACOBANA.

In this chapter, we will show how COPACOBANA can be used to break the Data Encryption Standard (DES) block cipher [Nat77] and other DES-related cryptosystems with (slightly) more advanced methods, like Time-Memory Tradeoffs (TMTO). Though DES was revoked as standard in 2004, it is still a popular choice for low-end security system as well as available in many legacy systems.

We identified a class of crypto tokens generating One-Time-Passwords (OTP) according to the ANSI X9.9 standard in which the DES encryption is still in use. Alarmingly, we are aware of online-banking systems in Europe, North and Central America which still distribute such tokens to users for authenticating their financial transactions[1]. We also present a setup to employ TMTO schemes for DES on COPACOBANA which is the preferable method when many keys for the same plaintext need to be identified. These schemes use precomputated tables to improve the duration of exhaustive key search attacks. We demonstrate how COPACOBANA can support the generation of the precomputations for attacking the DES block cipher.

Besides DES breaking, cryptanalysis on asymmetric ciphers can also be supported by COPACOBANA, e.g., for solving the Elliptic Curve Discrete Logarithm Problem [GPP07a] which is known as the fundamental primitive for cryptosystems based on elliptic curves. The corresponding implementation is discussed in Chapter 6. Moreover, we also adapted the Elliptic Curve Method for integer factorization for use with Xilinx Virtex-4 FPGAs on a variant of the original COPACOBANA cluster. This work is presented in Chapter 7. There is also further work with COPACOBANA which are not in the scope of this thesis. For example, an attack on a legacy hard disk encryption (*Norton Diskreet*) [KPP+06a], attacks on the GSM A5/1 stream cipher [GNR08] and the recent Machine Readable Travel Documents (*ePassport*) [LKLRP07] are further examples of cryptanalytic applications which also make use of the COPACOBANA cluster.

5.2 Previous Work

Since the invention of the computer, a continuous effort has been taken to build clusters providing the recent maximum of computing power. For our purpose, we focus on the cost-efficient COPACOBANA system instead of reviewing all recent variants of such high-performance cluster or supercomputer for investments of several millions of dollars. Thus, we now shortly survey the history of breaking the most popular block cipher of the last decades – the Data Encryption Standard (DES). After that, we will come up with an evaluation of the security margin provided by DES nowadays with respect to available machines like COPACOBANA.

[1] Since we do not want to support hacking of bank accounts, we will not give further details here.

Although the DES was reaffirmed for use in (US government) security systems several times until 1999, the worries about the inherent threat of its short key space was already raised in 1977 when it was first proposed. The first estimates were proposed by Diffie and Hellman [DH77] for a brute force machine that could find the key within a day at a cost of US$ 20 million. Some year after that, a first detailed hardware design description for a brute force attack was presented by Michael Wiener at the rump session of CRYPTO'93, a printed version is available in [Wie96]. It was estimated that the machine could be built for less than a million US dollars. The proposed machine consists of $57,000$ DES chips that could recover a key every three and half hours. In 1997, a detailed cost estimate for three different approaches for DES key search, distributed computing, FPGAs and custom ASIC designs, was presented by Blaze et al. [BDR+96]. In 1998, the Electronic Frontier Foundation (EFF) finally built a DES hardware cracker called *Deep Crack* which could perform an exhaustive key search 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. To our knowledge, the latest step in the history of DES brute-force attacks took place in 2006, when the Cost Optimal Parallel Code Breaker (COPACOBANA) was built for less than US$ 10,000 [KPP+06b]. COPACOBANA is capable of breaking DES in less than one week on average. We would like to note that software-only attacks against DES still take more than 1,000 PC-years (based on Intel Pentium-4@3GHz) in worst case.

Most of these attacks assume that at least one complete plaintext-ciphertext pair is given. We will see that crypto tokens for bank applications (compliant to ANSI X9.9) typically do not provide such inputs, so that a smarter attack must be chosen to tackle this kind of systems. There are some theoretical contributions by Coppersmith et al. [CKM00] as well as by Preneel and van Oorschot [PO96] considering the theoretical security of DES-based authentication methods (DES-MAC). But to our best knowledge an attack on an ANSI X9.9-based crypto system has not been proposed (or demonstrated) yet. Minor parts of the work presented in this chapter were done in collaboration with Andy Rupp and Martin Novotny and were also published in [GKN+08].

5.3 Mathematical Background

The impracticability of an exhaustive key search is an essential requirement for the security of symmetric ciphers, i.e., it should be infeasible for an attacker with a constrained set of resources to test each possible key candidate in order to determine the correct key used in the cryptosystem. The constraints (and implicitly, costs) of such attack is determined based on the available technology and expected future developments. Usually, the key size is chosen such that it allows for efficient implementation of the cryptosystem but that brute force attacks are only be successful with a negligible probability.

A more mathematical description of such a problem for block ciphers (and stream ciphers) is as follows: Let $e : P \times K \rightarrow C$ be a (one-way) function with the plaintext domain P and

ciphertext domain C of same size $|P| = |C|$. The set K denotes the key space of size $|K| = n$. In case of a block cipher, we define e to be a bijective encryption function which is used to encrypt a plaintext $p \in P$ under a key $k \in K$ into a ciphertext $c \in C$:

$$\begin{aligned} e : (P \times K) &\rightarrow C \\ e_k(p) &\mapsto c \end{aligned}$$

Given an image $c \in C$, the challenge for an attacker is to find a preimage of c, i.e., some plaintext $p \in P$ and key $k \in K$ so that $e_k(p) = c$. An attacker often knows at least one corresponding combination of plaintext p and ciphertext c (*known-plaintext attack*) and can use this to uniquely identify key k – however usually at a large computational cost.

By using a cryptanalytic Time-Memory Tradeoff method, one tries to find a compromise between the two well-known extreme approaches to recover the key k, i.e., either performing live, exhaustive key searches or precomputing exhaustive tables with all possible key combinations for a predefined tuple (p, c). A TMTO offers a way to reasonably reduce the actual search complexity (by doing some kind of precomputation) while keeping the amount of precomputed data reasonably low, whereas reasonably has to be defined more precisely. It depends on the actual attack scenario (e.g., real-time attack), the function e and the available resources for the precomputation and online (search) phase.

Existing TMTO methods [Hel80, Den82, Oec03] share the natural property that in order to achieve a significant success rate much precomputation effort is required. Since performing this task on PCs is usually too costly or time-consuming, cheap special-purpose hardware with massive computational power – like COPACOBANA – is required. In [SRQL03a] an FPGA design for an attack on a 40-bit DES variant using Rivest's TMTO method was proposed. In [MBPV06] a hardware architecture for UNIX password cracking based on Oechslin's method was presented. However, to the best of our knowledge, nobody has done a complete TMTO precomputation for *full* 56-bit DES so far.

The next sections give a brief overview of cryptanalytic time-memory tradeoff methods.

5.3.1 Hellman's Time-Memory Tradeoff Method for Cryptanalysis

The original Time-Memory Tradeoff Method, published in 1980 by Hellman [Hel80], tries to precompute all possible key-ciphertext pairs in advance by encrypting p with all n possible keys. However, to reduce memory requirements, these pairs are organized in several *chains* of a fixed length. The chains are generated deterministically and are uniquely identified by their respective start and end points. In this way, it suffices to save its start and end point to restore a chain later on. In the online phase of the attack, one then simply needs to identify and reconstruct the right chain containing the given ciphertext to get the desired key. The details of the two phases are described in the following.

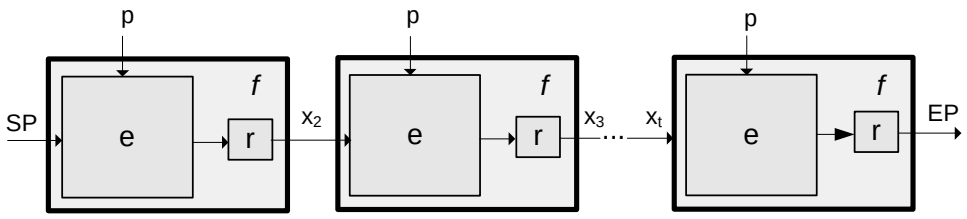


Figure 5.1: Chain generation according to Hellman's TMTO.

Precomputation phase.

Assume m as the number of concurrent computations used for precomputing. The first m different keys are chosen serving as start points SP of the chains. To generate a chain, one first computes $e_{\mathsf{SP}}(p)$ resulting in some ciphertext c (see Figure 5.1). In order to continue the chain, c is used to generate a new key. To do so, a so-called *reduction and re-randomization function* r is applied reducing the bit length of c to the bit length of a key for the encryption function e (if necessary) and performing a re-randomization of the output. By means of r, we can continue the chain by computing $r(e_{\mathsf{SP}}(p)) = x_2$, using the resulting key x_2 to compute $r(e_{x_2}(p)) = x_3$ and so on. The composition of e and r is called *step-function* f. After t applications of f the chain computation stops and we take the last output as the end point EP of the chain. The pair $(\mathsf{SP}, \mathsf{EP})$ is stored in a table sorted by end points. The number of distinct keys contained in a table divided by n is called the *coverage* of a table. Unfortunately, the occurrence of a key in a table is not necessarily unique because there is a chance that two chains collide and merge or that a chain runs into a loop. This is due to the non-injective function r mapping the space of ciphertexts to the key space. (which is often smaller, e.g., in the case of DES). Each merge or loop reduces the fraction of distinct keys contained in a table and thus the coverage (if m is fixed). Since the probability of merges increases with the size of a table, at a certain point we cannot significantly improve the coverage by simply adding more and more chains. Hellman calculated that this point is somewhere near $n^{\frac{2}{3}}$ for a single table. To cope with this problem, he suggested to generate multiple tables each associated with a different reduction function. In this way even if two chains from different tables collide, they will not merge because different functions are applied to the shared value in the next step.

Online phase.

In the online phase a ciphertext c' is given that is assumed to be the result of the encryption of p using some key k. We try to retrieve k from the precomputed tables in the following way: to find out if k is covered by a specific table, we compute a chain up to a length of t starting with $r(c')$ and compare the intermediate points with the end points in the table. More precisely, we first check if $r(c')$ is contained in the table. If not, we compute $f(r(c'))$ and look for a match, then we repeat this for $f(f(r(c')))$ and so on. If a match occurs after the i-th application of f for a pair $(\mathsf{SP}, \mathsf{EP})$, then $f^{t-i-1}(\mathsf{SP}) = x_{t-i}$ is a key candidate. This candidate needs to be

checked, by verifying $e_{x_{t-i}}(p) = c'$, and, if it is valid, the online phase ends. If it is not valid, a *false alarm* has occurred and the procedure continues while the chain has a length smaller than $t+1$. If no valid key is found in this table we repeat the same procedure for another table (and thus another r and f).

5.3.2 Alternative Time-Memory Tradoff Methods

For Hellman's approach two variants were proposed for improved performance with parallel computations and greater key coverage.

Distinguished Points.

In most attack setups for Hellman's TMTO, the time required to complete the online phase is dominated by the high number of table accesses. Random accesses to disk can be many orders of magnitude slower than the evaluation of f. The Distinguished Point (DP) method, introduced by Rivest [Den82] in 1982, addresses this problem. A DP is a key that fulfills a certain simple criterion (e.g., the first 20 bits are zero) which is usually given as a mask of length d. Rivest's idea was to admit only DPs as end points of a chain. For the precomputation phase this means that a chain is computed until a DP or a maximal chain length $t_{max}+1$ is reached. Only chains of length at most $t_{max}+1$ ending in a DP are stored. Using DPs, merging and looping chains can also be detected and are then discarded. In the online phase, the table does not need to be accessed after every application of f but only for the first occurring DP. If we have no match for this DP we can proceed with the next table.

Rainbow Tables.

Rainbow tables were introduced by Oechslin [Oec03] in 2003. He suggested not to use the same r when generating a chain for a single table but a (fixed) sequence $r_1, \ldots, r_t$ of different reduction functions. More precisely, due to the different reduction functions we get t different step-functions $f_1, \ldots, f_t$ that are applied one after another in order to create a chain of length $t+1$. The advantage of this approach is that the effect of chain collisions is reduced: while the collision of two chains inevitably leads to a merge of these chains in a Hellman table, a merge only happens if the shared value appears at the *same* position in both chains for the rainbow method . Otherwise they share only this single value. Thus, a merge of two chains in a rainbow table is not likely to occur. Furthermore, loops are completely prevented. Hence, regarding a space efficient coverage, these characteristics allow to put many more chains into a rainbow table than into a Hellman table. This in turn significantly reduces the total number of tables needed in order to achieve a certain coverage. Since fewer rainbow tables must be searched in the online phase (what is, however, slightly more complex) a lower number of calculations and table accesses is required compared to Hellman's method. To check for a key in a rainbow table, we first compute $r_t(c')$ and compare it to the end points, then we repeat this for $f_t(r_{t-1}(c'))$, $f_t(f_{t-1}(r_{t-2}(c')))$, etc. Moreover, compared to the distinguished point method the number of false alarms and the corresponding extra work is reduced.

5.4 COPACOBANA – A Reconfigurable Hardware Cluster

In the following section, we present the FPGA-based COPACOBANA hardware cluster. COPACOBANA was designed to provide a significant amount of computing resources to applications with only a minor demand on memory and communications. The majority of other FPGA-based computing clusters or supercomputers, however, focus on data-oriented applications requiring large amounts of memory and widely-dimensioned bandwidth. Examples for such universal supercomputing systems are Cray's XD1 system [Inc08a] as well as the SGI RASC technology [Inc08b] that also include reconfigurable devices in their design. Unfortunately, such platforms are inappropriate for most tasks in cryptanalysis due to their high costs and the related non-optimal cost-performance ratio. Here, to the best of our knowledge, COPACOBANA is the only low-cost alternative to commercial supercomputers offering no nameable amount of memory but a significant amount of computing resources for about US$ 10,000.

The hardware architecture of a Cost-Optimized Parallel Code Breaker (COPACOBANA) was developed according to the following design criteria [KPP+06b]: First, we assume that computationally costly operations can be parallelized. Secondly, all concurrent instances have only a very limited requirements to communicate with each other. Thirdly, the demand for data transfers between host and nodes is low due to the fact that computations heavily dominate communication requirements. Ideally, (low-speed) communication between the hardware and a host computer is only required for initialization and the transfer of results. Hence, a single conventional (low-cost) PC should be sufficient to transfer required data packets to and from the hardware, e.g., connected by a standardized interface. Fourthly, all presented algorithms and their corresponding hardware nodes demand very little local memory which can be provided by the on-chip RAM modules of an FGPA.

Since the cryptanalytical applications demand for plenty of computing power, we installed a total of 120 FPGA devices on the COPACOBANA cluster[2]. Building a system of comparable dimension with commercially available FPGA boards is certainly feasible but rather expensive. By stripping down functionality to the bare minimum and producing the hardware ourselves, we are able to achieve with COPACOBANA an optimal cost-performance for code breaking. For a modular and maintainable architecture, we designed small FPGA modules which can be dynamically plugged into a backplane. Each of these modules in DIMM form factor hosts 6 low-cost Xilinx Spartan3-XC3S1000 FPGA which are directly connected to a common 64-bit data bus on board. The data bus of the module is interfaced to the global data bus on a backplane. While disconnected from the global bus, the FPGAs on the same module can communicate via the local 64-bit data bus. Additionally, control signals are run over a separate 16-bit address bus. Figure 5.2 gives a detailed overview of the architecture of COPACOBANA. For simplicity, a single master bus was selected to avoid interrupt handling or bus arbitration. Hence, if the communication scheduling of an application is unknown in advance, the bus master need to poll the FPGAs.

[2]This number was determined by the size of the FPGA modules (DIMM form factor) and the maximum bus length.

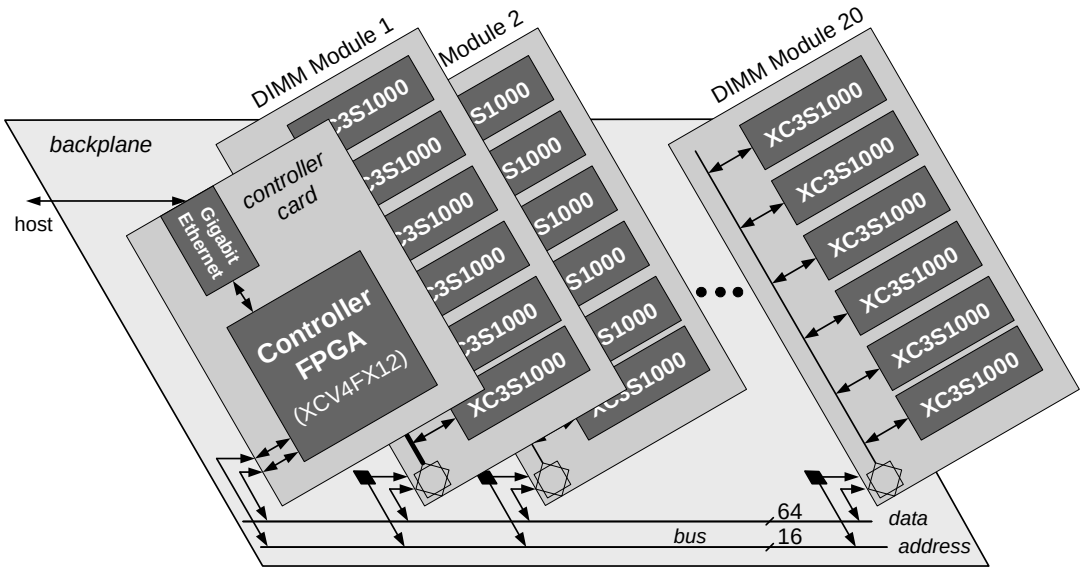


Figure 5.2: Schematic architecture of the COPACOBANA cluster.

The top level entity of COPACOBANA is a host-PC which is used to initialize and control the FPGAs, as well as for the accumulation of results. Programming can be done simultaneously for all or a specific subset of FPGAs. Data transfer between FPGAs and a host-PC is accomplished by a dedicated control interface. The controller has also been designed as a slot-in module so that COPACOBANA can be connected to a computer either via a USB or Ethernet controller card. A software library on the host-PC provides low-level functions that allow for addressing individual FPGAs, storing and reading FPGA-specific application data. With this approach, we can easily attach more than one COPACOBANA device to a single host-PC.

5.5 Exhaustive Key Search on DES

The Data Encryption Standard (DES) with a 56-bit key size was chosen as the first commercial cryptographic standard by NIST in 1977 [Nat77]. A key size of 56-bits was considered to be a good choice considering the huge development costs for computing power in the late 70's, that made a search over all the possible 2^{56} keys appear impractical.

Since DES was designed to be extremely efficient in terms of area and speed for hardware, an FPGA implementation of DES can be orders of magnitude faster than an implementation on a conventional PC at much lower costs [KPP+06b]. This allows a hardware based engine for a DES key search to be much faster and efficient compared to a software based approach.

Our core component is an improved version of the DES engine of the Université Catholique de Louvain's Crypto Group [RSQL03] based on 21 pipeline steps. Our design can test one key per clock cycle and engine. On the COPACOBANA, we can fit four such DES engines inside

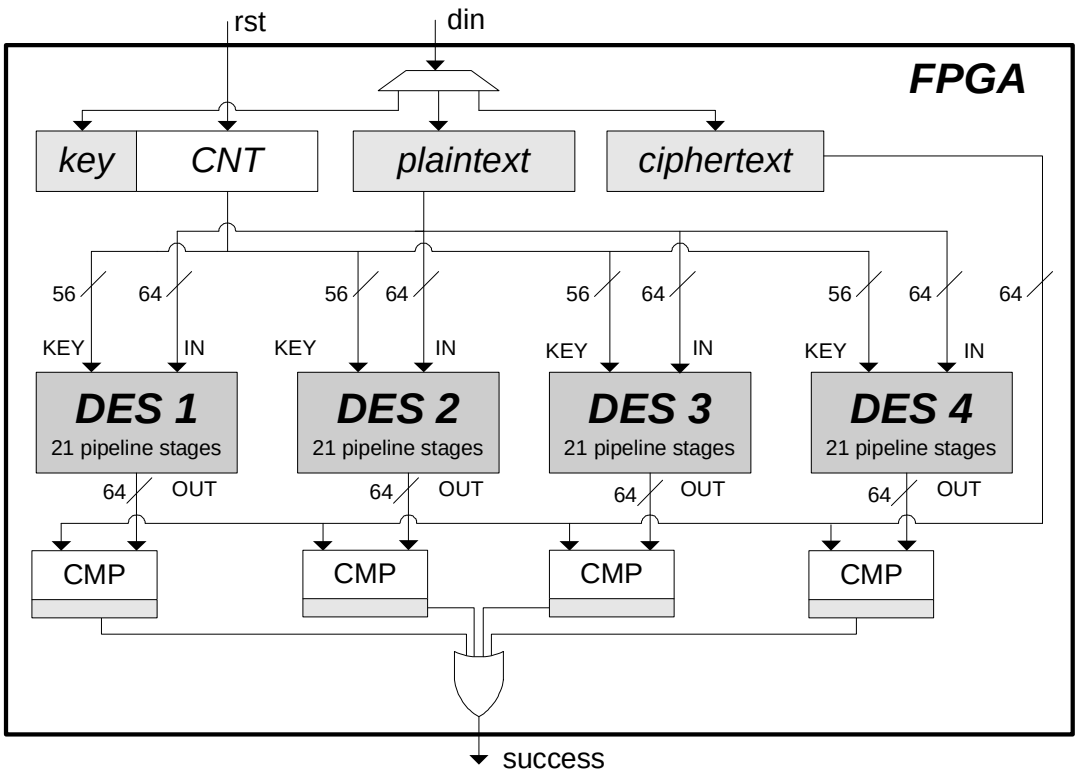


Figure 5.3: Architecture for exhaustive key search with four DES key search units.

a single FPGA, which allows for sharing plaintext-ciphertext input pairs and the key space as shown in Figure 5.3.

Since our first implementation, as presented in [KPP+06b], we were able to tweak our design for increased performance by the use of additional pipelined comparators and improved control logic. We now can operate each of the FPGAs at an increased clock rate of 136 MHz so that a gain in performance by 36% is achieved, compared to [KPP+06b]. Consequently, a partial key space of 2^{42} keys can completely be checked in $2^{40} \times 7.35\,\text{ns}$ by a single FPGA, which is approximately 135 minutes. Since COPACOBANA hosts 120 of these low-cost FPGAs, the key search machine can check $4 \times 120 = 480$ keys every 7.35 ns, i.e., 65.28 billion keys per second. To find the correct key, COPACOBANA has to search through an average of 2^{55} different keys. Thus, COPACOBANA can find the right key in approximately $T = 6.4$ days on average. Since more than one COPACOBANA can be attached to a single host and the key space can be shared among all machines, the search time then reduces to $\frac{T}{\ell}$, where ℓ denotes the number of machines.

Method	Chain Length	# SP	# Tables	# Bits per EP
Hellman	$t = 2^{19.2}$	$2^{16.7}$	2^{21}	73
Rainbow	$t = 2^{19.5}$	2^{35}	5	91
DP	$t_{min} = 2^{18}$, $t_{max} = 2^{20}$, $d = 19$	2^{18}	2^{21}	55

Table 5.1: Empirical TMTO parameters for optimal performance with COPACOBANA.

5.6 Time-Memory Tradeoff Attacks on DES

In this section, we will employ COPACOBANA for accelerating the precompution and online phase of a TMTO attack on DES. In such a scenario, primarily the hardware limitations of COPACOBANA with respect of communication bandwidth need to be considered. Since COPACOBANA does not allow for installation of directly attached storage, all TMTO tables must be managed by the connected host-PC. The recent Ethernet-based interface between COPACOBANA an host-PC provides a communication bit rate of $24 \cdot 10^6 \approx 2^{24.5}$ bit per second[3]. Compared to the number of possible DES encryptions per second, the bottleneck of the COPACOBANA is the data throughput to transfer $(\mathsf{SP}, \mathsf{EP})$ tuples from the FPGAs to the host. To address the constraint of limited bandwidth, we have determined a minimum rate of $2^{11.4}b$ computations to be run in sequence until a data transfer can be initiated, where b denotes the aggregate length of a tuple $(\mathsf{SP}, \mathsf{EP})$. For practical reasons, we have limited the disk space for the TMTO tables to a maximum of two terabyte and the required success rate to 80%. Based on experiments, we found following parameters for chain length, number of tables and start points to satisfy the given constraints.

To reduce data transfers to a bare minimum, we use the first m integers as start points SP and assign fixed subintervals of $[0, m]$ to each FPGA. In this way each SP can be stored with only $log_2(m)$ bits. Optionally, the host-PC can even track the sequence of start points for each individual FPGA so that data transfers of SPs can be omitted completely. Then, only the end points EP must be transmitted to the host-PC and matched with the corresponding SP in software.

For the DP method, we introduce a minimum chain length t_{min} to ensure that the generated data traffic from tuples $(\mathsf{SP}, \mathsf{EP})$ always complies with the available bandwidth on the COPACOBANA. More precisely, each DP chain leading to a total chain length less than $t_{min} + 1$ is discarded and not transfered to the host[4]. In this case, the storage of end points for the DP method can be limited to the remaining $56 - d$ bits not covered by the DP-criterion.

Table 5.6 presents our worst case expectations concerning success rate (SR), disk usage (DU), the duration of the precomputation phase (PT) on COPACOBANA as well as the number of

[3] At the time of writing, our current Gigabit-Ethernet controller does not yet provide a high-performance communication interface due to immaturity. After eliminating this bottleneck, currently existing constraints on the bandwidth will relax and assumed parameters are subject to change.

[4] Note that with tracking of start points in the host software this must be indicated to the host-PC to increment the SP counter accordingly.

Method	SR	DU	PT (COPA)	OT
Hellman	0.80	1897 GB	24 days	$2^{40.2}$ TA + $2^{40.2}$ C
DP	0.80	1690 GB	95 days	2^{21} TA + $2^{39.7}$ C
Rainbow	0.80	1820 GB	23 days	$2^{21.8}$ TA + $2^{40.3}$ C

Table 5.2: Expected runtimes and memory requirements for TMTOs.

table accesses (TA) and calculations (C) during the online phase (OT). Note that these figures for use with COPACOBANA are based on estimations given in [Hel80, Oec03, SRQL03a] (false alarms are neglected) and the given constraints mentioned above. Note further that for this initial extrapolation we have used the implementation of our exhaustive key search unit presented in Section 5.5. According to our findings, precomputations for the DP method on a single COPACOBANA take roughly four times longer compared to Hellman's and Oechslin's method based on the given constraints. Contrary, the subsequent online attack has the lowest complexity for the distinguished point method. Considering a TMTO scenario to use the COPACOBANA for precomputation only (implying that the online attack is performed by a PC), the rainbow table method can be assumed to provide best performance. When using COPACOBANA as well for precomputation *and* online phase, there is a strong indicator to select distinguished points as the method of choice: for the DP method, we can assume the frequency of table accesses to follow a uniform distribution, hence, we expect balanced bandwidth requirements over time. With respect to the online phase using rainbow tables, the computation trails are short in the beginning but increment in length over time. This results in significant congestion on COPACOBANA's communication interface since a large number of table lookups are required in the beginning of the online phase. Therefore, a scenario running both the precomputation and the online phase on COPACOBANA should be based on the DP method since this method is most promising with respect to the restrictions of the machine.

We have implemented the precomputation phase for generating DES rainbow tables on COPACOBANA. For this implementation, we have developed another DES core which operates with 16 pipeline stages only[5]. With 4 parallel DES units and 16 pipeline stages each we can run 64 chain computations in parallel per FPGA. Figure 5.4 graphically presents our architectures for generating rainbow tables in further detail. On the given Spartan-3 devices, our entire implementation including I/O and control logic consumes 7571 out of 7680 (98%) available slices of each FPGA and runs at a maximum clock frequency of 96MHz. Based on this figures, a single COPACOBANA is able to compute more than 46 billion iterations of the step function f per second. We are currently optimizing the I/O logic to support concurrent trail computations *and* data transfers to eliminate idle times of the DES cores during data transmission. With this improvement of our design, we can estimate the actual duration of the precomputation phase

[5]Recall that the DES implementation from Section 5.5 uses 21 instead of 16 pipeline stages. A 16-staged implementation allows for simpler addressing schemes when selecting a result from a specific pipeline position.

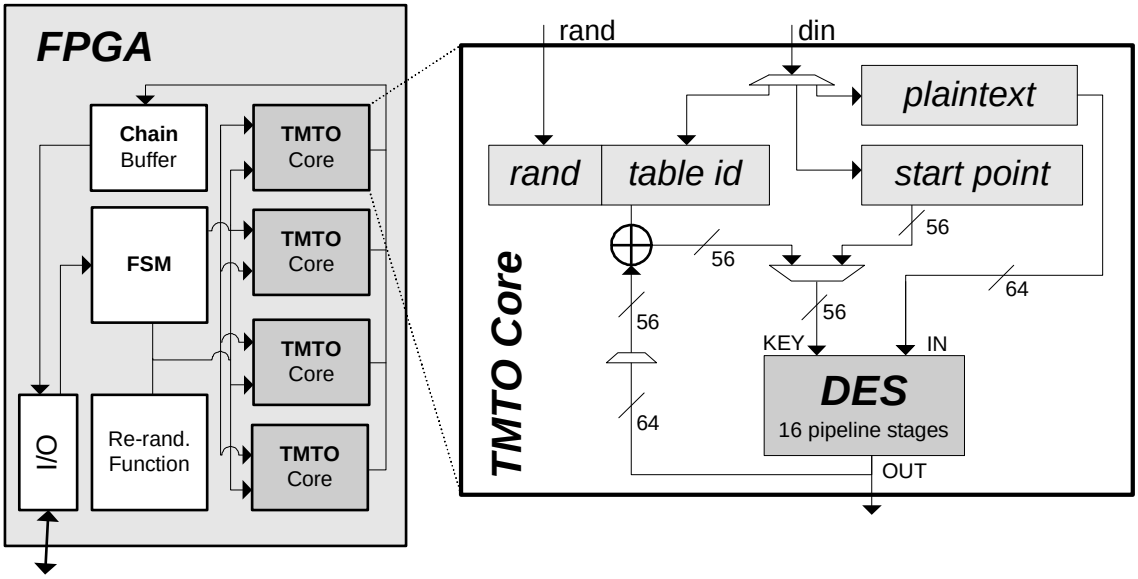


Figure 5.4: TMTO implementation to generate DES rainbow tables.

for generating the rainbow tables to last slightly less than 32 days. With a high-performance controller for COPACOBANA becoming available, we expect to verify our results practically and complete the generation of full precomputation tables for the DES.

5.7 Extracting Secrets from DES-based Crypto Tokens

In this section, we employ COPACOBANA for an attack on cryptographic tokens which are used for user authentication and identification according to FIPS 113 or ANSI X9.9 in real-world applications. Their authentication method is based on One-Time Passwords (OTP) generated using the DES algorithm. Unfortunately, such devices are still used in many security relevant applications[6]. Hence, the attack presented in the following still has an impact on affected online banking systems used worldwide.

5.7.1 Basics of Token Based Data Authentication

We will now describe a OTP token-based data protocol according to FIPS 113 or ANSI X9.9 which is used for authentication in some real-world online-banking systems. Please note that we assume that OTP tokens have a fixed, securely integrated static key inside and do not use additional entropy sources like time or events for computing the passwords. Indeed, there are tokens available which generate new passwords after a dedicated time interval (e.g., products like the RSA SecurID solution [RSA07]) but those will not be the focus of this chapter. These

[6]We are aware of online-banking systems in some places of the world still relying on ANSI X9.9 based tokens for authorization of financial transactions. We prefer not to give any details at this point.

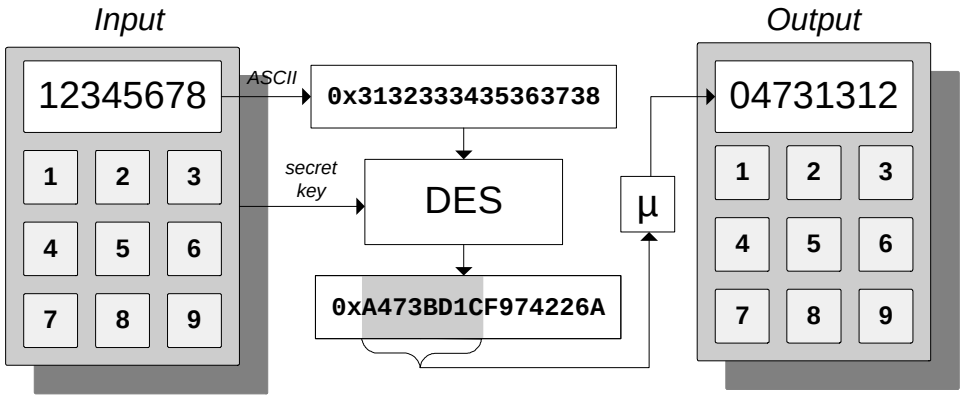


Figure 5.5: Principle of response generation with ANSI X9.9-based crypto tokens.

type of tokens require additional assumptions concerning the unknown plaintext and thus are harder to attack. More precisely, our contribution assumes fixed-key OTP tokens which can be used in combination with a challenge-response protocol. In such protocols, a decimal-digit challenge is manually entered into the token via an integrated keypad. The token in turn computes the corresponding response according to the ANSI X9.9 standard. Tokens implementing this standardized authentication scheme (incorporating ANSI 3.92 DES encryption) have a often a fixed size LCD allowing for displaying 8 decimal digits for input and output.

After the user has typed in eight decimal digits as input (challenge), the value is converted to binary representation using standard ASCII code notation according to the ANSI X9.9 standard. For instance, the typed number "12345678" is converted into the 64-bit challenge value in hexadecimal representation

$$c = (\texttt{0x31}, \texttt{0x32}, \texttt{0x33}, \texttt{0x34}, \texttt{0x35}, \texttt{0x36}, \texttt{0x37}, \texttt{0x38}).$$

After recoding, c is used as plaintext to the DES encryption function $r = e_k(c)$ with the static key k stored securely in the token. The output of the encryption function is the 64-bit ciphertext $r = (r_1, r_0)$ where each r_i denotes a 32 bit word to be transformed using a mapping μ to fit the 8-digit display of the token. The mapping μ takes the 8 hexadecimal digits of r_1 (32 bits) of the DES encryption as input, and converts each digit individually from hexadecimal (binary) notation to decimal representation. Let $H = \{\texttt{0}, \ldots, \texttt{9}, \texttt{A}, \ldots, \texttt{F}\}$ and $D = \{\texttt{0}, \ldots, \texttt{9}\}$ be the alphabets of hexadecimal and decimal digits, respectively. Then μ is defined as:

$$\mu : H \quad \rightarrow \quad D : \{\texttt{0}_H \mapsto \texttt{0}_D; \ldots; \texttt{9}_H \mapsto \texttt{9}_D; \texttt{A}_H \mapsto \texttt{0}_D; \ldots; \texttt{F}_H \mapsto \texttt{5}_D\}$$

Hence, the output after the mapping μ is an 8 decimal digit value which is displayed on the LCD of the token. Figure 5.5 shows how the response is generated on the token according to a given challenge. In several countries, this authentication method is used in banking applications whenever a customer needs to authenticate financial transactions. For this, each user of such

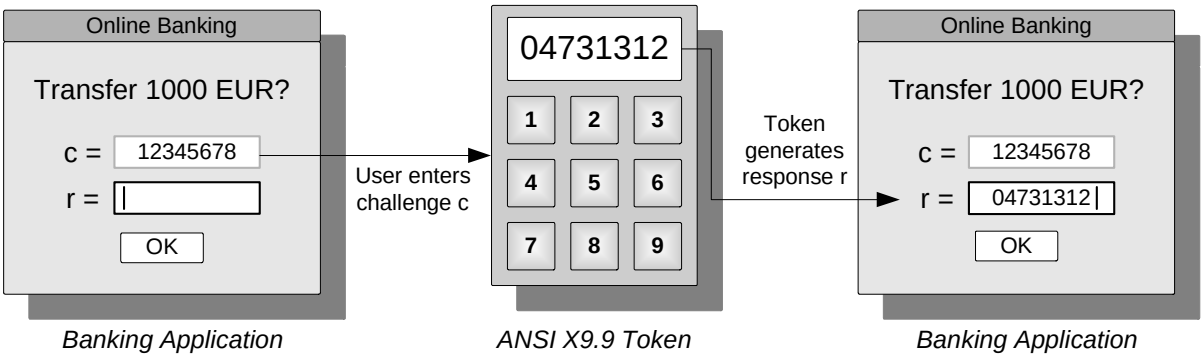


Figure 5.6: Token-based challenge-response protocol for online-banking.

an online-banking system owns a personal token used to respond to challenges presented by the banking system to authorize every security-critical operation. In this context for example, a security-critical operation can be the login to the banking system as well as the authorization of a money transfer. Figure 5.6 depicts a token-based challenge-response protocol interaction with an online-banking system from a user's perspective.

The central role in such a security-related application makes the secret token an interesting target for an attack.

5.7.2 Cryptanalysis of the ANSI X9.9-based Challenge-Response Authentication

With the knowledge of how an authenticator is computed in the challenge-response protocol, we will continue with identifying weaknesses to attack this authentication scheme. Firstly, ANSI X9.9 relies on the DES algorithm for which we built a low-cost special-purpose hardware machine and can perform an exhaustive key search in under a week. Secondly, the output r of the DES encryption is only slightly modified. Note that a more complex scrambling with additional dynamic input, like hash functions with salt, would make the attack considerably more complex or even impossible. The output r is only truncated to 32-bit and modified using the mapping μ to convert c_1 from hexadecimal to decimal notation. Due to the truncation to 32 bits, we need to acquire knowledge of at least two plaintext-ciphertext pairs when mounting an exhaustive key search to return a single key candidate only. The digit conversion μ additionally reduces the information leaked by a single pair of plaintext-ciphertext which is addressed by Observation 5.1.

Observation 5.1 *Let $D = \{0, \ldots, 9\}$ be the alphabet of decimal digits. With a single challenge-response pair (c, r) of an ANSI X9.9-based authentication scheme where $c, r \in D^8$, on average 26 bits of a DES key can be determined (24 bits in the worst case, 32 bits in the best case).*

Since only 32 bits of the output c for a given challenge c are exposed, this is a trivial upper bound for the information leakage for a single pair. Assuming the DES encryption function to be a pseudo-random function with appropriate statistical properties, the 32 most significant bits of c

form 8 hexadecimal digits uniformly distributed over $H^8 = \{0, \ldots, 9, A, \ldots, 8\}^8$. The surjective mapping μ has the domain $F = \{0, \ldots, 9\}$ of which $T = \{0, \ldots, 5\}$ are doubly assigned. Hence, we know that $\Delta = F \backslash T = \{6, \ldots, 9\}$ are four fixed points which directly correspond to output digits of c yielding four bit of key information (I). The six remaining decimal digits $\Omega = F \cap T$ can have two potential origins allowing for a potential deviation of one bit (II). According to a uniform distribution of the 8 hexadecimal output digits, the probability that (I) is given for an arbitrary digit i of c is $Pr(i \in \Delta) = 1/4$. Thus, on average we can expect 2 out of 8 hexadecimal digits of c to be in Δ revealing four bits of the key whereas the remaining 6 digits introduce a possible variance of one unknown bit per digit. Averaged, this leads to knowledge of $R = 2 \cdot 4 + 6 \cdot 3 = 26$ bits of DES key material. Obviously, the best case with all 8 digits in Δ and worst case with no digits out of the set Δ provide 32 and 24 key bits, respectively.

According to Observation 5.1, we can develop two distinguished attacks based on the knowledge of two and three known challenge-response pairs:

Observation 5.2 *Given two known challenge-response pairs (c_i, r_i) for $i = \{0, 1\}$ of the ANSI X9.9 authentication scheme, an exhaustive key search using both pairs will reveal $2^4 = 16$ potential key candidates on average (256 candidates in the worst case, in the best case the actual key is returned).*

Assuming independence of two different encrypted blocks related to the same key in block ciphers, we can use accumulated results from Observation 5.2 for key determination using multiple pairs (p_i, c_i). Hence, on average we can expect to determine 52 bits of the key where each c_i has 2 digits from the set Δ. Given a full DES key of 56 bit size results in 2^4 possible variations for key candidates. Having at least 4 digits from Δ for each c_i, this will lead to the best case resulting in a single key candidate. In the worst case and with no Δ digits in any c_i, we will end up with 48 bits of determined key material and $2^8 = 256$ possible remaining key candidates. As a consequence, the number of potential key candidates is directly dependent on how many digits of a c_i are fixed points and out of the set Δ.

Observation 5.3 *Given three known challenge-response pairs of the ANSI X9.9 authentication scheme, an exhaustive key search based on this information will uniquely reveal the DES key.*

This directly follows from Observation 5.2. For this attack, $3 \cdot 24 = 72 > 56$ bits of key material can directly determined (even in the worst case) resulting in the correct key to be definitely identified.

5.7.3 Possible Attack Scenarios on Banking Systems

With these fundamental observations at hand, we can begin to develop two attack variants for two and three plaintext-ciphertext pairs. Since we need only few pairs of information, an attack is feasible in a real-world scenario. For instance, if we consider a phishing attack on an online-banking system, we can easily imagine that two or three (faked) challenges are presented to the user, who is likely to respond with the appropriate values generated by his token. Alternatively,

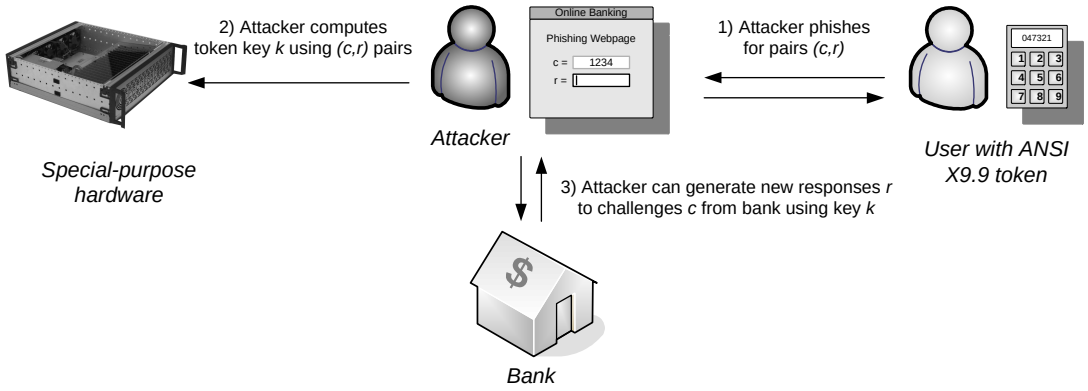


Figure 5.7: Attack scenario for token-based banking applications using phishing techniques.

spying techniques, for example based on malicious software like key-loggers or hidden cameras, can be used to observe the user while responding to a challenge. Note that the freshness of these values do not play a role since we use the information only for computing the secret key and not for an unauthorized login attempt. Figure 5.7 shows a possible attack scenario on ANSI X9.9 tokens and associated banking applications based on phishing of challenge-response pairs (c, r). With at least two pairs of challenge-response data, we can perform an exhaustive key search on the DES key space implementing the specific features of ANSI X9.9 authentication. To cope with the DES key space of 2^{56} potential key candidates we will propose an implementation based on dedicated special-purpose hardware. In case that three pairs of challenge-responses pairs are given, we are definitely able to uniquely determine the key of the secret token using a single exhaustive key search. When only two pairs (c_i, r_i) are available to the attacker, then it is likely that several potential key candidates are returned from the key search (cf. Observation 5.2). With 16 potential solutions on average, the attacker can attempt to guess the right solution by trial-and-error. Since most banking systems allow the user to enter up to three erroneous responds to a challenge in a row, two key candidates can be tried by the attacker at a time. Then, after a period of inactivity, the authorized user has probably logged into the banking application that resets the error counter and allows the attacker to start another trial session with further key candidates. On average, the attacker can expect to be successful after about four trial and error sessions, testing 8 out of the 16 keys from the candidate list. Hence, an attack on an ANSI X9.9-based token is very likely to be successful even with knowledge of only two given challenge-response pairs.

5.7.4 Implementing the Token Attack on COPACOBANA

As before, the main goal of our hardware design is a key search of the token to be done in a highly parallelized fashion by partitioning the key space among the available FPGAs on the COPACOBANA. This requires hardly any interprocess communication, as each of the DES engines can search for the right key within its allocated key subspace.

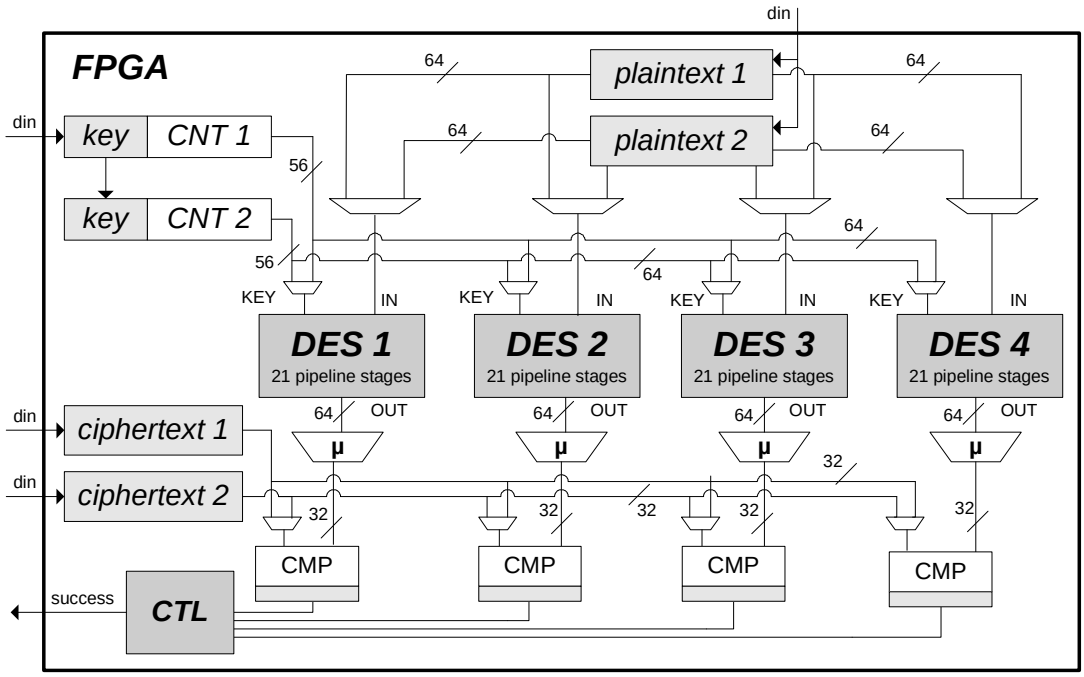


Figure 5.8: Four ANSI X9.9 key search units based on fully pipelined DES cores in a Xilinx Spartan-3 FPGA.

Within the FPGAs, we use again a slightly modified version of the highly pipelined DES implementation of the Université Catholique de Louvain's Crypto Group [RSQL03], which computes one encryption per clock per engine. As with the brute-force attack, we can fit four such DES engines inside a single FPGA, and therefore allow for sharing of control circuitry and the key space as shown in Figure 5.8. The FPGA architecture comprises two 64-bit *plaintext* registers for the challenges and two 32-bit *ciphertext* registers for storing the corresponding responses which can be acquired from the OTP-token. The key space to be searched is allocated to each chip as the most-significant 14-bits of the key which is stored in the *Key* register. The counter (*CNT 1*) is used to run through the least significant 40 bits of the key. The remaining two bits of the 56-bit key for each of the DES engines are hardwired and dedicated to each of them. Thus, for every such FPGA, a task is assigned to search through all the keys with the 16 most-significant bits fixed, in total 2^{40} different keys. The key space is partitioned by a connected host-PC so that each chip takes around 150 minutes (at 120 MHz) to test all ANSI X9.9 authenticators in its allocated key subspace. During a single check of an authenticator, the DES engines use the first challenge (*plaintext 1*) as a primary input to the encryption function. Then, the upper 32-bits of the generated ciphertext is mapped digit-per-digit by the function μ and compared to the value of the response stored in the register *ciphertext 1*. If any of the DES engines provides a positive match, the corresponding engine switches its input to the second challenge encrypting it with the same key. To match the pipelined design of the DES engine, we are using a shadow counter (*CNT 2*) tracking the key position at the beginning of the pipeline. In case that the

derived authenticator from the second encryption compares successfully to the second response, the controller *CTL* reports the counter value to the host-PC as a potential key candidate. The host-PC keeps track of the key range that is assigned to each of the FPGAs and, hence, can match the right key from a given counter value. If no match is found until the counter overflows, the FPGA reports completion of the task and remains idle until a new key space is assigned. In case that a third challenge-response pair was specified, the host-PC performs a verification operation of the reported key candidate in software. In case the verification was successful, the search is aborted and the key returned as result of the search.

We have implemented the FPGA architecture shown in Figure 5.8 using the Xilinx ISE 9.1 development platform. After synthesis of the design incorporating four DES engines and the additional logic for the derivation of the ANSI X9.9 authenticator, the usage of 8,729 Flip-Flops (FF) and 12,813 Look-Up Tables (LUT) was reported by the tools (56% FF and 83% LUT utilization of the Spartan-3 1000 device, respectively). As discussed in Section 5.5, we included specific optimizations like pipelined comparators since n-bit comparators are likely to introduce a long signal propagation path reducing the maximum clock frequency significantly. By removing these potential bottlenecks, the design can be clocked at 120MHz after place-and-route resulting in a throughput of 480 million keys per FPGA per second. In total, a fully equipped COPACOBANA with 120 FPGAs can compute 57.6 billion ANSI X9.9 authenticators per second. Based on this, we can present time estimates for an attack provided that two challenge-response pairs are given. Recall that in this scenario we will be faced with several potential key candidates per run so that we have to search the entire key space of 2^{56} to build a list with all of them. This ensures that we are able to identify the actual key in a separate step.

Similarly, we can present figures for an attack scenario where three challenge-response pairs are available. In this attack, we must test 2^{55} ANSI X9.9 authenticators on average to find the corresponding key what is half the time complexity of an attack having two known pairs of data. Note that all presented figures of Table 5.3 include material costs only (not taking energy and development costs into account).

For comparison with our hardware-based cluster, we have included estimations for a Intel Pentium 4 processor operating at 3GHz. This microprocessor allows for a throughput of about 2 million DES computations a second what we also assume as appropriate throughput estimate for generating ANSI X9.9 authenticators.

5.8 Conclusions

In this chapter we have shown advanced attack implementations to break the DES block cipher and related products with COPACOBANA. On up to 120 low-cost FPGAs, COPACOBANA is able to perform the required cryptographic operations simultaneously and in parallel for applications with high computational but low memory and communication requirements.

Hardware System	**Cost**	**Two pairs** (c_i, r_i)	**Three pairs** (c_i, r_i)
1 Pentium4 @ 3GHz	US$ 50	1170 years	585 years
1 FPGA XC3S1000	US$ 50	4.8 years	2.4 years
1 COPACOBANA	US$ 10,000	14.5 days	7.2 days
100 COPACOBANAs	US$ 1 million	3.5 hours	104 min

Table 5.3: Cost-performance figures for attacking the ANSI X9.9 scheme with two and three known challenge-response pairs (c_i, r_i).

We presented an improved key search implementation that can break DES within less than a week at an average computation rate of 65.3 billion tested keys per second. Beside the simple brute-force scenario on DES, we have extended the attack scheme for tackling the complexity of ANSI X9.9 OTP tokens whose security assumptions rely on the DES. Smarter brute-force attacks, particularly when identical plaintexts are encrypted with different keys, can be achieved by Time-Memory Tradeoffs. Due to immature communication facilities of COPACOBANA, we could not fully verify our estimated results yet. However, as soon as higher bandwidth becomes available on COPACOBANA, we will complete the generation of the precomputation tables for the TMTO attack on DES in less than a month.

Chapter 6

Parallelized Pollard-Rho Hardware Implementations for Solving the ECLDP

As discussed in Chapter 3, the use of Elliptic Curves (EC) in cryptography is very promising for embedded systems since they are resistant against powerful index-calculus attacks and thus allow small parameter sizes. In this chapter, we analyze the actual security margin provided by ECC more precisely and present a first concrete hardware implementation of an attack against ECC over prime fields. In detail, we describe an FPGA-based hardware architecture of the Pollard-Rho method which is, to our knowledge, currently the most efficient attack against ECC. With the implementation at hand, a fairly accurate estimate about the cost of such an attack based on FPGAs as well as on ASICs can be given.

Contents of this Chapter

6.1 Motivation

Elliptic Curve Cryptosystems are based on the difficulty of the Diffie-Hellman Problem (DHP) in the group of points on an Elliptic Curve (EC) over a finite field. The DHP is closely related to the well studied Discrete Logarithm Problem (DLP). In contrast to the more efficient index-calculus attacks on the DLP over finite fields, ECC only allows for generic attacks such as Pollard's Rho method [Pol78, vOW99]. This benefit yields much shorter underlying bit lengths for ECC (160–256 bit) compared to RSA or DLP in finite fields (1024–4096 bit) at an equivalent level of security [LV01].

For RSA and DLP in finite fields, many publications address the issue of hardware-based attacks and provide security estimates based on proposals of architectures attacking such cryptosystems. For ECC, however, no *precise* architecture for a hardware attack has been described yet. Cryptanalysis of ECC requires the same algorithmic primitives as the cryptosystem itself, namely point addition and point doubling which can be implemented very efficiently in hardware. An algorithm designed for concurrency, the parallel Pollard's Rho, is described in [vOW99] which requires a unique point representation but achieves a perfect linear speedup dependent on the number of available processors. To the best of our knowledge, except for estimations given in this contribution, no actual results of a hardware implementation of Pollard's Rho algorithm for solving the ECDLP have been published.

In this chapter, we will present first results of a hardware implementation of an attack against ECC over prime fields, allowing a security evaluation of ECC taking cryptanalytical hardware into account. We propose an efficient architecture for the parallel variant of Pollard's Rho method and its implementation in hardware. Besides the hardware architecture which has been completely programmed in VHDL and realized on a low-cost FPGA (Xilinx Spartan-3), this project involves external software components required for managing the hardware components. Based upon first results of the running hardware implementation, we estimate the expected runtime to break ECC for actual security parameters. We also give estimates for an ASIC design, solely dedicated to tackle actual ECC Challenges [Cer97] based on our architecture. Extracts of this work were published as joint work with Jan Pelzl [GPP07a, GPP08].

6.2 Previous Work

We briefly summarize previously published results of importance to this contribution. Relevant work includes publications describing attacks against ECC in hardware over prime fields. To our knowledge, there is no actual hardware implementation realizing such an attack. Since the method of choice for attacking ECC is Pollard's Rho which involves EC group operations, recent results on the efficiency of EC group operations over $GF(p)$ in hardware are of interest.

Proposals of hardware-based attacks are very rare. The most important work in this field is provided by [vOW99]: Besides an algorithmic improvement which allows for an efficient parallelization of Pollard's Rho method, the authors estimate the cost of a dedicated hardware solving the DLP over a curve over $GF(2^{155})$ to 32 days for US$ 10 million. However, specific details of the parametrization are omitted (i.e., it is unclear how the distinguished point property can be used in the projective domain).

A recently published contribution discusses attacks on ECC over binary fields with reconfigurable hardware [dDBQ07]. The authors report their results also taking COPACOBANA as reference platform. However, in contrast to curves over $GF(2^m)$, curves over GF(p) have not been examined yet.

6.3 Mathematical Background

In this section, we will briefly introduce to the mathematical background relevant for this chapter. We will start with a review of the Elliptic Curve Discrete Logarithm Problem (ECDLP) and describe possible ways of solving it. For the scope of this work, the parallel variant of Pollard's Rho method is of major interest and thus will be described in more detail.

6.3.1 The Elliptic Curve Discrete Logarithm Problem

The ECDLP is analagous to the DLP in finite fields. The security of many cryptographic protocols such as, e.g., the Diffie-Hellman key exchange [DH76] and the ElGamal encryption scheme [Elg85] is based on the DLP.

Let p be a prime with $p > 3$ and $\mathbb{F}_p = GF(p)$ the Galois Field with p elements. The ECDLP input is an elliptic curve

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

with $a, b \in \mathbb{F}_p$ and $4a^3 + 27b^2 \neq 0$, two points $P, Q \in E(\mathbb{F}_p)$, and $\langle P \rangle$ generator of a sufficiently large subgroup. The ECDLP problem is to find an integer ℓ such that

$$\ell \cdot P = Q, \tag{6.1}$$

where $Q \in \langle P \rangle$ holds. The parameter ℓ often is denoted as the elliptic curve discrete logarithm

$$\ell = \log_P(Q).$$

6.3.2 Best Practice to Solve the ECDLP

Most known attacks on ECC have exponential complexity. This statement holds for generic curves and excludes attacks on special subclasses like supersingular and anomalous curves. This work intends to analyze the security of cryptosystems based on cryptographically strong curves and, thus, weak curves are not considered. Please see [HMV04, ACD+05] for more information how to generate elliptic curves suitable for practical cryptosystems and how to avoid the usage of weak curves.

The ECDLP, i.e., the solution of Equation (6.2) can be computed by using following techniques:

- *Näive exhaustive search:* this method sequentially adds the point $P \in E$ to itself. The addition chain $P, 2P, 3P, 4P, \ldots$ will eventually reach Q and reveal ℓ with $\ell \cdot P = Q$. In the worst case, this computation can take up to n steps where $n = ord(P)$, making this attack infeasible in practice when n is large.
- *Baby Step Giant Step (BSGS):* the BSGS algorithm is an improvement to the näive exhaustive search [Sha71]. For $n = ord(P)$, memory for about $\sqrt{n}$ points and approximately

$\sqrt{n}$ computational steps are required. Due to its high memory complexity, BSGS is suitable only for very restricted sizes of n.

- *Pollard's Rho and Lambda method:* in 1978 J. Pollard proposed [Pol78] two collision based algorithms. Although having a similar time complexity compared to BSGS, both methods are superior due to their negligible memory requirements.

  The ECDLP can be most efficiently computed using Pollard's Rho method or in case of an a priori specified search interval $[a, b]$ by Pollard's Lambda method (also known as Kangaroo method). In combination with adequate parallelization, Pollard's Rho method is the fastest known attack against general ECC for solving the ECDLP in time of roughly $\sqrt{\pi n/2}$. Thus, it will be the algorithm of choice for the remainder of this chapter.

6.3.3 Pollard's Rho Method

The Pollard-Rho attack as proposed by J. Pollard in 1978 [Pol78] is a collision based algorithm performing on a random walk in the point domain of an EC. As mentioned above, it has similar computational complexity compared to the Baby Step Giant Step algorithm of about $\sqrt{\pi n/2}$, but it is superior due to its negligible memory requirements. In combination with parallel processing, the Pollard-Rho is the fastest known attack [HMV04] against ECC.

To explain the Pollard-Rho in more detail, we should first outline why point collisions can help to reveal the ECDLP. Let $R_1 = c_1P + d_1Q$ and $R_2 = c_2P + d_2Q$ be two points with $R_1, R_2 \in E(\mathbb{F}_p)$ and $R_1 = R_2$ but $c_1 \neq c_2$ and $d_1 \neq d_2$. Then the following statements hold [HMV04]:

$$\begin{aligned} c_1P + d_1Q &= c_2P + d_2Q \\ (c_1 - c_2)P &= (d_2 - d_1)Q \\ (c_1 - c_2)P &= (d_2 - d_1)\ell P \\ \ell &= (c_1 - c_2)(d_2 - d_1)^{-1} \bmod n \end{aligned} \tag{6.2}$$

Hence, in case of a point collision in the subgroup of P the ECDLP can be solved efficiently if $gcd(d_2 - d_1, n) = 1$. Next is the issue of how to find such a collision.
A naïve approach would be to take a starting point $S = c_sP + d_sQ$ with $c_s, d_s \in_R \{2, \ldots, n\}$ chosen randomly. A second point $T_1 = c_{t1}P + d_{t1}Q$ with other randomly chosen coefficients is used to compute $R_1 = S + T_1$. Then a third random point T_2 determined the same way will lead to a further point $R_2 = S + T_2$ which is compared against previous results. This procedure can be continued until a correspondence of points is located with the drawback that all results (about $\sqrt{n}$ due to the birthday paradox) need to be stored. The enormous space requirements would make this attack similarly costly as the Baby Step Giant Step algorithm.

The better solution is to have a random walk [Pol78] within the group. This is a pseudo-random function determining a collision candidate using an addition chain with a finite set of preinitialized random points. In other words, we have a function f taking a current point X_j of

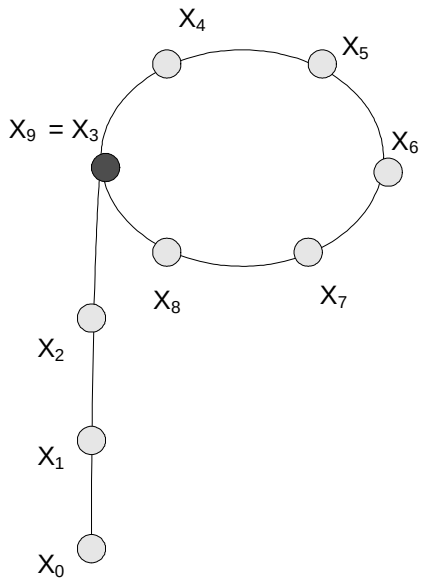


Figure 6.1: Single Processor Pollard-Rho (SPPR).

the EC as input and computing its successor X_{j+1} by simply adding another point. A repetition of this procedure produces a pseudo-random trail of points in the domain of $\langle P \rangle$. The other point, which is added each time, is determined from a set $\mathcal{R}$ of previously randomly chosen points and is selected in each iteration by a partitioning function. Let $R_i \in \mathcal{R}$ be the i-th out of s total random points with $i = \{0, \ldots, s-1\}$. Then, we can define a partitioning function g which determines the next random point R_i to add:

$$g : E(\mathbb{F}_p) \rightarrow \{0, \ldots, s-1\} \ : \ X \mapsto i.$$

When we integrate g into the function f, we obtain a next point X_{j+1} by:

$$f : E(\mathbb{F}_p) \rightarrow E(\mathbb{F}_p) \mid X_{j+1} := X_j + R_{g(X_j)}.$$

Due to the finiteness of $\mathcal{R}$, the trail generated by a repetitive utilization of f will always run into a cycle and therefore eventually collide in some point. The shape depicted by the random walk is similar to the greek letter ρ and is hereby namesake for this algorithm. The collision itself can easily detected using Floyd's cycle finding algorithm which requires only a second computation advancing twice as fast as the first one [MvOV96]. Hence, except for two computations, no additional storage is required. Figure 6.1 depicts the construction of the trail respectively a random walk in the Pollard-Rho algorithm. Assume X_0 to be the starting point of the trail. A repeated application of f with $X_{j+1} = f(X_j)$ will lead to each next point in the walk. Finally, we will encounter a collision caused by a duplicate visit at point X_3 and X_9, respectively.

The complexity for this Single-Processor Pollard-Rho (SPPR) algorithm is derived directly from the collision probability given by the birthday paradox [Pol78]. The birthday paradox

deals with the random and repetitive selection of elements from a distinct set until a duplicate is chosen. Assuming Z to be a random variable for the number of chosen elements, the probability that j out of n elements can be selected without duplication is

$$Pr(Z > j) = (1 - \frac{1}{n})(1 - \frac{2}{n}) \dots (1 - \frac{j-1}{n}) \approx e^{\frac{-j^2}{2n}}.$$

This finally leads to an expected number of distinct elements of roughly $\sqrt{\pi n/2}$ before a collision occurs and the algorithm terminates. A proof of this statement can be found in [vOW99, FO90].

The Multi-Processor Pollard-Rho (MPPR), proposed by van Oorschot and Wiener [vOW99], is basically a variation of the previously presented SPPR with some modification for better support of the parallel idea and providing a linear speedup with the number of available processors.

Each processor $w_i \in \mathcal{W}$ where $\mathcal{W}$ is the set of processors starts an individual trail but does not focus on terminating in a cycle like in the original method. The primary goal is to find collisions of different computationally independent trails. For this purpose, a selection criteria for points on the trail is defined, marking a small partition of all computable points as *distinguished*. In our case we simply assign this property to points with the bits of an x-coordinate showing a specific number of consecutive zeros. This is similar to the methodology as used in Section 5.3.1 for distinguished points in TMTOs.

When a trail arrives at a Distinguished Point (DP), the corresponding processor transmits the DP to a central server which keeps track of all submitted DPs and checks for any duplicates. In case of a duplicate resulting from a different origin, a collision is found and the algorithm terminates successfully (when $c_1 \neq c_2$ and $d_1 \neq d_2$).

Figure 6.2 depicts how several point processors contribute to the collision search. Every darker spot in the trail of a processor corresponds to a DP. An eventual collision of two trails in a common distinguished point is highlighted in black for the processors w_3 and w_4. In contrast to the application of completely independent instances according to the original Pollard-Rho, the centralized collection of DP in the MPPR achieves a linear speedup with multiple instances. On average, a collision occurs after the computation of

$$T = \sqrt{\pi n/2} + c \tag{6.3}$$

points, where n is the order of the actual subgroup. The central unit will detect the merging of two trails when the next DP in the (joint) trail is found. The term c takes the additional overhead of collecting points in a joint trail into account. If we assume that all available processors directly contribute to the threshold T, the workload of a single processor $w_i \in \mathcal{W}$ is given by Equation (6.4) [vOW99]. Algorithm 6.1 depicts the MPPR method.

$$T_{\mathcal{W}} = \sqrt{\pi n/2}/\left|\mathcal{W}\right| + c \tag{6.4}$$

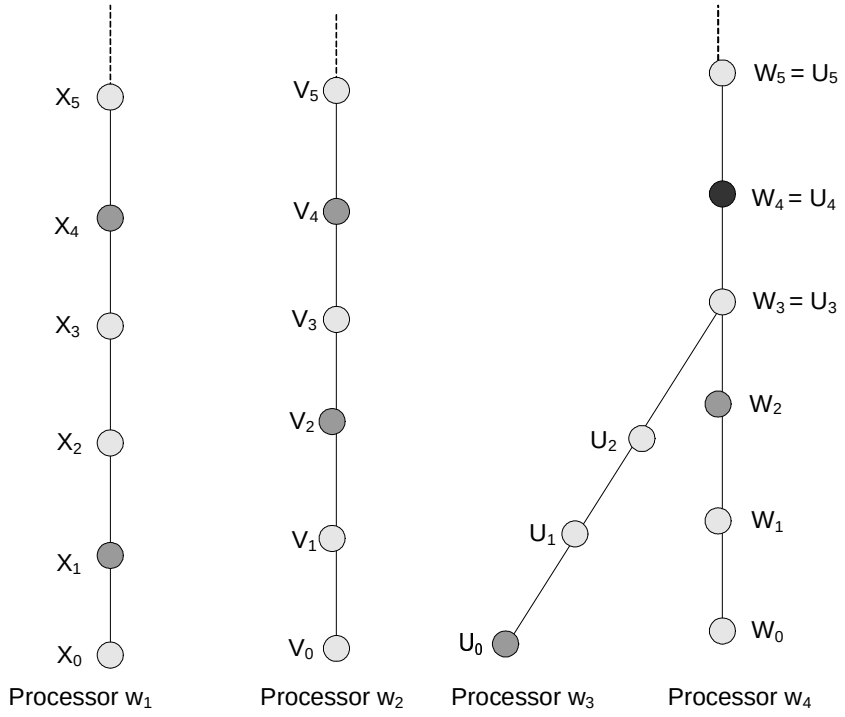


Figure 6.2: Multi-Processor Pollard's Rho.

6.4 An Efficient Hardware Architecture for MPPR

For the implementation of Algorithm 6.1, we need a star topology in which a central server is collecting distinguished points from a multitude of attached computational (FPGA-based) processors $w_i \in \mathcal{W}$.

6.4.1 Requirements

For the realization of MPPR the server requires the following features:

- A **communication controller** for data exchange with the $|\mathcal{W}|$ point processors.
- A **database** for storing the tuples (c, d, Y) for a point $Y = cP + dQ$ in a sorted table according to Y for efficient point recovery.
- A unit for **validating distinguished points** received from a processor w_i. This step is mandatory to protect against defective processors which might spoil the entire computational result by eventually corrupted results from a point processor w_i. This happens rarely when voltage drops and power fluctuations in a large hardware system spoil individual arithmetic computations.
- An arithmetic unit for **computing the discrete logarithm** from a detected collision.

Algorithm 6.1 Multi-Processor Pollard's Rho

Input: $P \in E(\mathbb{F}_p); n = ord(P), Q \in \langle P \rangle$

Output: The discrete logarithm $\ell = log_P(Q)$

1: Select size s and partitioning function $g : \langle P \rangle \rightarrow \{0, 2, \ldots, s-1\}$
2: Select set D out of $\langle P \rangle$
3: **for** $i = 0$ to $s - 1$ **do**
4: Select random coefficients $a_i, b_i \in_R [1, \ldots, n-1]$
5: Compute $R_i \leftarrow a_i P + b_i Q$
6: **end for**
7: **for** each parallel processor **do**
8: Select $c, d \in_R [1, \ldots, n-1]$
9: Compute $X \leftarrow cP + dQ$
10: **repeat**
11: **if** $X \in D$ is a distinguished point **then**
12: Send (c, d, X) to the central server
13: **end if**
14: Compute $i = g(X)$.
15: Compute $X \leftarrow X + R_i;\ c \leftarrow c + a_i \ mod\ n;\ d \leftarrow d + b_i \ mod\ n$
16: **until** a collision in two points was detected on the server
17: **end for**
18: Let the two colliding triples in point Y be (c_1, d_1, Y) and (c_2, d_2, Y)
19: **if** $c_1 = c_2$ **then**
20: **return** failure
21: **else**
22: Compute $\ell \leftarrow (c_1 - c_2)(d_2 - d_1)^{-1} \ mod\ n$
23: **return** ℓ
24: **end if**

- An optional **EC generator** for testing purposes.

In contrast to the connected point processors which are designed to run on special-purpose hardware, the central server has only modest computational and throughput requirements. Depending on the size of the subset of distinguished points and on the number of actual point processors, a distinguished point has only to be processed once in a while. Thus, it is sufficient to model our central station in software which simplifies the development process.

6.4.2 Proposed Architecture

With the central server implemented in software (cf. Figure 6.3), we can focus on the point processors $w_i \in \mathcal{W}$ which are subject to the hardware implementation. In this section, we will describe an FPGA implementation of the point processors. Due to the relatively low cost at low quantities and the reconfigurability, the choice of FPGAs seems optimal for a first evaluation

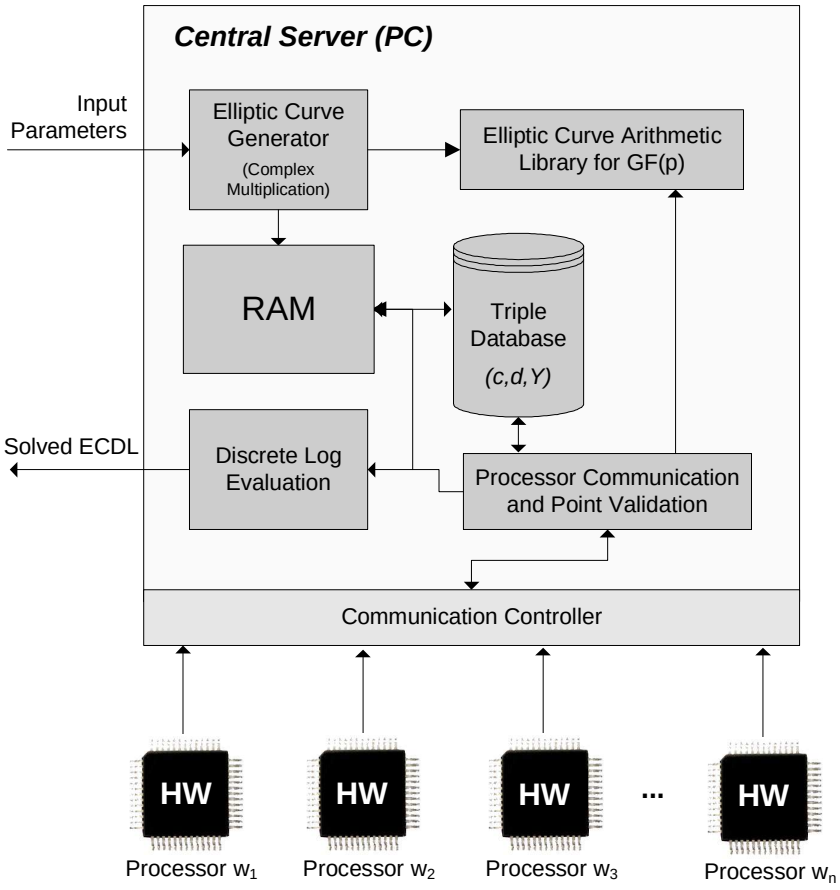


Figure 6.3: The central software-based management server.

of MPPR in hardware. Note that a possible ASIC implementation would dramatically decrease the monetary effort of MPPR if produced at high volumes. However, for solving ECDLP over smaller groups ($\leq$128 bit), an FPGA-based design can be implemented with COPACOBANA (cf. Chapter 5).

For the implementation of a point processor, the computation of a trail of consecutive points consists of four steps:

(1) **DP Detection.** We need to detect if the current point X is distinguished. In hardware, we can agree on a simple DP property which is satisfied when the most significant δ bits of the x-coordinate of the current point $X = (x, y)$ are zero. This can be easily implemented by a δ-bit comparison. If we find such a point, we transmit it to the central server. The benefit of this strategy is the immediate control of the number of point transfers between database server and point processors in hardware. Choosing a bigger value for δ will result in longer computation trails and less points to be found on the point processors which

reduces the overall bandwidth requirement. Hence, we can adapt δ accordingly to meet any bandwidth limitation between server and hardware processors.

Note that an efficient verification of the distinguished property does only work in affine coordinates. Affine coordinates suffer from expensive field inversions which are mandatory to compute the group addition. Projective coordinate systems avoid inversions by encoding the inversion in a separate variable at the cost of additional field multiplications. Unfortunately, the projective point representation is not unique and does not allow for an efficient check of the distinguished property. The reason for this is that an affine point $P = (x, y)$ has $p - 1$ different projective representations, e.g., $P = (X, Y, Z)$ satisfying $x = X/Z^{\alpha}$ and $y = Y/Z^{\beta}$ which do not yield a unique property for DP detection. Consequently, we will use affine coordinates in our implementation.

(2) **Partitioning.** In order to select the next random point R_i and its corresponding coefficients a_i and b_i, we need to identify a partition $i = 0, \ldots, s - 1$ according to the current point X. In hardware, it is straightforward to choose a power of 2 for s since we simply can use $\log_2 s$ least significant bits from x to map to the partitioning value i. We choose a (heuristically determined) optimal value $s = 16$ which comes close to Teske's proposal in [Tes98].

(3) **Point Operation.** We need to update the current point by adding a random R_i which requires a point addition or, in case of $X = R_i$, a point doubling. The latter case actually means that a collision is not detected among two distinguished points rather than the current point X and a random point R_i. Due to the fact that this case is very unlikely with a probability of $Pr(X = R_i) = s/n$, we will omit the additionally required logic for this situation to save hardware area.

(4) **Coefficient Update.** Finally, we need to update the corresponding coefficients c and d for point X with $X = cP + dQ$ by adding the random coefficients $c = c + a_i \bmod n$ and $d = d + b_i \bmod n$ according to the selected partition i.

In the following, we introduce the design of a top layer in Subsection 6.4.2 which cares for administrative tasks such as, e.g., providing a centralized distinguished point buffer (DPB) as well as a processor controller (PC) which is responsible for the control of the several computational cores. In Subsection 6.4.2 we will describe a single computational core and its arithmetic unit.

Top Layer

The top layer is composed of the previously mentioned DPB, PC, and a physical Communication Controller (CC), as depicted by a schematic outline in Figure 6.4. The CC is required to control the data input and output from and to the central server, respectively. Recall that no high data communication between FPGA and central server is required since we can adjust the distinguished property to reduce the load of the communication path.

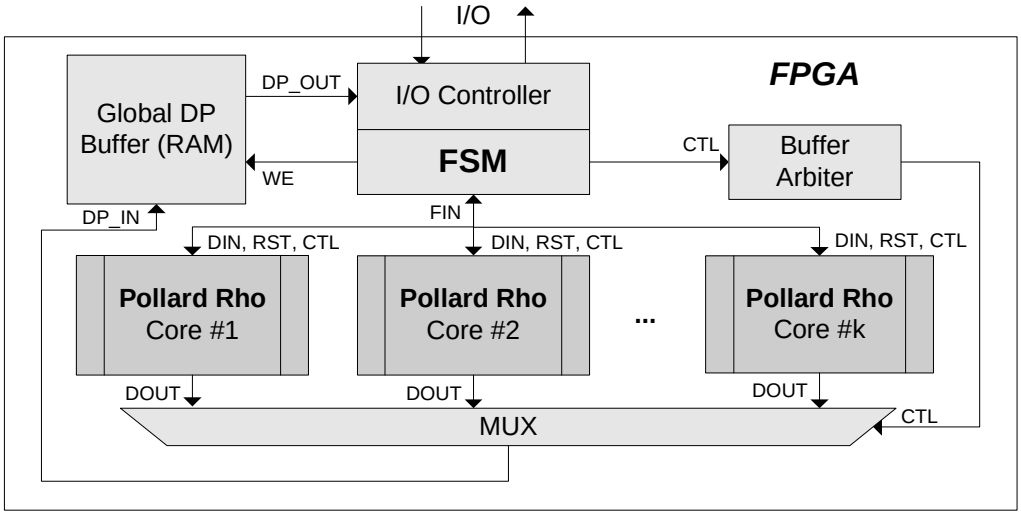


Figure 6.4: Top layer of an FPGA-based point processor.

A gain in performance can be achieved by using more computational cores on a single FPGA, computing many trails in parallel. Obviously, this procedure requires a careful management regarding the data input and output. With many PR cores and the centralized buffer, the access to the DPB demands multiplexing and buffer locking capabilities to prevent two or more cores from writing to the buffer at the same time. On the other hand, the DPB is very useful since it allows for an independent external access to the stored distinguished points without the need to interrupt a core from computing. Figure 6.4 shows the computational cores as an entity which will be discussed in detail in the following.

Core Layer

For an optimal design, all operations of the core should make use of extensive resource sharing because we are not able to use embedded, arithmetic function blocks (e.g., DSP blocks as seen in Chapter 3) which are not available on Spartan-3 FPGAs. Hence, due to the design flexibility with reconfigurable logic, we can combine all costly field operations in a central Arithmetic Unit (AU). The AU entity will be discussed in Subsection 6.4.2.

An important component of the core layer is the core controller. It is primarily responsible for managing the operations of the AU and delegating its output to memory locations. Another element of this layer is the main memory, providing the operands to the AU. A schematic of the core layer architecture is given by Figure 6.5.

The AU demands for three concurrent inputs: Two inputs provide the operands IN1 and IN2, one input provides the modulus MOD. The modulus can take two possible values: $n = ord(P)$ for updating the coefficients c_i, d_i and p for all other computations. For the operands, we use a dual-port block memory module which is available on the actual Spartan-3 FPGA. The dual-port memory contains all k-bit variables for the actual point addition including the starting

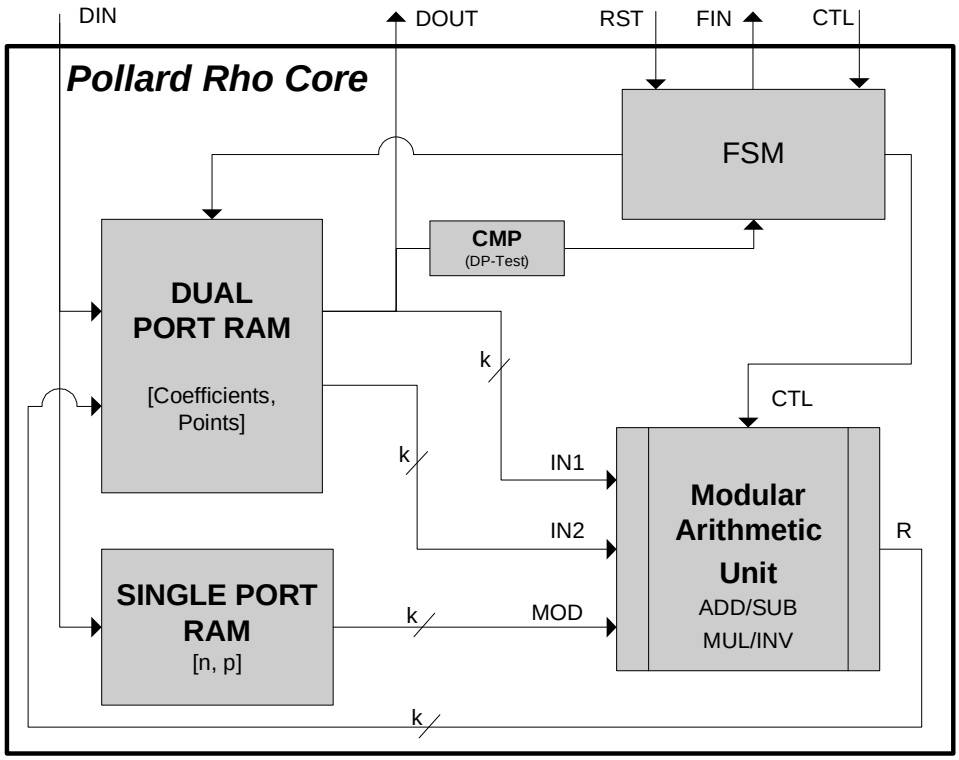


Figure 6.5: Core layer of an FPGA-based point processor.

point X, associated coefficients c, d, and temporary values. Besides, it can hold the random point data $(R_i = (x_i, y_i), a_i, b_i)$ for $i = 0, ..., s-1$ in a separate region. Hence, variables and random data are available via a common access path requiring no further multiplexing. For the modulus we use a separate single-port memory module. Compared to two individual registers with multiplexers and separate control lines, this approach is more area efficient.

Arithmetic Unit

The overall runtime of an implementation of MPPR is dominated by the performance of the field arithmetic, used for the point addition and coefficient updates. Hence, the efficient implementation of the arithmetic functions such as field addition, subtraction, multiplication, and inversion is crucial for the performance of the overall system. Due to the predominance of the modular inversion within the point addition function, the optimization of other operations has not been considered since a dramatic reduction of the critical path is not expected.

The arithmetic unit provides the functionality of field addition, subtraction, multiplication, and inversion required by the upper layers. Modular addition and subtraction are rather simple operations. In contrast, modular multiplication and inversion takes much more computational effort. For an efficient implementation of such we use coordinates in the Montgomery domain, allowing for algorithms which replace a costly modular reduction by divisions by two (imple-

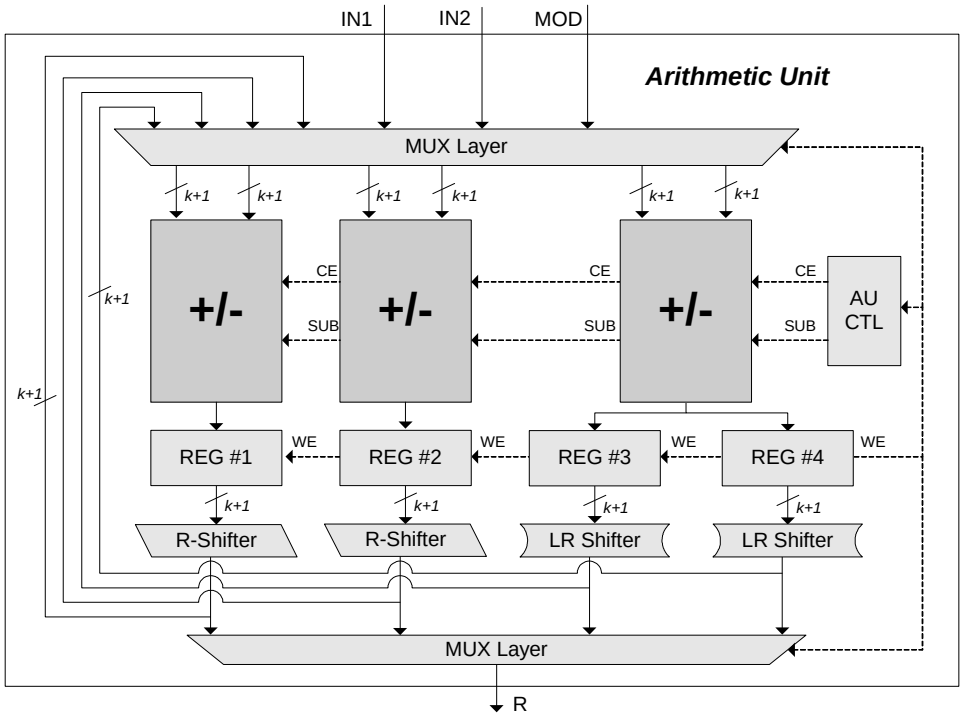


Figure 6.6: Arithmetic layer of an FPGA-based point processor.

mented by right shifts by one) [MvOV96]. Thus, we use a bit-wise Montgomery multiplication (cf. Section 4.4.1) and a modified Kaliski inversion algorithm [Kal95]. For more information regarding the implementation see [Gün06]. All field operations require k-bit additions and subtractions which we realize with Xilinx IP cores. The IP cores are highly optimized for a specific FPGA and usually provide a better performance-area ratio than conventional implementations.

The final design of our AU is shown in Figure 6.6. For the sake of clarity, we combined the logic for the shifting units and the multiplexing units in combined blocks.

6.5 Results

A Xilinx Spartan-3 device (XC3S1000) was used as target platform for the implementation of the proposed architecture. Corresponding results obtained from Xilinx ISE 8.2 are presented in Subsection 6.5.1. For comparison and debugging purposes, a software implementation of the complete MPPR has been realized on a conventional processor (Pentium M). Based on the running FPGA implementation, we estimate the cost and performance of an ASIC implementation of the MPPR in a further step in Subsection 6.5.3 and finally compare our findings in Subsection 6.5.4.

k	**Cores**	**Slices/Core**	**Max. Clock**	**BRAMs**	**PRCore RAM**	**Buffer**
160	2	3,230	40.0 MHz	15	13k	20k
128	3	2,520	40.1 MHz	16	10k	16k
96	3	2,030	48.1 MHz	12	8k	12k
96	4	1,890	44.3 MHz	15	8k	12k
80	4	1,710	50.9 MHz	15	7k	10k
64	4	1,450	54.8 MHz	10	5k	8k
64	5	1,360	52.0 MHz	12	5k	8k

Table 6.1: Synthesis results of MPPR for Spartan-3 XC3S1000 FPGAs.

6.5.1 Synthesis

We synthesized the design for various bit sizes k and the maximum number of cores fitting on one FPGA. Table 6.1 gives the results for a Xilinx XC3S1000 which provides 17,280 logic cells in 7,680 slices with a variable number of cores. Note that the number of occupied slices includes all top-layer components for management and communication, as well. Additionally, the required number of memory bits and BRAMs for the PRCore and buffer components for the synthesized design are given. Please note that for the RAM-size in bit, k denotes $2^{10} = 1024$.

6.5.2 Time Complexity of MPPR

With the given time complexity of all core functions, we can provide the exact number of cycles required for an iteration of MPPR. Along with the maximum clock frequency, detailed numbers for the throughput of the whole design can be given.

An elliptic curve point addition in affine coordinates can be performed using 6 subtractions, 3 multiplications, and a single inversion. The update of coefficients demands another two additions. A few administrative cycles are required by the core for resetting (RST) and post-

Operation Type	**Count**	$C_{\mathrm{CMP}}(k)$	C_{RST}	C_{FIN}
Partitioning	1	$1/2$	-	-
DP Check	1	$1/2$	-	-
Subtractions	6	6	12	6
Multiplications	3	$3(k+2)$	6	3
Inversion	1	$2(k+2)$	2	1
Addition	2	2	4	2
Total	14	$5(k+2)+9$	24	12

Table 6.2: Required cycles for one MPPR iteration with bit size k.

k	$C(k)$	**Time Pt/μs**	**Max. Clock**	**Pts Core/s**	**Total Pts/s**
160	855	40.0 MHz	21.4	46,800	93,600
128	695	40.1 MHz	17.3	57,800	173,000
96	535	48.1 MHz	11.1	90,000	270,000
96	535	44.3 MHz	12.1	82,700	331,000
80	455	50.9 MHz	8.94	111,900	447,000
64	375	54.8 MHz	6.84	146,200	585,000
64	375	52.0 MHz	7.21	138,600	693,000

Table 6.3: The attack performance of MPPR on Spartan-3 XC3S1000 FPGAs.

processing (FIN). Finally, we can provide a list for the overall cycle count of a single MPPR iteration as shown in Table 6.2, resulting in following complexity function:

$$C(k) := 5(k+2) + 9 + 24 + 12 = 5k + 55. \tag{6.5}$$

With Equation (6.5), we can easily compute the performance for the selected architectures including the number of required cycles per iteration and the number of points per second. The overall system performance for our FPGA-based implementation of MPPR is shown in Table 6.3.

Note that the numbers above exclude additional cycles for writing to the distinguished point buffer. Only when computing the ECDLP in small groups (< 80 bit), the buffer might become a bottleneck and the additional cycles for data output have to be taken into account. Additionally, a larger number of MPPR cores per FPGA increases the demand for a large output buffer. However, this can easily be avoided by reducing the size of the distinguished point set D. A direct consequence are longer search trails for distinguished points and, thus, collisions of two or more cores requesting a buffer lock at the same time will become extremely unlikely.

6.5.3 Extrapolation for a Custom ASIC Design of MPPR

The cost and performance of an ASIC realization will be estimated based on the results of the FPGA implementation. For an appropriate estimate, our application specific design will be limited to 10 million gates which is approximately a tenfold of the logic resources of the Xilinx XC3S1000 FPGA and which we believe to be fairly realizable with an acceptable effort. Furthermore, we can expect to run such a chip at clock rates beyond the scope of low-cost FPGAs. Hence, a maximum clock frequency of 500 MHz seems realistic for the chosen algorithms. Table 6.5 provides the estimate of the throughput for a possible chip design for MPPR over $GF(p)$ with a k-bit prime p and a maximum possible number of MPPR cores per chip.

Please remark that all ASIC related numbers in this work are estimates and are subject to change in case of a future ASIC simulation. However, we believe that the estimates are fairly accurate and will not differ in any order of magnitude. Moreover, in case of a deviation we

k	$\mu(T_H, T_L)$	**Pentium M**	**XC3S1000**	**ASIC**
64	$3.25 \cdot 10^9$	5.14 h	78.1 min	-
80	$8.32 \cdot 10^{11}$	70.5 d	21.5 d	-
96	$2.13 \cdot 10^{14}$	55.1 y	20.4 y	-
128	$1.40 \cdot 10^{19}$	$4.86 \cdot 10^6$ y	$2.55 \cdot 10^6$ y	$2.05 \cdot 10^4$ y
160	$9.15 \cdot 10^{23}$	$3.78 \cdot 10^{11}$ y	$3.10 \cdot 10^{11}$ y	$2.48 \cdot 10^9$ y

Table 6.4: Expected runtime for MPPR on a single chip.

expect an improvement of area-time complexity and, thus, the numbers provided should be seen as an upper bound on the hardware complexity of MPPR.

6.5.4 Estimated Runtimes for Different Platforms

Along with the given throughput from Subsection 6.5.2, projections are performed for bit lengths $k = \{64, 80, 96, 128, 160\}$ using a fixed distinguished point proportion of $\Theta = 2^{-24}$. We will regard following architectures: First, projections are done for the software reference, running on a conventional off-the-shelf processor (Pentium M 735@1.7 GHz). Second, we will estimate the performance on the XC3S1000 FPGA. Thirdly, an estimate for dedicated ASIC will be given for bit sizes of 128 bit and above[1]. Clearly, the performance of the hardware architecture heavily depends on the varying bit size, indicated by different runtimes and varying area consumption. For small bit lengths, several cores can be realized on the FPGA and, thus, slightly decrease the maximum clock speed of the entire system. However, the additional computational power compensates the lower clock speed.

We compute the expected number of points for a specified bit size to determine the ECDLP of an associated curve. At this time, we discard any negative effect of overwriting points on the central station which in fact needs to be considered when the number of available distinguished points becomes too huge to be stored without compression. If we assume a geometrical distribution of distinguished points we can show that only negligible effects from overwriting points are expected for bit sizes ≤ 96 bit. This statement is true for an available size of server storage of $2^{24} \approx 1.68 \cdot 10^7$ points.

Table 6.4 presents the results for the expected computation time with a single chip (Pentium M, XC3S1000 FPGA, and ASIC). We have determined the expected number of steps to finish a successful attack based on an interval T_H and T_L denoting the upper and lower limit of the point order n for a given ECC bit size k. Using the mean $\mu(T_H, T_L)$ based on Equation 6.3, we determined a step complexity of an average n for a bit size k. Relating the number of steps and the performance of the analyzed platforms, a successful computation of the ECDLP for $k > 96$ seems to be unrealistic with any of the presented architectures when considering only a single chip.

[1]For lower bit sizes, the relatively high NRE costs outweigh the production costs.

6.6 Security Evaluation of ECC

Attacks against ECC based on the three previously presented architectures are considered for the detailed security evaluation:

- a software implementation on a cluster of conventional off-the-shelf processors
- an FPGA cluster built of the FPGA-type of our implementation, and
- a dedicated ASIC hardware consisting of a cluster of the ASICs.

We will estimate the cost of such cryptanalytical systems for different scenarios: We consider a successful attack against ECC in one year and compare the results to the previously proposed hardware attacks against RSA in [ST03b, FKP$^+$05]. Finally, we estimate the approximate cost to solve the ECC challenges given by [Cer97]. Please remark that these estimates are based on 2007 numbers for hardware costs. For actual comparisons, Moore's Law should be taken into account.

6.6.1 Costs of the Different Platforms

Giving a simple comparison of the software implementation and the optimized hardware architectures is not very meaningful. Since the optimization of our software implementation is not within the scope of this contribution, we will take into account possible improvements by using a correction factor of 1.25. Hence, a possible speedup by 25% can be achieved by using assembler code, placing our software solution into a more competitive position against its relative in hardware. Secondly, we have to take the different monetary cost of general purpose processors, FPGAs, or ASICs into account. For the sake of simplicity, we do not consider power consumption. However, we assume the power consumption of special-purpose hardware to be far below that of general purpose architectures. Thirdly, the software processor (Intel Pentium M) contains approx. 140 million transistors, neglecting possible additional memory besides the processor's cache. For comparison, the XC3S1000 FPGA has only one million system gates. To overcome this inherent imparity and to provide a fair comparison of the platforms, we will use a very simple metric which is of high practical relevance: the *cost-performance ratio*. We can relate the software and hardware performance with respect to the throughput in point computations and financial costs. The monetary aspect is of central importance when investigating the feasibility of attacks on actual ECC.

We assume the cost of a Pentium M 735 processor and a Xilinx XC3S1000 FPGA to be approximately US$ 220 and US$ 50 per chip, respectively[2]. Additionally, we need housing and peripheral interconnections for each chip. In the case of FPGAs, the realization of a cost-efficient parallel architecture called COPACABANA as described in Chapter 5. Taking the amount of US$ 10,000 as a reference, we determine a corresponding number of workstations

[2]These are market prices for low quantities as in February 2007; prices at high volumes might differ.

k	**Number MPPR cores**	**Performance (pts/sec)**
64	50	$6.66 \cdot 10^7$
80	40	$4.40 \cdot 10^7$
96	40	$3.74 \cdot 10^7$
128	30	$2.16 \cdot 10^7$
160	20	$1.17 \cdot 10^7$

Table 6.5: MPPR performance estimates for an ASIC design (10^7 gates, 500 MHz).

with Pentium M processors. Including housing and additional peripherals, we estimate the cost for a single workstation to be US$ 400 in total. Thus, a cluster of 25 Pentium M workstations costs approximately the same as the COPACOBANA machine with 120 low-cost FPGAs. For the ASIC design from Subsection 6.5.3, we assume a cost of US$ 50 per chip including overhead, which seems to be a fair assumption for the production of large quantities. Hence, for US$ 10,000 we can build an ASIC cluster consisting of 200 ASICs. Due to the relatively high NRE costs of ASICs however, such a design will only be considered when targeting ECC with bit sizes of 128 bit and above and many chips have to be produced. Table 6.5 provides a performance estimate for an ASIC design. Table 6.6 compares the computational power for a US$ 10,000 MPPR-attack in terms of software and hardware cluster implementations. Figure 6.7 provides associated runtimes in days for these platform clusters.

6.6.2 A Note on the Scalability of Hardware and Software Implementations

As indicated by Tables 6.6 and 6.8, the possible speedup of hardware-driven implementations compared to a software-based implementation is not constant. Interestingly, our results show that the gain of special-purpose hardware (FPGA and ASIC) slightly decreases with increasing bit size k of the curves. This fact shows that larger bit sizes can be handled more efficiently with general-purpose hardware such as conventional CPUs. Hence, in case of a larger k, general-purpose processors become slightly more cost-efficient. However, this affects only a constant factor in the relative comparison of the software-based and hardware-based computation. If the

k	**Pentium M cluster**	**FPGA cluster** (COPACOBANA)	**ASIC cluster**
160	$1.92 \cdot 10^6$ (1.0)	$11.2 \cdot 10^6$ (5.8)	$2.34 \cdot 10^9$ (1219)
128	$2.28 \cdot 10^6$ (1.0)	$20.8 \cdot 10^6$ (9.1)	$4.32 \cdot 10^9$ (1895)
96	$3.07 \cdot 10^6$ (1.0)	$39.7 \cdot 10^6$ (12.9)	$7.48 \cdot 10^9$ (2436)
80	$3.41 \cdot 10^6$ (1.0)	$53.7 \cdot 10^6$ (15.8)	$8.79 \cdot 10^9$ (2578)
64	$4.39 \cdot 10^6$ (1.0)	$83.2 \cdot 10^6$ (19.0)	$13.3 \cdot 10^9$ (3030)

Table 6.6: MPPR performance in pts/sec comparison for a US$ 10,000 investment and speed-up factor compared to the software-based implementation (in parentheses).

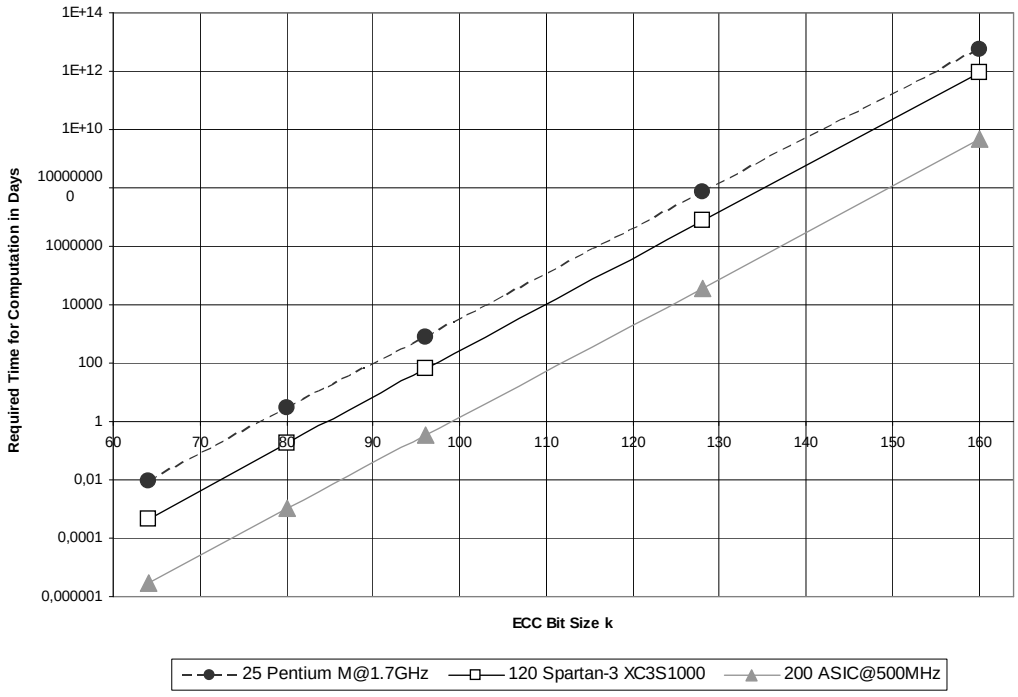


Figure 6.7: MPPR performance in days for a US$ 10,000 investment

software implementation is compared to a potential hardware implementation, there still is a difference of more than one order of magnitude.

Note that the given basic conditions do not include the overhead such as networking infrastructure and additional IT building blocks for the communication. However, we assume that this overhead is approximately the same for both setups. As a consequence, this overhead will not have a major influence on the relative performance of the different settings in software and hardware.

6.6.3 A Security Comparison of ECC and RSA

Manufacturing dedicated hardware for breaking a single RSA-1024 modulus within one year will cost approx. US$ 10 million, if we believe the very optimistic estimate in [ST03b]. A different and more conservative security consideration of RSA has been published in [FKP+05]: The authors assumed the cost of a successful attack within one year to approximately US$ 200 million. According to the wide-spread opinion, RSA-1024 is considered to provide a similar level of security as ECC-163.[3]

The expenses for a successful attack against ECC-163 within one year are equal to the cost

[3]ECC-163 is of particular interest, since 163-bit curves are standardized.

k	**Certicom Est.** [Cer97]	**Pentium M**	**XC3S1000**	**ASIC**
79	146 d	49.0 d	15.3 d	-
89	12.0 y	4.64 y	1.62 y	-
97	197 y	74.7 y	30.7 y	-
109	$2.47 \cdot 10^4$ y	$5.57 \cdot 10^3$ y	$2.91 \cdot 10^3$ y	-
131	$6.30 \cdot 10^7$ y	$1.40 \cdot 10^7$ y	$7.40 \cdot 10^6$ y	$9.34 \cdot 10^4$ y
163	$6.30 \cdot 10^{12}$ y	$1.09 \cdot 10^{12}$ y	$9.15 \cdot 10^{11}$ y	$1.16 \cdot 10^{10}$ y
191	$1.32 \cdot 10^{17}$ y	$2.17 \cdot 10^{16}$ y	$1.89 \cdot 10^{16}$ y	$2.39 \cdot 10^{14}$ y
239	$3.84 \cdot 10^{24}$ y	$4.44 \cdot 10^{23}$ y	$8.62 \cdot 10^{23}$ y	$1.01 \cdot 10^{22}$ y

Table 6.7: Expected runtime on different platforms for the Certicom ECC challenges.

of 7.6 billion COPACOBANA machines and amount to approximately US$ $7.6 \cdot 10^{13}$. Hence, attacks of such dimension are far beyond the feasibility of today's (monetary) capabilities.

Due to the vast amount of computations, ECC with bit size of 128 bit and above can only be efficiently attacked with dedicated ASICs. With the estimate from Subsection 6.5.3, a successful attack against ECC-163 within one year based on $1.16 \cdot 10^{10}$ ASICs will cost approximately US$ $5.8 \cdot 10^{11}$. This amount still is a factor of $58,000$ and $2,900$ higher compared to the estimates for breaking RSA-1024 as described in [ST03b] and [FKP$^+$05], respectively. Although all figures are reasonable estimates for recent hardware architectures with no claim for representing a definite lower bound, we can still assume that an attack on ECC-163 requires significantly more efforts than an attack on RSA-1024.

6.6.4 The ECC Challenges

In 1997, the company Certicom has issued the so-called *ECC challenges* to demonstrate the security of ECC by announcing a list of elliptic curves and associated ECDLP parameters [Cer97]. For numerous bit lengths, a reward for solving such an ECDLP is announced.

For ECC over prime fields $GF(p)$, challenges have been defined for various bit sizes with relevance to cryptography. Table 6.9 shows the expected runtimes for solving the ECDLP for different bit lengths and monetary efforts. Certicom provided estimates for the required number of machine days for solving each challenge based on an outdated Intel Pentium 100 processor. Consequently, we will additionally provide runtimes for our software reference for comparison with the hardware implementations. Table 6.7 depicts how fast our architectures can solve these challenges. A comparison to the estimates given by Certicom in [Cer97] is shown in Table 6.8.

Considering the latest unsolved Certicom challenge over $GF(p)$ ($k = 131$ bit), we estimate the required computational power to be at least $62,000$ COPACOBANA machines ($7.40 \cdot 10^6$ FPGAs) for solving the ECDLP within a year. Unlike $k = 160$, this is also an enormous but not an absolutely unrealistic amount of computational power. With $93,400$ ASICs, this challenge would take one year to finish at a cost of approxi-

k	**Certicom Est.**	**Pentium M**	**XC3S1000**	**ASIC**
79	1.0	2.98	9.54	-
89	1.0	2.59	7.41	-
97	1.0	2.64	6.42	-
109	1.0	4.43	8.49	-
131	1.0	4.50	8.51	78.5
163	1.0	5.78	6.89	94.0
191	1.0	6.08	6.98	90.8
239	1.0	8.65	4.45	44.0

Table 6.8: Relative speed-up compared of different Certicom ECC challenges compared to the Certicom Reference.

k	**Expected runtimes in years for attacks with**			
	US$ 10^5	US$ 10^6	US$ 10^7	US$ 10^8
128	$1.03 \cdot 10^1$	1.03	0.103	0.0103
160	$1.24 \cdot 10^6$	$1.24 \cdot 10^5$	$1.24 \cdot 10^4$	$1.24 \cdot 10^3$
192	$9.64 \cdot 10^{10}$	$9.64 \cdot 10^9$	$9.64 \cdot 10^8$	$9.64 \cdot 10^7$
256	$1.09 \cdot 10^{21}$	$1.09 \cdot 10^{20}$	$1.09 \cdot 10^{19}$	$1.09 \cdot 10^{18}$

Table 6.9: Cost-performance consideration of MPPR attacks with ASICs (10^7 gates, 500 MHz, NRE costs excluded).

mately US$ 5,000,000, excluding NRE costs. Analyzing the last solved challenge with $k = 109$ bits, we can state that about 300 COPACOBANA machines are already sufficient to solve this ECDLP in only 30 days with an expense of about US$ 3 million. For bit lengths exceeding 131 bit however, the only feasible way to solve the ECDLP is the application of dedicated ASIC devices. Assuming the high-performance ASICs from Subsection 6.5.3, we can estimate the runtime for MPPR attacks with respect to different financial background (cf. Table 6.9).

6.7 Conclusion

This contribution fills the gap of the missing analysis in hardware-based cryptanalysis of elliptic curve cryptosystems. Up to now, no actual hardware implementation of an algorithm solving the elliptic curve discrete logarithm problem over prime fields has been realized. In this chapter, we described an efficient architecture for the parallelized Pollard's Rho method (MPPR) to solve the elliptic curve discrete logarithm problem over prime fields. We provided results and a detailed analysis of the first realization of such an architecture in hardware. Besides the hardware implementation, a software implementation was realized on a conventional processor for comparison and for debugging purposes.

The excellent suitability of the MPPR algorithm for hardware was demonstrated by the use of low-cost FPGAs (Xilinx Spartan-3) for a proof-of-concept implementation. Compared to existing architectures for $GF(p)$ arithmetic units, we can state that our AU implementation requires $19.995\mu s$ per point addition at $k = 160$ bit occupying about $2,660$ slices of our FPGA. Thus, our implementation provides a better performance at a smaller area cost with respect to [OP01, OBPV03] and a competitive design concerning the architecture of [DMKP04] when balancing the differences between the varying FPGA types.

For solving the ECDLP on curves of practical interest (160 bit and above) however, FPGA implementations are still too costly. In order to derive an estimate of the security of actual ECC in practice, we provide an estimate for an ASIC implementation based on the outcomes of the FPGA implementation at hand. As a result, ECC turns out to be more secure than commonly believed. The expenses of a successful attack against ECC-163 within one year based on $1.16 \cdot 10^{10}$ ASICs will cost approximately US$ $5.8 \cdot 10^{11}$. Compared to recent (conservative) estimates for a special-purpose hardware attacking RSA-1024 in [FKP$^+$05], this amount still is a factor of $2,900$ larger. However, we could show that low-security standards such as the SECG proposed in [Cer00a, Cer00b] for parameters of length of, e.g., 80 bit and 112 bit, are vulnerable to such efficient FPGA-based attacks.

Chapter 7

Improving the Elliptic Curve Method in Hardware

In this chapter we discuss improvements for hardware implementations of the Elliptic Curve Method (ECM) which is a widely used method for (co-)factorization of integers. Previous architectures of ECM on FPGAs were reported by Pelzl et al. [ŠPK+05] and Gaj et al. [GKB+06a]. In addition to their work, we will optimize their architectures with respect to the low-level arithmetic by employing the DSP blocks of modern FPGAs and also discuss high-level decisions as the choice of alternative elliptic curve representation like Edwards curves. To verify our results in practical experiments, we built a variant of the COPACOBANA cluster with high-performance FPGAs which are capable to support our improved, low-level arithmetic design.

Contents of this Chapter

7.1 Motivation

The Elliptic Curve Method (ECM) was introduced by H. W. Lenstra [Len87] for integer factorization generalizing the concept of Pollard's $p-1$ and Williams' $p+1$ method [Pol74, Wil82]. Although the ECM is known not to be the fastest method for factorization with respect to asymptotical time complexity, it is widely used to factor composite numbers up to 200 bits due to its very limited requirements on memory. As of February 2009, the largest factor, which was revealed by use of ECM, was a 222-bit factor of the special integer $10^{381}-1$ [Zim09].

The most prominent application that relies on the hardness of the factorization problem is the RSA cryptosystem. An attacker on RSA is faced to find the factorization of a composite number n which consists of two large primes p, q. More precisely, the RSA parameter n used in practice to match today's security requirements are 1024 bits and up and hence out of reach for ECM. Up to date, such large bit sizes are preferably attacked with the most powerful methods known so far, such as the *Number Field Sieve* (NFS). However, the complex NFS[1] involves the search of relations in which many mid-sized numbers need to be tested if they are "'smooth"', i.e., composed only of small prime factors not larger than a fixed boundary B. In this context, ECM is an important tool to determine the smoothness of such integers (i.e., if they can be factored into small primes), in particular due to its moderate resource requirements.

The fastest ECM implementations for retrieving factors of composite integers are software-based; a state-of-the-art system is the GMP-ECM software published by Zimmermann *et al.* [FFK+09]. As a promising alternative, efficient hardware implementations of the ECM were firstly proposed in 2006: Šimka *et al.* [ŠPK+05] demonstrated the feasibility to implement the ECM in reconfigurable hardware by presenting a first proof-of-concept implementation. Their results were improved by Gaj *et al.* [GKB+06a] in 2006 who also showed a complete hardware implementation of ECM stage 2. However, the low-level arithmetic in both implementations were only implemented using straightforward techniques within the configurable logic which yet leaves room for further improvements. To fill this gap, de Meulenaer *et al.* [dMGdDQ07] proposed an unrolled Montgomery multiplier based on a two-dimensional pipeline on Xilinx Virtex-4 FPGAs to accelerate the field arithmetic. However, due to limitations in area and the long pipeline design, their design only efficiently supports the first stage of ECM.

In this chapter we describe an alternative ECM architecture for reconfigurable devices. Our focus is to accelerate the underlying field arithmetic of ECM on FPGAs without sacrificing the option to combine both stage 1 and 2 in a single core. Thus, we adopt some high-level decisions, like memory-management and the use of SIMD instructions, from [GKB+06a] which also supports both stages on the same hardware. To improve the field arithmetic, we use a similar technique introduced in Chapter 3 placing fundamental arithmetic functions like adders and multipliers in embedded DSP blocks of modern FPGAs. As a second goal, we develop a cluster architecture for high-performance Virtex-4 FPGAs capable to run our presented ECM implementation in massive parallelism. The presented cluster adopts similar design criteria as the COPACOBANA architecture discussed in Chapter 5. This original version of COPACOBANA provides 120 independent low-cost FPGAs (Xilinx Spartan-3 XC3S1000). The lack of additional memory or high-speed communication facilities supported the simple design approach and provided bare computational resources at low costs. However, the usability of COPACOBANA was limited to applications which do not have high demands to aspects such as availability of local memory and high-speed communications. Moreover, although providing a high density of logic resources, the low-cost Spartan-3 FPGA series only offers rather generic support arithmetic but

[1] The NFS comprises of four steps, the polynomial selection, relation finding, a linear algebra step and finally the square root step. The relation finding step is most time-consuming taking roughly 90% of the runtime. For more information on the NFS refer to [LL93].

no dedicated DSP blocks. More precisely, wide multipliers with more than 160 bits as typically required for computations for public-key crypto (and cryptanalysis) consume large portions of logic when implemented with conventional structures[2] (e.g., cf. the MPPR implementation in generic logic presented in Chapter 6).

Our new COPACOBANA architectures is designed for Virtex-4 FPGA devices[3]. We employ FPGA Virtex-4 SX devices which provide a large number of DSP blocks. Unfortunately, the cost per unit for the largest XC4VSX55 device of this family is disproportional to the gain in logic; hence we opted for the second largest XC4SX35 device and a (slightly) better cost ratio. However, the costs for any Virtex-4 device is still much higher compared to the previously used Spartan-3 FPGAs.

Previous versions of COPACOBANA did not require a fast communication interface since applications performing an exhaustive key search rarely exchange data. However, parameters and results for ECM operations need to be transferred between the cluster and a host-PC. In particular, some operations, like gcd computations, are costly in hardware. More precisely, the generation of elliptic curves and corresponding parameters are preferably precomputed in software and transferred to the FPGAs. This demands for higher bandwidth so that further customization of COPACOBANA should target the communication interface between host and the FPGA cluster. The enhanced design of COPACOBANA based on Virtex-4 FPGAs was developed in joint work with Gerd Pfeiffer [GPPS08].

7.2 Mathematical Background

We now introduce the principle of Lenstra's elliptic curve method [Len87] for factoring composite integers. The core idea of ECM is based on Pollard's $p-1$ [Pol75] and Williams' $p+1$ [Wil82] method, however mapped onto groups over elliptic curves.

7.2.1 Principle of the Elliptic Curve Method

We here review the $p-1$ method to motivate the concept of ECM. Let $k \in \mathbb{N}$ and n be the composite to be factored. Furthermore, let $p|n$ with $p \in \mathbb{P}$, $a \in \mathbb{Z}$ and n be co-prime, i.e., $gcd(a,n) = 1$. Now take Fermat's little Theorem [MvOV96, Fact 2.127] with $a^{p-1} \equiv 1 \bmod p$. The extension by the k-multiple of $(p-1)$ leading to $a^{k(p-1)} \equiv 1 \bmod p$ holds as well since $(a^{p-1})^k \equiv 1^k = 1 \bmod p$. Then, with $e = k(p-1)$ we have

$$\begin{aligned} & a^{k(p-1)} = a^e \equiv 1 \bmod p \\ \Rightarrow \quad & a^e - 1 \equiv 0 \bmod p \\ \Rightarrow \quad & p|(a^e - 1) \end{aligned}$$

[2] In fact, there are a few 18×18 bit multiplier hardcores on XC3S1000 devices but not sufficiently many to support complex cryptographic operations.

[3] At time of development, newer Virtex-5 and low-cost Spartan-3A DSP devices – which also provide embedded DSP blocks – were not yet readily available.

Hence, if $a^e \neq 1 \bmod n$ then we know

$$1 < gcd(a^e - 1, n) < n.$$

In this case, we found a non-trivial divisor of n. However, we are not able to compute $e = k(p-1)$ without the knowledge of p. Thus, we assume that $p-1$ decomposes solely into small prime factors p_i less than a defined bound $B1$ (in this case, $p-1$ is called $B1$-smooth). Thus, we choose e as product of all prime powers p_i^r lower than $B1$ and hope that e is a multiple of $p-1$:

$$e = \prod_{p_i \in \mathbb{P}; p_i < B1} p_i^{\left\lfloor \log_{p_i}(B1) \right\rfloor} \tag{7.1}$$

By computing $d = gcd(a^e - 1, n)$ we finally hope to find a divisor d of n. Note that it is possible that this method fails by returning a trivial divisor.

In 1987, H. W. Lenstra came up with the idea of translating Pollard's method from the groups $\mathbb{Z}_p^*$ and $\mathbb{Z}_n^*$ to the groups of points on elliptic curves $\mathcal{E}(\mathbb{Z}_q)$ and $\mathcal{E}(\mathbb{Z}_n)$ [Len87]. In fact, the group operation in $\mathcal{E}(\mathbb{Z}_n)$ can be defined by using the given addition formulas [Bos85] and replaces the multiplication in $\mathbb{Z}_n$ as used in the $p-1$ method.

Since its proposal, many improvements have been made to ECM. The most fundamental modification was an extension by Brent introducing a second stage to the ECM to improve the success probability of the algorithm [Bre86]. In [Mon87], Montgomery presents further variants of this so-called stage two and introduces a homogeneous parameterization with projective coordinates that avoid inversions modulo n. Furthermore, by using special elliptic curves in so-called Montgomery form and by omitting computations for y-coordinates, group operations of such elliptic curves can be performed with only 6 and 5 modular multiplications per point addition and doubling on $\mathcal{E}$, respectively. In addition to that, the construction of elliptic curves with group order divisible by 12 or 16 [AM93, Mon87] increases the probability of obtaining a group order that is $B1$-smooth.

Now, we briefly review the ECM using elliptic curves $\mathcal{E}$ that are defined by the (homogeneous) Weierstrass Equation:

$$\mathcal{E} : y^2 z = x^3 + axz^2 + bz^3 \tag{7.2}$$

All computations based on this Equation (7.2) are performed using projective coordinates for points $X = (x, y, z)$. Let us assume q to be a factor of n and $|\mathcal{E}(\mathbb{Z}_q)|$ is $B1$-smooth so that e – according to the construction in Equation (7.1) – is a multiple of $q-1$. Note that point multiplication by $|\mathcal{E}(\mathbb{Z}_q)|$ (or multiples of $|\mathcal{E}(\mathbb{Z}_q)|$) returns the point at infinity, e.g., $Q = eP = \mathcal{O}$ for an arbitrary point P and resulting point Q. Recall that the resulting point $Q = \mathcal{O}$ implies a prior impossible division by z_Q so that $z_Q \equiv 0 \bmod q$. Note that we actually perform all point operations in $\mathcal{E}(\mathbb{Z}_n)$ since we do not know q. Hence, we compute $Q = eP$ in $\mathcal{E}(\mathbb{Z}_n)$ and hope to yield a point Q with coordinate $z_Q \neq 0 \bmod n$ but $z_Q \equiv 0 \bmod q$. Then, the factor q of n is

obtained by $q = gcd(z_Q, n)$.

From an algorithmic point of view, we can discover a factor q of n as follows: in the first stage of ECM, we compute $Q = eP$ where e is a product of prime powers $p^i \leq B1$ with appropriately chosen smoothness bounds. The second phase of ECM checks for each prime $B1 < p \leq B2$ whether pQ reduces to the neutral element in $E(\mathbb{Z}_q)$. Algorithm 7.1 summarizes all necessary steps for both stages of ECM. Stage 2 can be done efficiently, e.g., using the Weierstrass form and projective coordinates $pQ = (x_{pQ} : y_{pQ} : z_{pQ})$ again by testing whether $gcd(z_{pQ}, n) > 1$. Note that we can avoid all gcd computations except for a final one at the expense of one modular multiplication per gcd and accumulating all intermediate results in a product modulo n. On this product, we perform a single gcd at the end.

Algorithm 7.1 The Elliptic Curve Method

Input: Composite $n = f_1 \cdot f_2 \cdot \ldots \cdot \ldots \cdot f_n$.
Output: Factor f_i of n.

1: **Stage 1:**
2: Choose arbitrary curve $E(\mathbb{Z}_n)$ and random point $P \in E(\mathbb{Z}_n) \neq \mathcal{O}$.
3: Choose smoothness bounds $B1, B2 \in \mathbb{N}$.
4: Compute $e = \prod_{p_i \in \mathbb{P}; p_i < B1} p_i^{\lfloor \log_{p_i}(B1) \rfloor}$
5: Compute $Q = eP = (x_Q; y_Q; z_Q)$.
6: Compute $d = gcd(z_Q, n)$.
7: **Stage 2:**
8: Set $s := 1$.
9: **for** each prime p with $B1 < p \leq B2$ **do**
10: Compute $pQ = (xpQ : ypQ : zpQ)$.
11: Compute $s = s \cdot zpQ$.
12: **end for**
13: Compute $f_i = gcd(s, n)$.
14: **if** $1 < f_i < n$ **then**
15: A non-trivial factor d is found.
16: return f_i
17: **else**
18: Restart from Step 2 in Stage 1.
19: **end if**

If we regard a single curve only, the properties of ECM are closely related to those of Pollard's $(p-1)$-method that can fail by returning a trivial divisor. The advantage of ECM is apparent with the possibility of choosing another curve (and thus group order) after each unsuccessful trial, increasing the probability of retrieving factors of n. If the final gcd of the product s and n satisfies $1 < gcd(s, n) < n$, a factor is found. Note that the parameters $B1$ and $B2$ control the probability of finding a divisor q. More precisely, if the order of P factors into a product

of coprime prime powers (each $\leq B1$) and at most one additional prime between $B1$ and $B2$, the prime factor q is discovered. It is possible that more than one or even all prime divisors of n are discovered simultaneously. This happens rarely for reasonable parameter choices and can be ignored by proceeding to the next elliptic curve. The runtime of ECM is given by

$$T(q) = e^{(\sqrt{2}+o(1))\sqrt{\log q \log\log q}}$$

operations, thus, it mainly depends on the size of the factors to be found and not on the size of n [Bre86]. However, remark that the underlying operations are computed modulo n so that the runtime of each single operations still depends on the size of n. To generate elliptic curves, we start with an initial point P and construct an elliptic curve such that P lies on it. As already mentioned, curves with particular properties (e.g., a group order divisible by 12 or 16) can increase the probability of success. The construction of such curves and corresponding ECM parameters, for example the parametrization[4], is more thoroughly discussed in [ZD06, BBLP08].

7.2.2 Suitable Elliptic Curves for ECM

Already identified by Chudnovsky and Chudnovsky in [CC86] the choice of an underlying algebraic group structure in which the group operations are most time efficient is a central issue for the runtime of ECM. Typically, the Weierstrass form is the most popular flavor for elliptic curves. In the last years, however, most designs for ECM preferably use elliptic curves in Montgomery's form (as shown in Equation (7.3)) since it allows to compute points with simplified formulas, e.g., by only using the x- and z-coordinates. The utilization of such elliptic curves for ECM is suggested in [Mon87] which are defined by the curve equation

$$\mathcal{E}_M : bx^2z = x^3 + ax^2z + xz^2. \tag{7.3}$$

Note that Montgomery curves always have an order divisible by 4 what leads to the observation that not every curve in Weierstrass form can be transformed to Montgomery form.

Formulas for point addition on Montgomery curves can be given that do not involve any computations of y; hence it is not possible to distinguish between (x, y) and $(x, -y)$. Consequently, we need additional information when constructing an addition chain for point multiplications which is often denoted as "differential sum" $P - Q$ of the two points P, Q. The addition of $P + Q$ can be determined from P, Q and $P - Q$ using 6 multiplications[5] and 5 multiplications for point doubling $2P$ assuming that the fixed quantity $(a + 2)/4$ is precomputed. Note that it is also possible to reduce combined addition formulas for both operations to 10 multiplications by fixing $z_P = 1$ and even 9 multiplications when additionally choosing a small x_P, e.g., $x_P = 2$.

[4] A popular parametrization method for Montgomery curves with a group order divisible by 12 is due to Suyama [ZD06].

[5] Note that we do not distinguish between multiplications and squarings since we will not implement a dedicated hardware squaring unit. This limits a priori our requirements on hardware resources. Besides, we do not regard modular additions in any runtime consideration since they can be done in parallel to modular multiplications.

Addition:

$$\begin{aligned} x_{P+Q} &\equiv z_{P-Q}[(x_P - z_P)(x_Q + z_Q) + (x_P + z_P)(x_Q - z_Q)]^2 \bmod n \\ z_{P+Q} &\equiv x_{P-Q}[(x_P - z_P)(x_Q + z_Q) - (x_P + z_P)(x_Q - z_Q)]^2 \bmod n \end{aligned} \tag{7.4}$$

Doubling:

$$\begin{aligned} 4x_P z_P &\equiv (x_P + z_P)^2 - (x_P - z_P)^2 \bmod n \\ x_{2P} &\equiv (x_P + z_P)^2 (x_P - z_P)^2 \bmod n \\ z_{2P} &\equiv 4x_P z_P[(x_P - z_P)^2 + 4x_P z_P(a+2)/4] \bmod n \end{aligned} \tag{7.5}$$

Since the formulas require availability of P,Q and $P-Q$, Montgomery proposed a suitable point multiplication method denoted as Montgomery ladder [Mon87] which was later also extended to other abelian groups by Joye and Yen [JY03]. Let $(mP, (m+1)P)$ be an initial state of the point multiplication, where m is a scalar and P an arbitrary point on $\mathcal{E}_M$, then the Montgomery ladder determines in sequence either $(2mP, (2m+1)P)$ or $((2m+1)P, (2m+2)P)$ dependent if m is to be raised by a bit 0 and 1, respectively. For more information regarding the Montgomery ladder, see [Mon87, JY03].

Recently, an alternative form of elliptic curves suggested by Edwards [Edw07] was proposed by Bernstein *et al.* for use with ECM [BBLP08]. Twisted Edwards curves with homogeneous inverted coordinates can be obtained by the following formula:

$$\mathcal{E}_E : (x^2 + ay^2)z^2 = x^2y^2 + dz^4 \tag{7.6}$$

Compared to ECM with Montgomery curves, a variant of such Edwards curves with twisted inverted coordinates [BBJ+08] are reported to lead to even shorter computation times in software [BBLP08]. Although more operations are required in total, a gain in performance is achieved by choosing input and curve twist parameters with small heights and a more efficient method for point multiplication compared to the Montgomery ladder. The point addition on (specifically constructed) Edwards curves takes only 6 full-size multiplications and 6 small-height multiplications. A point doubling on $\mathcal{E}_E$ can be done with 7 full-size and 2 small-height multiplications. Besides an adaption of the original GMP-ECM software for such Edwards curves [BBLP08], the authors also implemented ECM very efficiently on modern graphics cards using the CUDA framework [BCC+09]. The formulas for group operations $P + Q$ and $2P$ on Edwards curves (in inverted twisted coordinates) are given as follows [BBJ+08]:

Addition:

$$\begin{aligned} x_{P+Q} &\equiv (x_P x_Q y_P y_Q + dz_P^2 z_Q^2)(x_P x_Q - a y_P y_Q) \bmod n \\ y_{P+Q} &\equiv (x_P x_Q y_P y_Q - dz_P^2 z_Q^2)[(x_P + y_P)(x_Q + y_Q) - x_P x_Q - y_P y_Q] \bmod n \\ z_{P+Q} &\equiv z_P z_Q (x_P x_Q - a y_P y_Q)[(x_P + y_P)(x_Q + y_Q) - x_P x_Q - y_P y_Q] \bmod n \end{aligned} \tag{7.7}$$

Doubling:

$$\begin{aligned} x_{2P} &\equiv (x_P^2 + ay_P^2)(x_P^2 - ay_P^2) \bmod n \\ y_{2P} &\equiv [(x_P + y_P)^2 - x_P^2 - y_P^2](x_P^2 + ay_P^2 - 2dz_P^2) \bmod n \\ z_{2P} &\equiv (x_P^2 - ay_P^2)[(x_P + y_P)^2 - x_P^2 - y_P^2] \bmod n \end{aligned} \tag{7.8}$$

Next, we will elaborate on an optimal instruction schedule for point addition and point doubling on Montgomery and Edwards curves. Let e be a scalar used in the computation $Q = e \cdot P$ of ECM stage 1 with n-bits and binary representation $e = \sum_0^{n-1} e_i 2^i$ where $e_i \in \{0, 1\}$. With Montgomery curves, we need to compute $9n$ full-width multiplications for each bit e_i assuming that $z_P = 1$ and x_P is small. Stage 2 does not allow any preliminary simplifications so that we need compute $11m$ multiplications where m denotes the number of accumulated point operations depending of the continuation method for ECM stage 2 (cf. Algorithm 7.1).

With Edwards curves, it is possible to use a state-of-the-art point multiplication algorithm with addition chains instead of the Montgomery ladder[6]. Such advanced methods for point multiplication do not require a point addition for each bit e_i of the scalar what reduces the overall operation count compared to the Montgomery ladder. The point multiplication selected in [BBLP08] is the signed sliding window method. Note that – depending on the algorithm – the determination of an optimal addition chain is not always straightforward so that the implementation of a corresponding hardware circuit can be costly. Thus, many concepts for point multiplication with optimal addition chains (which can significantly accelerate computations in software) *cannot* easily be applied in hardware[7]. For Edwards curves we will thus assume the (straightforward) sliding window technique for which efficient hardware implementations are known [SBG+03]. Summarizing, computations on Edwards curves take a constant number of $(7 + 2r)$ modular multiplications per point doubling and $(6 + 6r)$ modular multiplications for a point addition where r is a factor representing the clock cycle ratio of a small height multiplication with respect to a full multiplication[8].

At this point, we need to know the complexity functions of the individual operations in order to determine the ratio r for small-height multiplications with respect to a standard multiplications. Thus, we elaborate on this further in Section 7.3.3.

7.3 Implementing an ECM System for Xilinx Virtex-4 FPGAs

Due to the broad coverage of previous work on FPGA-based ECM implementations [ŠPK+05, GKB+06a, dMGdDQ07] we here focus on optimizations of the fundamental arithmetic functions. Again, we exploit the embedded features available in modern FPGAs.

[6] Many publications on addition chains are available; for a survey see [HMV04, ACD+05].

[7] A possible solution to this problem could be to precompute addition chains in software and download them on demand to the hardware. However, we will not consider this in this thesis.

[8] For our hardware architecture based on an m-bit digit-multiplier, we assume that *small* parameters as introduced in [BBLP08] can be represented as single m-bit digit, i.e., $0 \geq j < 2^m - 1$.

Algorithm 7.2 Modular multiplication with quotient pipelining [Oru95]

Input: $0 \leq A, B \leq 2\tilde{M}$, $M'' = \frac{\tilde{M}+1}{2^k}$, word width k
Output: $S_{\beta+1} \equiv A \cdot B \cdot R^{-1} \pmod{M}$ and $0 \leq S_{\beta+1} < 2\tilde{M}$
1: $S_0 = 0$, $q_0 = 0$
2: **for** $i = 0$ to β **do**
3: $\quad q_i \equiv S_i \bmod 2^k$
4: $\quad S_{i+1} = \frac{S_i}{2^k} + q_i M'' + b_i A$
5: **end for**
6: **return** $S_{\beta+1}$

We *cannot* reuse directly the presented ECDSA core in Chapter 3, since the modular arithmetic supports a *special* modulus only. Hence, to support arbitrary moduli, we decided to implement a high-radix Montgomery multiplication algorithm [Mon85] and took efforts that all basic arithmetic functions are handled by DSP blocks.

7.3.1 A Generic Montgomery Multiplier based on DSP Blocks

Montgomery's original multiplication method (cf. Section 4.4.1) has been improved by many publications available in open literature. Orup proposed rewriting Montgomery's algorithms for multiplication of which one simplifies the quotient handling [Oru95]. Straightforward quotient handling is particularly important since it allows consecutive arithmetic operations in the DSP blocks with realignment of operands. Orup's improvements only demand for word-wise (i.e., k-bit) multiplications, additions and shifting. All operations are natively supported by all DSP blocks of Virtex-4 FPGAs. Hence, using Orup's variant all arithmetic operations can be performed by sequential instructions in DSP blocks (by issuing a different *operation mode* or *opmode* to the DSP block) without need for additional resources in the configurable logic.
For our hardware design of the ECM, we will thus use Orup's *Modular Multiplication with Quotient Pipelining* as given in Algorithm 7.2. Note that Suzuki [Suz07] already promoted the use of Orup's algorithms with DSP blocks. However, Suzuki used static opmodes and thus performed all additions in the configurable logic instead of using the embedded adder of the DSP block. Hence, Suzuki's approach turns out to prove more costly in terms of resources and clock cycles.

We will now shortly revise Orup's modification to the Montgomery multiplication as introduced in Section 4.4.1. Let $M > 3$ be a modulus such that M and 2 are coprime. We fix the delay parameter $d = 0$ in Orup's original notation simplifying our implementation. Since the multiplier in DSP blocks has a static width, we choose the word width $k = 17$. Let β be the maximum number of words required to represent all operands, determined by $\tilde{M} = M' \cdot M$ and $M' \equiv -M^{-1} \bmod 2^k$ with the constraint that $4\tilde{M} < 2^{k \cdot \beta}$. We define R to be an integer $R \equiv 2^{k\beta} \bmod M$. Let A and B be factors in radix 2^k representation such that $0 \leq A, B \leq 2\tilde{M}$. The result of Algorithm 7.2 is $S_{\beta+1} \equiv A \cdot B \cdot R^{-1} \bmod M$ where $0 \leq S_{\beta+1} < 2\tilde{M}$.

We now present our modified high radix multiplication algorithm that can be implemented almost solely with DSP blocks. In addition to Orup's description, we define the number of required DSP block δ as additional parameter. For small multiplier configurations, δ can be chosen equal to the number of blocks β. Note that δ should consider the maximum number of DSP blocks available in a single column[9]. In such case, each 17×17-bit multiplication is performed by a dedicated DSP block. In order to realize larger multipliers, a cascade of DSP blocks can be used to process individual limbs of a multi-precision multiplication in several iterations. In such a case, let $n = \delta r$ with $r \in \mathbb{N}$ and r denotes the number of iterations for each DSP block. This iterative approach is also followed in [Suz07]. Suzuki's multiplier comprises of $\delta = 17$ DSP blocks that can handle 512-bit up to 2048-bit moduli dependent on the number of iterations $r \in \{2, 4, 6, 8\}$. Since we assume the ECM to handle relatively small moduli (e.g., we target composite integers n that are smaller than 200-bit), we focus solely on the special case with $\delta = \beta$.

To use DSP blocks, all operands and intermediate results need to be handled in radix 2^k representation with data path width of $k = 17$. The input parameters A and B, and the resulting parameter S_{n+1} are of the same order of magnitude (i.e., $S_{\beta+1} < 2^{k \cdot \beta}$) such that the result can be used as input parameter again. In addition to that, all intermediate and final results S_i and $S_{\beta+1}$ are less than $2\tilde{M}$ so that all parameters can be expressed with a maximum number of blocks β.

Lemma 7.1 *Given an operand with $h = \lfloor log_2(M) \rfloor + 1$ bits, the number of blocks β for the Montgomery multiplication method given by Algorithm 7.2 with word size k is determined by*

$$\beta = \left\lceil 1 + \frac{h+2}{k} \right\rceil .$$

Proof: Note that $\tilde{M} = (M' \bmod 2^k)M$ and $4\tilde{M} < 2^{k\beta}$. Thus, the maximum possible value of $\tilde{M}$ is $\tilde{M}_{max} = (2^k - 1)(2^h - 1)$. Hence,

$$\tilde{M} < \tilde{M}_{max} + 1 \Leftrightarrow \tilde{M} < 2^{k+h} - 2^k - 2^h + 2$$

Since $(-2^k - 2^h + 2) < 0$ holds, we can assume

$$\tilde{M} < 2^{k+h} \Leftrightarrow 4\tilde{M} < 2^{k+h+2}$$

and thus

$$\Rightarrow k + h + 2 = k \cdot \beta \Leftrightarrow \beta = \left\lceil \frac{k+h+2}{k} \right\rceil$$

Lemma 7.2 relates the number of DSP blocks to a configuration supporting integer multiplications for operands up to h bits. Note that we allocate a DSP block for each block β of the input.

[9]Dependent on the device size, a single column of a Virtex-4 FPGA comprises of 32 or 48 DSP blocks, respectively.

Algorithm 7.3 Modular multiplication with quotient pipelining optimized for DSP blocks

Input: Multiplicand $A = \sum_{i=0}^{\beta-1} a_i \ (2^k)^i$; multiplier $B = \sum_{i=0}^{\beta} b_i \ (2^k)^i$; $0 \leq A, B \leq 2\tilde{M}$; $M'' = \sum_{i=0}^{\beta-1} m_i 2^k i$; $0 < M'' < 2^h$; word width $k = 17$; number of blocks β.
Output: $S_{\beta+1} \equiv A \cdot B \cdot R^{-1} \pmod{M}$; $S_i = \sum_{j=0}^{\beta-1} S_{i,j} \ (2^k)^j$; $0 \leq S_{\beta+1} < 2\tilde{M}$.

```
S_{0,j} ← 0
q_0 ← 0
for i = 0 to β do
    q_i ← S_{i,0} (mod 2^k)
    S_{i+1,0} ← [S_{i,1} (mod 2^k) + q_i · m_0 + b_i · a_0] (mod 2^k)
    for j = 1 to β − 1 do
        S_{i+1,j} ← [S_{i,j+1} (mod 2^k) + q_i · m_j + b_i · a_j + ⌊S_{i+1,j−1} / 2^k⌋ ] (mod 2^k)
    end for
    S_{i+1,β} ← [q_i · m_β + b_i · a_β + ⌊S_{i+1,β−1} / 2^k⌋ ] (mod 2^k)
end for
return S_{β+1}
```

Lemma 7.2 *Given a number of $\delta > 2$ DSP blocks for a single-pass Montgomery multiplication method as in Algorithm 7.2 with word size k, the maximum width h of the operands is given by*

$$h = k(\delta - 1) - 2.$$

Proof: In a single-pass computation ($\delta = \beta$), the trivial upper bound for $\delta > 2$ DSP blocks is $k\delta$ bits. However, from the assumptions of Algorithm 7.2 we need to enforce that we can also represent $4\tilde{M}$ with β blocks. In other words,

$$4\tilde{M} < 2^{k\beta} \Rightarrow \tilde{M} < 2^{k\delta-2}.$$

Recalling that $\tilde{M} = (M' \bmod 2^k)M < 2^{k+h}$ holds, we can simply extract the exponents yielding

$$k + h = k\delta - 2 \Rightarrow h = k(\delta - 1) - 2.$$

Consequently, an architecture based on 10 DSP blocks can support moduli up to 151 bits and one with 16 DSP units up to 253 bits, respectively. A version of the *Modular Multiplication with Quotient Pipelining* algorithm rewritten for use with DSP blocks is shown by Algorithm 7.3. The implementation of algorithm 7.3 is depicted in Figure 7.1. Each DSP block successively performs and accumulates $a_i \cdot b_j$, $m_i \cdot q_j$ and $S_{i,j+1}$ with a total latency of 5 clock cycles (one per accumulation operation and two for additional input/output registers). This has to be repeated β times for all blocks a_i for $i = 0, \ldots, \beta - 1$ and an additional final round. Hence, the total runtime of an full h-bit multiplication with a pipeline consisting of δ DSP blocks is

$$C_{\mathsf{MUL}}(h) = 5(\beta + 1) + \delta + 1 = 5(2 + \frac{h+2}{k}) + 2 + \frac{h+2}{k} = 12 + \frac{6(h+2)}{k}$$

As an example, a full 151-bit multiplications using $\delta = 10$ DSP blocks takes $T(151) = 66$ clock cycles. Note that this algorithm can also be reused repeatedly by using results as inputs and

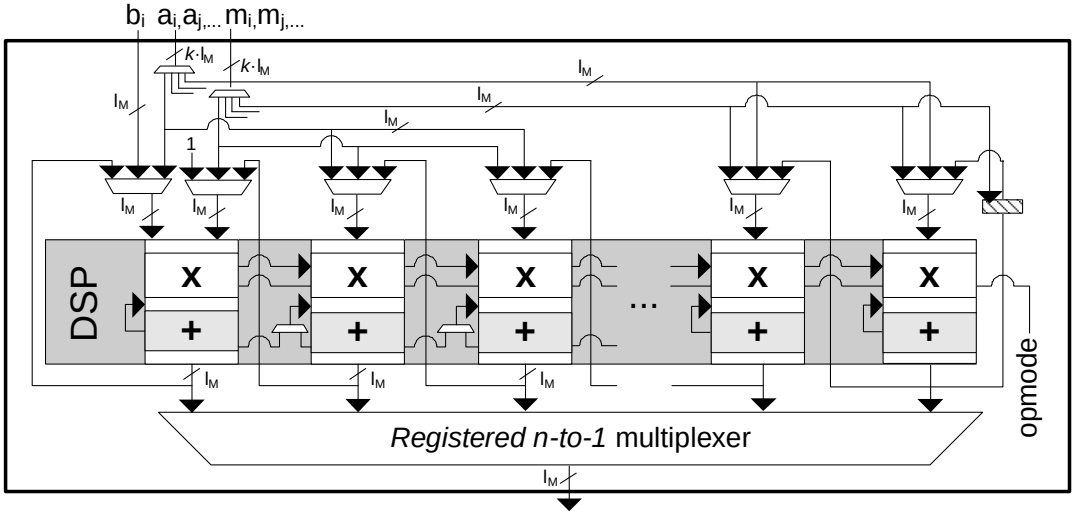


Figure 7.1: Generic Montgomery multiplier designed for use with Virtex-4 DSP blocks.

thus is not limited to the use with ECM. As already mentioned, our modular multiplier is capable to perform accumulation *within* the DSP block by dynamic opmode manipulation and thus computes the final product with a reduced number of cycles when comparing our design to [Suz07].

7.3.2 Choice of Elliptic Curves for ECM in Hardware

We now determine the optimal choice for elliptic curves for a hardware implementation based on the multiplier introduced above. Our main optimization goal for the architecture is high performance since the reduction on logical elements is coming for free due to the extensive use of embedded DSP blocks (cf. Chapter 3).

Furthermore, we do not want to spend additional hardware on modular inversion so that elliptic curves in Edwards or Montgomery form for fast explicit formulas in the projective, inversion-free domain are preferable. According to [BBLP08], ECM with Edwards curves can be faster when exploiting that single-digit multiplications (SDMUL) (i.e., one factor A is of full size and the other only a single digit b_i) allow faster computations. We can enable our multiplier design for SDMULs by introducing an additional interrupt in the finite state machine such that the first digit a_i is processed only. Then, the SDMUL requires 5 cycles for a DSP for the computations of the first a_i, 5 cycles for a finalizing round, and additional cycles from the pipeline δ. This leads to a time complexity (in clock cycles) given by the function

$$C_{\mathsf{SDMUL}}(h) = 10 + \delta + 1 = 12 + \frac{h+2}{k}$$

for a single-digit multiplication. The ratio between full and single-digit multiplications can be thus obtained by $\lfloor C_{\mathsf{MUL}}/C_{\mathsf{SDMUL}} \rfloor$. Assuming the most practical bit range $49 < h < 304$ for

hardware-based ECM (cf. parameter sizes in previous work), three single-digit multiplications can be performed in the same time as one multiplication with two full-width operand.
The modular adder and subtracter is based on the same unit proposed in Section 3.4.2. The time complexity for this component is even less than that of a single-digit multiplication since it processes words at double-precision (e.g., $2k = 34$). More precisely, the number of clock cycles for modular addition or subtraction based on two DSP units with a pipeline length of 4 cycles is given by

$$C_{\mathsf{ADD/SUB}}(h) = 4 + \left\lceil \frac{h}{2k} \right\rceil .$$

For simplicity, we will assume $C_{\mathsf{ADD/SUB}} < C_{\mathsf{SDMUL}}$ since three modular additions or subtractions can be executed in parallel to one full-width multiplication.

We now identify the optimal number of arithmetic units for fast instruction scheduling for both point doubling and addition. Based on our experiments, we found the use of two parallel multiplier and one combined adder/subtracter optimal for both Edwards and Montgomery curves. More parallelism in terms of arithmetic units turned out to be not beneficial due to data dependencies. In other words, adding further arithmetic units to the system resulted in computational idle times since the components need to wait until previous computations has finished. An example of such a dependency is given by the computation of y_{2P} (Edwards coordinates) where two multiplications ($T_1 = z_P^2$, $T_2 = 2dT_1$) followed by another two subtractions ($T_3 = P_1 - P_2 - T_2$) and a final multiplication ($T_4 = P_3T_3$) need to performed in sequence since all inputs T_i are dependent on previous results T_{i-1} (note that we do not care for P_j values since we assume they are computed concurrently to T_i operations).

Next, we focus on the instruction scheduling of the explicit formulas for point addition and point doubling. For the given setup with two multipliers and one adder/subtracter, we present a combined sequence of operations for an interleaved point doubling and point addition based on the formulas in Montgomery and Edwards form. The combination of both point operations into a single sequence of instructions allows further reduction of computational idle times. This is particularly beneficial when using the Montgomery ladder where point addition and point doubling follow in sequence in each step. On the contrary, the more advanced point multiplication techniques as for use with Edwards curves compute a more unpredictable sequence of point doublings and point additions. However, this is not an issue: to perform a sole point doubling independent of the point addition, the instruction sequence can be terminated right after the last arithmetic operation of the point doubling has finished (additional computations for point addition performed up to this point can operate on dummy inputs in such a case). We optimized the instruction flow to avoid a performance penalty by the interleaved processing of both operations. Table 7.1 shows the operation schedule for a single step on the Montgomery ladder (i.e., combined point addition and point doubling) in the case $z_{P-Q} = 1$. For the instruction schedule, we assumed that at least two addition/subtractions can be issued in parallel to one multiplication. Note that optimal memory and register allocation is not included in the listing

#	ADD/SUB	MUL # 1	MUL # 2
1.1	$AD_1 = x_P + z_P$		
1.2	$AD_2 = x_P - z_P$		
2.1	$AD_3 = x_Q + z_Q$	$D_1 = AD_2^2$	$D_2 = AD_1^2$
2.2	$AD_4 = x_Q - z_Q$		
3	$D_3 = D_2 - D_1$	$A_1 = AD_2 \cdot AD_3$	$A_2 = AD_4 \cdot AD_1$
4.1	$A_3 = A_1 + A_2$	$x_{2P} = D_1 \cdot D_2$	$D_4 = D_3 \cdot C_{(a+2)/4}$
4.2	$A_4 = A_1 - A_2$		
5	$D_5 = D_1 + D_4$	$x_{P+Q} = A_3^2$	$A_5 = A_4^2$
6		$z_{P+Q} = A_5 \cdot x_{P-Q}$	$z_{2P} = D_3 \cdot D_5$

Table 7.1: Combined point addition and doubling ($2P$ and $P + Q$) on Montgomery curves for the case $z_{P-Q} = 1$.

for better readability. Instead we denote A_i as a register used for point addition, D_i one for point doubling and AD_i is used for both operations.

Similarly, Table 7.2 presents combined formulas for point doubling and subsequent point addition for Edwards curves with inverted, twisted coordinates. Assuming practical bit lengths h with $49 < h < 304$, we allocated in our model three time slots for modular additions/subtractions or single-digit multiplications that can be individually issued during the runtime of one full multiplication.

Based on our model, the combined point addition and doubling based on two multipliers and one adder unit takes $9C_{\mathsf{FULL}}$ cycles. The sequence for Montgomery coordinates requires $5C_{\mathsf{FULL}} + 2C_{\mathsf{ADD/SUB}}$ cycles. Although more expensive in terms of the total number of operations, the point multiplication methods applicable with Edwards curves perform mainly point doublings that take $5C_{\mathsf{FULL}}$ cycles. Evaluating minimum requirements so that a point multiplication method is more efficient than the Montgomery ladder, we obtain a ratio between point doublings (DBL) and point addition (ADD) of $\#DBL/\#ADD > 13.7$ for practical operand lengths with $h > 100$ bits. Thus, a beneficial ratio in favor for Edwards curves is only likely for very long scalars e of several thousands bits with large window sizes. Since ECM parameters for hardware-based implementations need to be smaller due to the limited amount of available on-chip memory[10], it is unlikely that we can reach this threshold with FPGAs.

Hence, although Edwards curve were reported to be more efficient in software, we finally select Montgomery curves for our hardware implementation, mostly due to memory restrictions.

[10]Recall that only strictly limited bound $B1 = 960$ and $B2 = 57000$ were used in previous hardware implementations [ŠPK+05, GKB+06a]. Larger bounds will exceed the available memory for storing prime tables on the FPGA required for the second stage of ECM.

#	ADD/SUB	MUL #1	MUL #2
1	$D_6 = x_P + y_P$	$\mathbf{D_1 = x_P^2}$	$\mathbf{D_2 = y_P^2}$
2.1		$D_3 = a \cdot D_2$	$\mathbf{D_6 = D_6 \cdot D_6}$
2.2	$D_4 = D_1 + D_3$		
2.3	$D_5 = D_1 - D_3$		
3.1	$D_6 = D_6 - D_1$	$\mathbf{x_{2P} = D_4 \cdot D_5}$	$\mathbf{y_{2Q} = z_P^2}$
3.2	$D_6 = D_6 - D_2$		
4.1		$y_{2P} = 2d \cdot y_{2P}$	$\mathbf{z_{2P} = D_5 \cdot D_6}$
4.2	$y_{2P} = D_4 - y_{2P}$		
5.1		$A_1 = z_{2P} \cdot z_Q$	$\mathbf{y_{2P} = D_6 \cdot y_{2P}}$
5.2		$A_3 = x_P \cdot x_2$	
6.1	$A_7 = x_Q + y_Q$	$A_4 = y_P \cdot y_Q$	$\mathbf{A_2 = A_1^2}$
6.2	$A_8 = x_P + y_P$	$A_6 = a \cdot A_4$	
6.3		$A_8 = A_7 \cdot A_8$	
7.1	$A_6 = A_3 - A_6$	$A_2 = d \cdot A_2$	$\mathbf{A_5 = A_3 \cdot A_4}$
7.2	$x_{2P+Q} = A_5 + A_2$		
8.1	$A_8 = A_8 - A_3$	$\mathbf{x_{2P+Q} = x_{P+Q} \cdot A_6}$	$\mathbf{z_{2P+Q} = A1 \cdot A_6}$
8.2	$A_8 = A_8 - A_4$		
8.3	$y_{2P+Q} = A_5 - A_2$		
9		$\mathbf{y_{2P+Q} = y_{P+Q} \cdot A_8}$	$\mathbf{z_{2P+Q} = z_{2P+Q} \cdot A_8}$

Table 7.2: Combined point addition and doubling ($2P$ and $P+Q$) in inverted, twisted Edwards coordinates. Bold-faced operations denote full-size modular multiplications with $h \times h$ bits, all other operations take at most a third of the runtime of the full multiplication. The double line marks a possible early termination point of the instruction sequence to perform a sole point doubling.

7.3.3 Architecture of an ECM System for Reconfigurable Logic

In this section, we develop an architecture based on the modular multiplier proposed in Section 7.3.1. In our implementation, we adopt the high-level design including the global instruction set and memory management presented in [GKB+06a]. Hence, we developed a similar multi-core design which can support both stage 1 and 2 of ECM. Each core is designed to compute the group operations on the elliptic curves, i.e., supplementary operations like gcd computations and parameter generation for the elliptic curves are performed off-chip. Thus, an ECM core consists of an arithmetic unit for modular multiplication and addition and control logic for computing a point multiplication step. For both stages, all instructions are kept in a global instruction

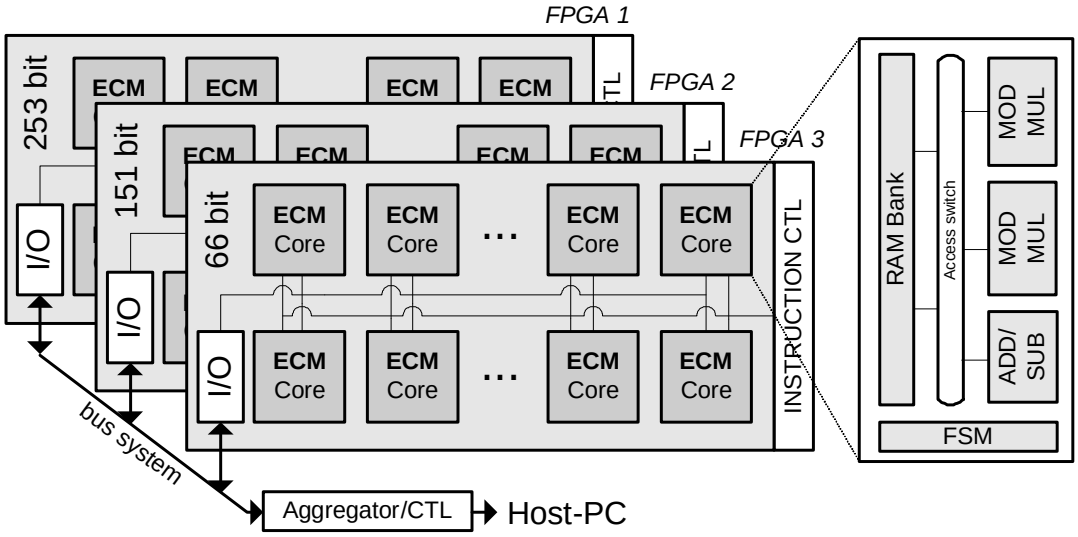


Figure 7.2: Each FPGA contains an individual ECM system with multiple ECM cores operating in SIMD fashion. Factorization of different integer bit lengths can be supported by different FPGAs.

memory and issued to all cores in parallel, i.e., the individual cores operate in Single Instruction Multiple Data (SIMD) fashion. The global control also includes a few additional ROM tables for stage 2 to drive the *standard continuation* method proposed by Montgomery [Mon87]. We considered the improvements for the standard continuation method in hardware (e.g., bit tables for prime representation) proposed in [GKB+06a] as well. Figure 7.2 depicts the architecture of the ECM system implemented in this work.

7.4 A Reconfigurable Hardware Cluster for ECM

The hardware platform for the presented ECM system above requires the availability of FPGA devices with embedded DSP blocks and a communication interface providing sufficient performance to exchange required parameters. Note that arithmetic computations prevail so that the hardware platform still does not need to support very high performance data transfers. Only the elliptic curve parameter a, the base point $P = (x_P, z_P)$ and the result z_Q need to be exchanged. Though, significantly more bandwidth is necessary than in the brute-force scenarios as described in Chapter 5.

The original COPACOBANA cluster combined 120 Spartan-3 XC3S1000 FPGAs distributed along 20 plug-in modules in a single backplane with 6 FPGAs per module. Note that this approach is not optimal with a binary address encoding since the required address space of 5 bit for module selection is not fully utilized. As a remedy, the new design carries 16 plug-in modules each hosting 8 FPGAs. Instead of Spartan-3 FPGA, we selected Virtex-4 devices which

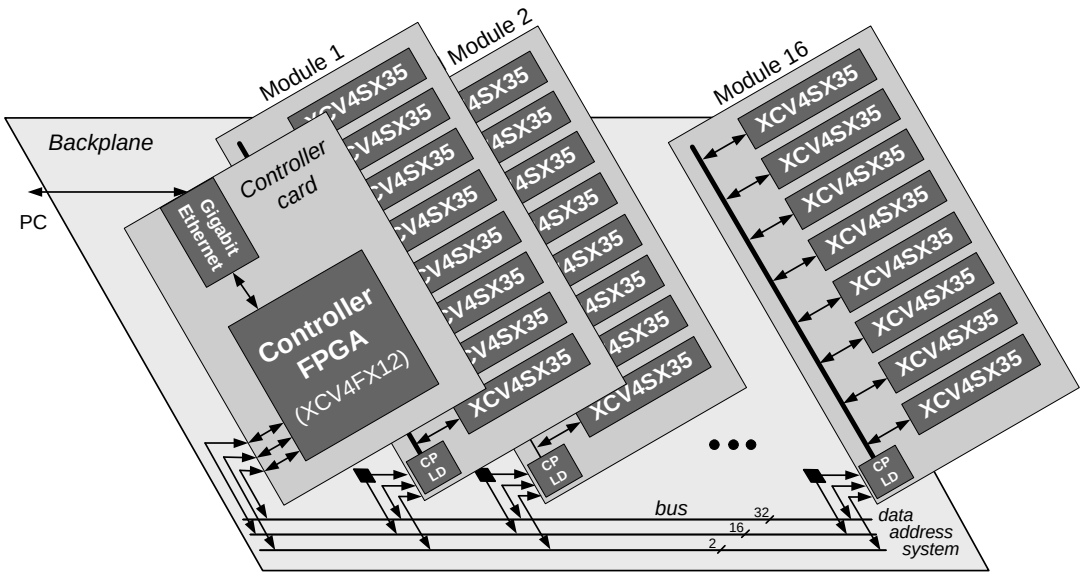


Figure 7.3: Architecture of modified COPACOBANA cluster based on Virtex-4 SX35 FPGAs.

provide embedded DSP blocks[11]. We stick to the shared single master bus (32 bit) and provided an additional 16 bit address bus on a backplane. As a design option, the bus architecture can be switched to a 32 serial point-to-point connections so that four FPGAs share a serial line using differential I/O. In this setup, each serial link can support throughputs up to 2.5 GBit per second (RocketIO transceivers) so that each FPGA of the system can transfer data with up to 625 MBit/s.

We selected a Xilinx Virtex-4 SX35-10 in FF668 package (27×27 mm) that provided the best tradeoff with respect to the number of embedded DSP blocks and financial costs. The SX class of Virtex-4 FPGAs are optimized for arithmetic operations and thus provide a large number of DSP blocks. Note that we can alternatively deploy other Virtex-4 devices with the same 27×27 mm footprint like the Virtex-4 LX[12] or Virtex-4 FX[13]. The architecture of the new FPGA-cluster (COPACOBANA with Virtex-4 FPGAs) is depicted in Figure 7.3.

On the same plug-in module, all eight FPGAs are connected to a two CPLDs (CoolRunner-II) which act as bus driver and communication bridge between the shared bus on the backplane and the local bus on the module. The shared 32 bit bus on the backplane is driven and controlled by a further Virtex-4 FX FPGA at 20MHz placed on a separate controller module. This FPGA also integrates a full-blown TCP/IP stack running on the integrated PowerPC hardcore so that the FPGA cluster can establish a connection to a host computer via Gigabit Ethernet. We designed the whole communication system as a three-tier architecture allowing a target application to

[11]The design decision in favor for Virtex-4 FPGAs was taken in March 2007. At that time, the more cost-efficient Spartan-3A DSP devices were not available and Virtex-4 devices were the only option when DSP blocks are required.

[12]LX devices are designed to provide large amounts of generic logic.

[13]Virtex-4 FX FPGAs come with an integrated PowerPC core and support a larger variety of I/O modes

implement a Hierarchical Communication Model (HCM). However, in practical experiments we found the three tier HCM as not optimal for high throughputs. With our current platform implementation we are faced with the effect that host-to-FPGA transfers are the performance bottleneck and strictly limit the number of possible ECM operations.

The improved performance of the Virtex-4 FPGAs comes in line with an increased energy consumption per chip. The required power per chip was estimated based on assumption that cryptanalytical applications are likely to utilize all available hardware resources. According to these requirements, the power distribution system can supply a maximum of 10W to each FPGA. Consequently, we chose a global DC power supply unit providing 125A output at 12V. The corresponding 1500W of output power are distributed by the backplane to all plug-in cards and are locally transformed into the 1.2V core and 2.5V I/O voltage by individual DC/DC converters. The dissipation of 1500W electrical power requires a sophisticated thermal management in terms of the selection of fans, routing of air flow, and choice of effective heat sinks.

For monitoring purposes, we have also added a bit-serial bus throughout the system which complies to the SMBus specification. On each plug-in card the CoolRunner-II CPLD operates as monitoring client and runs all system management operations exclusively. The temperature measurement diode of all Virtex-4 devices is used to initiate an automatic power down when the core temperature is about to exceed the maximum value of 85°C. Hence, the CPLD is connected to all monitoring diodes of each Virtex-4 FPGA and also to the power enabler of the DC/DC converters to control the shut-down of the plug-in module in case of overheating.

7.5 Results

We implemented an ECM architecture in VHDL for Xilinx Virtex-4 SX35 and Spartan-3A DSP 3400 FPGAs and synthesized the design using Xilinx ISE 9.2. Although our design is prepared to be used both for stage 1 and 2, we only implemented the instructions for stage 1 up to now due to time constraints. As an example implementation, the resource consumption of a single ECM core for parameters with $h = 151$ bits is shown in Table 7.3. Note that the requirements on flip-flops are lower for Spartan-3A DSP devices since they come with integrated register stages at the outputs of the BRAMs. Virtex-4 devices, however, need to implement these memory output registers in generic logic and thus consume more flip-flops. For comparison with our implementation, we refer to Gaj *et al.* who reported the use of 3224 slices (5047 LUTs and 3077 flip-flops) on a larger and faster Virtex-4 XC4VLX200-11, however targeting 198 bit parameters.

The enhanced COPACOBANA machine was built and is already fully functional. Unfortunately, we cannot do (reasonable) practical measurements due to a bottleneck in the communication interface[14]. Therefore, Table 7.4 presents runtime estimates obtained by simulation and without limitation on the communication interface. We compared our results to the implementation reported by Gaj *et al.* [GKB+06b, GKB+06a] on the same FPGA platform assuming

[14]More precisely, the bottleneck is inside the FPGA-based Gigabit Ethernet controller which does not support Direct Memory Access (DMA) to support the Gigabit data link.

Aspect	**Spartan-3A DSP 3400**	**Virtex-4 SX35**
Slices occupied	1,142 (4%)	1,306 (8%)
4-input LUTs	1,358	1,289
Flip-flops	703	1,022
DSP blocks	22	22
BRAMs	9	9
Clock Frequency	120 MHz	188 MHz

Table 7.3: Resource consumption of a single ECM core after place-and-route.

Aspect	Spartan-3A DSP	Virtex-4	Virtex-4 [GKB+06b]
FPGA Type	XC3SD3400A-4	XC4VSX35-4	XC4VLX200-4
Clock Frequency	120 MHz	188 MHz	135 MHz
Point Doubling & Addition	355 cycles		947* cycles
Point Multiplication	347,900 cycles		928,060* cycles
ECM Stage 1	345 op/s	540 op/s	140 op/s*
Speedup ratio for ECM	2.46	3.85	1.0

Table 7.4: Clock cycles and frequency for a single ECM core (stage 1) to factor a 151 bit integer with bound $B1 = 960$ and $B2 = 57000$. For comparison, values with asterisk were scaled down from figures for 198 bit integers.

same bounds $B1 = 960$ and $B2 = 57000$. Due to our optimizations with respect to low-level arithmetic, we obtained an architectures which more time and area-efficient (however, we require the availability of DSP blocks). Note further that we are not able to achieve similar high clock frequencies and throughputs as with our ECC design presented in Chapter 3. This is mainly due to the second multiplier and the associated, more complex routing to the memory banks[15]. In addition to that, the multiplier also requires access to an additional memory for storing the modulus n (including different representations of n); a simple ROM holding the special primes P-224 or P-256 is not sufficient anymore. This is another factor contributing to a more complex (and thus less optimal) core arithmetic leading to degraded performance.

Based on the ECM core for 151-bit integers, we can place 5 and 8 such cores on the Spartan-3A DSP 3400 and Virtex-4 SX35, respectively. On both devices, the number of DSP blocks limit the possible number of cores. However, the less efficient routing of multiple parallel cores on the same chip results again in lower clock frequency. First experiments with the Virtex-4 SX35 showed that a clock frequency of 100 MHz can be achieved. Based on this figure, we can

[15]Recall that our design from Chapter 3 only integrates two arithmetic units which can be efficiently operated in parallel with a dual-ported memory.

estimate that 2295 ECM stage 1 operations per second can be performed on a single Virtex-4 SX35 to factorize 151 bit integers (with $B1 = 960$,$B2 = 57000$). At this point, we like to stress that the costs for a Virtex-4 SX35 FPGA device are significant, currently about US$ 500 per device[16]. A more cost-efficient device is the Spartan-3A DSP 3400 FPGA which was recently released. Suche a devices is only about US$ 75 per piece and can support 5 such ECM cores for 151 bit integers at (estimated) 60 MHz clock frequency. This results in around 860 operations per second for stage 1 (assuming same bounds as above).

7.6 Conclusions and Future Work

The work at hand proposes an ECM core which encloses an improvement in time and area with respect to previous ECM implementations in hardware [ŠPK+05, GKB+06b, GKB+06a]. Beside the proposal of a more efficient architecture for stage 1 and 2 of ECM, we built a dedicated FPGA cluster that supports accelerated low-level arithmetic using the DSP blocks of Virtex-4 FPGAs. Although testing of our architectures is not fully accomplished, we estimate that up to 860 and 2300 ECM stage 1 operations per second can be performed on a single Spartan-3A 3400 and Virtex-4 SX35 FPGA, respectively (given integers up to 151 bit, bounds $B1 = 960$ and $B2 = 57000$). Practical verification of our results will follow as soon a sufficiently fast communication controller for COPACOBANA becomes available.

At this point, we like to refer again (cf. Chapter 4) to the power of commodity graphics cards with respect to cryptography (and cryptanalysis): the implementation of ECM in [BCC+09] showed that recent GPUs can already outperform CPUs in this domain. According to the authors, a recent NVIDIA GTX 295 is able to perform 41.88 million modular multiplications per second for general 280 bit moduli. In the same work, an Intel Core 2 Duo Q6600 microprocessor is reported to perform 13.03 million such multiplications per second.

If we compare the reported performance of CPUs and GPUs to our architecture, we could place 3 ECM cores supporting 280 bit integers on a Xilinx Spartan-3A DSP 3400. Based on the assumption that the system runs at a clock frequency of about 60 MHz, we can compute 5.35 million multiplications with 280 bit factors per second. Taking the cost of US$ 75 for such an FPGA into account, this results in a cost-performance ratio which is about the same with respect to a recent Q6600 CPU (approx. US$ 200) but worse than that of a GTX 295 GPU (approx. US$ 500) when comparing the sole chip prices. The potential and power of recent and upcoming GPU generations is remarkable: continuously dropping costs for hardware will certainly make GPUs become even more attractive for future cryptanalytic applications.

[16] Price in December 2008, not including discount for purchase of larger quantities.

Part III

Trust and Protection Models for Reconfigurable Devices

Chapter 8

Intellectual Property Protection for FPGA Bitstreams

The option to load the hardware configuration of FPGAs dynamically opens up the threat of theft of Intellectual Property (IP) of the circuitry. Since the configuration is usually stored in external memory, this can be easily tapped and read out by an eavesdropper. Thus, some FPGA devices allow to use an encrypted copy of the FPGA configuration so that the device cannot be forged without knowledge of the secret key. However, such protection systems based on straightforward use of symmetric cryptography are not well-suited with respect to business and licensing processes, since they are lacking a convenient scheme for key transport and installation. We propose a new protection scheme for the IP of circuits in configuration files that provides a significant improvement to the current unsatisfying situation. It uses both public-key and symmetric cryptography, but does not burden FPGAs with the usual overhead of public-key cryptography.

Contents of this Chapter

8.1 Motivation

When Field Programmable Gate Arrays (FPGA) were first introduced in the 1980s, this was a revolutionary step from static ASIC and VLSI solutions to flexible and maintainable hardware applications. It has become possible to avoid the static designs of standard VLSI technology, and instead to compile electrical circuits for arbitrary hardware functions into configuration files[1]

[1]Note that we also use the terminology *configuration bitstream* or *configuration bit file* as synonyms for the configuration file of FPGAs.

used to program a fabric of reconfigurable logic. A new market has evolved where companies have specialized on the development of abstract hardware functions that can be distributed and licensed to system integrators by using only a logical description file. However, the flexibility of SRAM-based FPGAs also brings up the issue of protecting the Intellectual Property (IP) of such circuit layouts from unauthorized duplication or reverse engineering. Unfortunately, a configuration file of an FPGA can easily be retrieved from a product and used to clone a device with only little effort. Furthermore, IP vendors that deliver configuration files to licensees do not have any control over how many times the IP is actually used. To cope with these problems, several different approaches have been proposed. As an example, a simple "security by obscurity" approach is to split the IP among multiple FPGAs and create a unique timing relationship between the components [Bar05]. This type of mechanism, however, will not protect proprietary IP from more intensive attacks. Moreover, such techniques force IP vendors (who only intend to sell configuration files) to deal with the customer's board layout as well.

In a smarter approach, IP vendors can insist on installing their configuration file only on encryption-enabled FPGA devices using a previously inserted secret key. Common high-end FPGA types like Virtex-II, Virtex-4 and Virtex-5 from Xilinx [Xil08b] as well as Altera's Stratix II GX and Stratix III [Alt06] devices provide decryption cores based on symmetric 3DES and AES block ciphers. With an encrypted configuration file, the IP can only be used on a device that has knowledge of the appropriate secret key. But here the issue of key transfer arises. One approach is to ship FPGAs to the IP owner for key installation: the IP owner installs secret keys in the devices such that these keys are available to decrypt configuration files but remain otherwise unaccessible. After key installation, the devices are returned to the customer. The high logistical effort makes this a very unsatisfying solution to the problem.

Further solutions are based on separate security chips that dongle the IP to a specific FPGA by exchanging cryptographic handshaking tokens between the components [Alt]. Similarly, this approach requires modification to the customer's board layout, additional hardware, and a secure domain for installing the secret parameters. This way, it provides only a partial solution at a high cost.

In the open literature, there are only very few suggestions to enhance this situation. In [Kea01a, Kea01b], a strategy was proposed based on a static secret key already inserted during the manufacturing process. The issue of key transfer is solved by including cores both for encryption and for decryption in the FPGA. Each FPGA specimen containing contains a private or group key which is used to encrypt a bitstream during installation of the configuration. Thus, after encryption, the configuration will work only for itself and for other FPGAs sharing the same fixed key. In [Kea02, SS06], more complex protocols have been proposed for a more complete solution. Both approaches require the implementation of additional security features in the FPGA. Furthermore, they also require the participation of the FPGA manufacturer (as a trusted party) whenever a bitstream is to be encrypted for a particular FPGA. In other words, such transactions cannot be kept just between the IP vendor and the customer.

In this chapter, we propose a new protection scheme for configuration files that provides IP vendors with means for exact tracking and control of their licensed designs. Our solution

does not impose a need for additional hardware components or major modifications of recent FPGA technology. Instead of demanding a crypto component for key establishment in the static logic as needed by [Kea02, SS06], we use the reconfigurable logic for a setup process based on public key cryptography. Besides exact tracking of the number of licensed IP, our approach provides the invaluable advantage of off-site IP installation: the installation of the designs can be performed by the licensees without any shipping of hardware. Our approach does not require the continuing participation of a third party (such as the FPGA manufacturer) either. To enable FPGAs for these new features, only marginal modification are required on recently available FPGA models. The protection scheme presented in this chapter was developed in joint work with Bodo Möller and published in [GMP07a, GMP07b].

8.2 Protection Scheme

In this section, we introduce the new protection scheme for the IP in configuration files for FPGAs.

8.2.1 Participating Parties

Our idea assumes a use case with three participating business parties. The first contributor is the Hardware Manufacturer (HM), who designs and produces FPGA devices. A second participant is the Intellectual Property Owner (IPO), who has created some novel logic design for a specific problem. This IP is synthesized as a configuration bit file for a specific class of FPGAs manufactured and provided by the HM. The IPO wants to distribute the configuration bit file using a special cost or licensing model, usually on a per-volume basis. The final participant is a System Integrator (SI), who intends to use the IPO's design in products employing the HM's FPGA devices. To support a volume licensing model, we want to allow the IPO to limit the number of FPGAs that can use the design.

8.2.2 Cryptographic Primitives

For a protection scheme that is not just based on obscurity, we need to use symmetric as well as asymmetric cryptography to achieve security.

Symmetric Cryptography: For design protection and configuration confidentiality, some FPGAs already implement symmetric encryption based on well-known block ciphers like AES and 3DES [Alt06, Xil08b]. Such cryptographic functionality can actually be used for more than just confidentiality: In the CMAC construction [Nat05], a computational process mostly identical to that of CBC encryption with a block cipher is used to generate a one-block message authentication code (MAC). Thus, to keep the footprint of the crypto component small, an implementation of a single block cipher can be used both for decryption and for MAC verification, using a CBC scheme in both cases. Using separate block cipher keys, a combination of such a MAC with encryption can achieve *authenticated encryption* [BN00]. Based on such a scheme, we simply have to provide a MAC of the CBC-encrypted ciphertext. We can consider the pair

of such keys a single (longer) key k. In this chapter, we will write $\mathsf{AuthEnc}_k(x)$ for authenticated encryption of a plaintext x yielding a ciphertext including a MAC value, and $\mathsf{DecVer}_k(y)$ for the reverse step of decryption of a ciphertext y while also checking for an authentication error based on the MAC value provided as part of the ciphertext.

The use of CBC for confidentiality with CMAC for authentication is just an example of a convenient scheme. Alternatively, we could use any suitable symmetric cryptographic scheme that provides authenticated encryption in an appropriate sense. Regarding the combined scheme using CBC with CMAC, note that a single implementation of either block cipher encryption or block cipher decryption is sufficient in the FPGA. We can use the block cipher "in reverse" for one of the two cryptographic steps, e.g., use a CMAC based on block cipher decryption rather than on block cipher encryption.

Asymmetric Cryptography: If we want to use symmetric data encryption, this means we have the issue of establishing a shared key k between the parties (usually over an untrusted communication channel). As mentioned in the beginning of this thesis, asymmetric cryptography provides a pair of keys consisting of a public (PK) and a private (SK) component that can be used to overcome the key transport deficiencies of symmetric methods. The first publicly known example of public-key cryptography was the Diffie-Hellman (DH) scheme [DH76], which can be used to establish keys for symmetric cryptography. Appropriately used in conjunction with a key derivation function (KDF) based on a cryptographic hash function, the DH scheme remains state-of-the-art to derive a symmetric key. In this context, an important variant of this is the DH scheme using elliptic curve cryptography [ACD+05], namely ECDH. See [Nat06] for elaborate recommendations on the proper use of DH and ECDH. Public-key cryptography can also be used for authentication through digital signatures.

8.2.3 Key Establishment

For the protocol interactions, we define the set of parties as

$$Z = \{\mathrm{HM}, \mathrm{IPO}, \mathrm{SI}, \mathrm{FPGA}\},$$

which corresponds to the participants introduced in Subsection 8.2.1. Additionally, a specific FPGA device is considered a party on its own. A key for symmetric cryptography chosen by party $z \in Z$ is denoted K_z. As already mentioned in the beginning, keys for asymmetric cryptography are given in pairs (PK_z, SK_z) where PK_z is the public key component, and SK_z is the private, or secret, component.

An asymmetric key establishment scheme based on Diffie-Hellman key exchange (including the ECDH variant for elliptic curve cryptography) is described in detail in [Nat06]. Given any two parties' public and private keys $PK_z, SK_z, PK_{z'}, SK_{z'}$ and an additional bit string $OtherInfo$, these parties can determine symmetric key material by evaluating $key(PK_z, SK_{z'}, OtherInfo)$ and $key(PK_{z'}, SK_z, OtherInfo)$ where key is a function combining the basic DH primitive (or the ECDH variant) with a key derivation function (KDF). Each party uses the other party's public key and its own secret key as input to the Diffie-Hellman primitive, which then will yield identical intermediate results for both parties. To get the final output, the KDF is applied to

the given inputs. By varying the *OtherInfo* value (which directly becomes part of the KDF input), the static-key Diffie-Hellman scheme can be used to generate many seemingly independent symmetric keys. Note that the recommendations in [Nat06] for static-key Diffie-Hellman settings additionally require the use of a nonce in the KDF input for each invocation of the key establishment scheme. This use of non-repeated random values ensures that different results can be obtained based on otherwise identical inputs. However, we do not need this nonce here: for our application, the reproducibility of key establishment results is a feature, not a deficiency.

8.2.4 Prerequisites and Assumptions

For realizing our protection scheme, we assume the following prerequisites (P).

P1. **Trusted Party.** We assume the FPGA hardware manufacturer (HM) as a commonly trusted party. All other participating and non-involved parties in the protocol can assume the HM to be trustworthy and unbiased, i.e., the HM will neither share any secrets with other parties nor unfairly act in favor of someone else. All other parties, however, are regarded per se as untrusted, and may try to cheat.

P2. **Secure Cryptography.** Furthermore, we assume that all cryptographic primitives to be computationally secure, i.e., no attacker with a practical amount of computational power will likely be able to compute any secret keys or otherwise break the cryptography in any reasonable amount of time. This includes the symmetric and asymmetric cryptographic primitives as well as the auxiliary cryptographic hash function. Moreover, implementations used for the protocol are assumed to be fault-tolerant, tamper-resistant (cf. [ÖOP03]) and to remain secure against side-channel attacks over multiple executions, i.e., it must not be possible to learn any further information concerning a secret key by analyzing observations from multiple independent protocol runs.

P3. **FPGA Security Environment.** For the scheme, we assume an FPGA with an integrated symmetric decryption engine that is able to handle encrypted configuration files. This reference device is extended with the following features:

   a) A unique device identifier ID (an l-bit value) is assigned by the hardware manufacturer (HM), accessible from the FPGA's fabric.

   b) A symmetric key K_{HM} (an m-bit value) that is statically integrated by the HM during the manufacturing process, and which can only be read by the internal decryption engine but not from the FPGA fabric.

   c) A symmetric key store K_{FPGA} (also an m-bit value) that is implemented as non-volatile memory and allows for storing a variable key. The key stored in K_{FPGA} can either be updated using an external interface (e.g., JTAG, SelectMAP) or via an internal port from the reconfigurable logic of the FPGA. However, it can only be read from the internal decryption engine (not from the fabric).

d) A data exchange register that can be accessed via a standardized configuration interface like JTAG as well as from the reconfigurable fabric using a dual-ported logic. This feature is already available on many common FPGAs based on user-defined instructions in the JTAG protocol.

e) Tamper-resistant control and security components that can withstand invasive and non-invasive attacks on the device. Particular protection mechanisms should cover the integrated keys K_{HM} and K_{FPGA}, the decryption engine and the FPGA controller distributing the unencrypted configuration bits in the fabric of the FPGA after decryption. Hence, a readback of the plain configuration bits or partial reconfiguration must not be possible on a device with an encrypted configuration loaded. Possible physical threats to security-enabled devices as well as side-channel attacks have been discussed more thoroughly in [KK99, BGB06, WP03].

We assume that the decryption engine of the FPGA can be invoked such that either K_{HM} or K_{FPGA} is used to decrypt a configuration file.

8.2.5 Steps for IP-Protection

This section covers the steps performed by each party according to an ideal protection protocol, i.e., assuming that all parties behave as desired without any fraud attempts or protocol abuse. We have five main stages in our scheme (described in detail further below):

A. **SETUP.** On the launch of a new class of FPGA devices, the HM creates a specific bitstream for them, a *Personalization Module* (PM) that later will be used in the personalization stage. An encrypted version of this PM is made available to all participants, together with a public key associated to it.

B. **LICENSING.** When an IPO offers licenses to an SI, it provides a public key of its own. The device identifier *ID* of each FPGA on which the SI intends to use the licensed IP is transferred to the IPO.

C. **PERSONALIZATION.** A personalization stage is performed for each FPGA device in the domain of the SI. The PM is used to install a device-specific key in each FPGA on which it is executed.

D. **CONFIGURATION.** Using the device information, the IPO sends copies of the configuration file containing its IP to the SI, specifically encrypted for each FPGA.

E. **INSTALLATION.** The SI installs the IP in each FPGA (using the appropriate encrypted copy).

The information exchange between the parties is simple. Figure 8.1 shows the data flow between the participants. Steps 1 through 3 can be considered one-time operations (these are part of the setup and licensing stage). On the contrary, steps 4 and 5, which are part of the

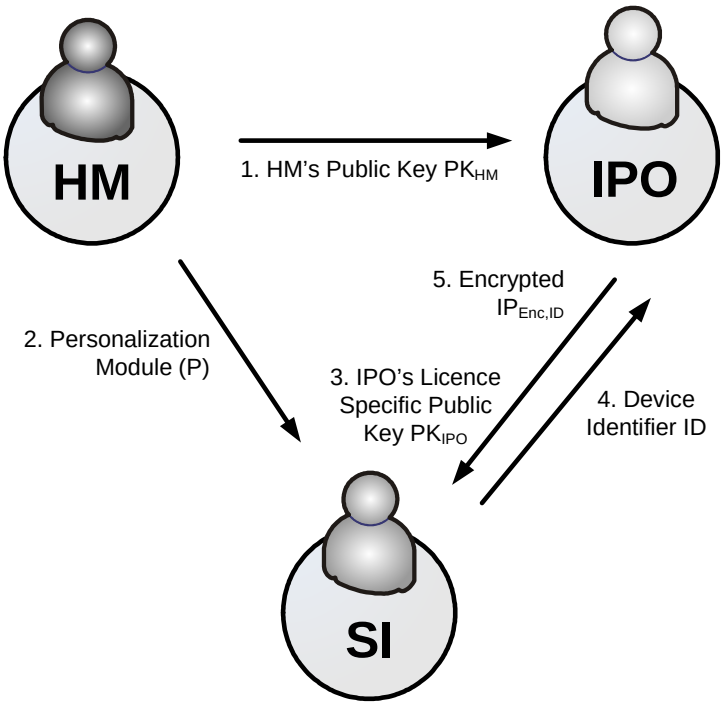


Figure 8.1: Data flow between parties.

licensing and configuration stage, are required to be performed repetitively for each FPGA that should make use of the protected IP.

SETUP Stage (A)

The setup stage, described in the following, is performed once by the HM for each FPGA class to be enabled for the proposed security features. Note that it is reasonable to keep each FPGA class relatively small by splitting a large series of otherwise identical FPGAs into multiple separate classes. This limits the potential damage done if an FPGA's permanent secret K_{HM} or the secret SK_{HM} is compromised (cf. Section 8.2.4). Any such class should be treated as independent from any others, i.e., the results of the setup stage as described below cannot be shared between classes. For practical reasons, a specific part of the device ID should be used to denote the class that a given FPGA belongs to.

A1. The HM generates a symmetric key K_{HM} and an asymmetric key pair $(SK_{\mathrm{HM}}, PK_{\mathrm{HM}})$ for key establishment.

A2. The HM creates a specific bitstream P for the FPGA class such that P implements a key establishment scheme, as described in Section 8.2.3. P includes the private key SK_{HM}. All components employed should be fault-tolerant and include countermeasures against external tampering [KK99]. The bitstream P acts as a personalization module and implements the FPGA behavior that we present in Subsection 8.2.5.

A3. After the configuration bit file P has been built, it is encrypted using the secret key K_{HM}, yielding an encrypted configuration file $P_{\mathrm{enc}} = \mathsf{AuthEnc}_{K_{\mathrm{HM}}}(P)$.

A4. The secret key K_{HM} is statically integrated in each FPGA (cf. Subsection 8.2.4) during the manufacturing process.

After these setup steps have been completed, the HM distributes the encrypted personalization bitstream P_{enc} and the public key component PK_{HM} to all participating parties.

The other parties (notably IPO) must convince themselves that PK_{HM} actually originates from HM. How this authentication is performed in detail is outside the scope of this protocol. However, we remark that if HM supports many different FPGA classes, a Public-Key Infrastructure (PKI) can be very useful.

LICENSING Stage (B)

The licensing stage can be regarded as a first interaction between IPO and SI. To use external IP, the SI signs a contract with the IPO (usually for a volume license). Then, the following steps are required.

B1. The IPO creates a key pair $(SK_{\mathrm{IPO}}, PK_{\mathrm{IPO}})$ and sends the public component PK_{IPO} to the SI.

B2. SI provides IPO with a list of the ID values of those FPGAs for which the SI intends to obtain a license.

Again, authentication of the communication between IPO and SI is not explicitly covered in this work since well-known solutions do exist (e.g., digital signatures). It should be remarked that authentication is required to avoid any abuse of the license model (again, a PKI can be useful).

PERSONALIZATION Stage (C)

In contrast to current solutions, our scheme allows for a key installation that is done in the untrusted domain of the SI. This can take place due to the secret key K_{HM} available in the specific FPGAs from the setup stage. The following steps need to be performed for every FPGA intended to be operated with the IP from the IPO. Note that they can be automated very efficiently. The personalization and the key establishment within an FPGA makes use of the encrypted configuration P_{enc} performing the (one-time) key setup in logic residing in the fabric. Compared with the option of demanding static security components for this step in the static part of an FPGA, this provides huge efficiency advantages since it limits the resources that have to be permanently allocated in the FPGA device.

C1. Using a programming interface, the FPGA is configured with the encrypted personalization module P_{enc} made available by the HM, which is decrypted using the statically integrated key K_{HM}.

C2. Then, the data exchange register of the FPGA is loaded with the public key PK_{IPO} via a common interface (e.g., JTAG). After PK_{IPO} is loaded, the computation process is initiated.

C3. The personalization module determines a symmetric key $key(PK_{\mathrm{IPO}}, SK_{\mathrm{HM}}, ID)$ using the integrated key agreement scheme, and stores the resulting symmetric key in K_{FPGA}. From now on, the FPGA can decrypt designs that are encrypted using this key.

The security properties of the key establishment scheme imply that knowledge of either SK_{HM} or SK_{IPO} is required to compute the key K_{FPGA}. Thus, SI cannot determine K_{FPGA}. Including ID in the KDF input ensures that K_{FPGA} will differ for different FPGA instances. For further implementation aspects, see Section 8.4.

CONFIGURATION Stage (D)

For each FPGA ID for which SI has bought a license, the IPO returns a corresponding configuration file to the SI usable only on the specific FPGA. This mechanism allows the IPO to easily track the number of FPGAs that have been configured to use the licensed IP. In detail, the IPO performs the following steps to generate the FPGA-specific configuration file.

D1. The IPO recovers the specific key K_{FPGA} using its own secret key and the HM's public key:

$$K_{\mathrm{FPGA}} = key(PK_{\mathrm{HM}}, SK_{\mathrm{IPO}}, ID)$$

D2. The IPO encrypts the plain IP configuration bit file using the secret key, thus binding the IP to a specific FPGA device:

$$\mathrm{IP}_{\mathrm{enc},ID} = \mathsf{AuthEnc}_{K_{\mathrm{FPGA}}}(\mathrm{IP}).$$

IPO then transmits this encrypted file to SI.

By the properties of the key establishment scheme, we have

$$key(PK_{\mathrm{IPO}}, SK_{\mathrm{HM}}, ID) = key(PK_{\mathrm{HM}}, SK_{\mathrm{IPO}}, ID),$$

so the key K_{FPGA} as determined by the IPO is identical to the key K_{FPGA} as installed into the FPGA during the personalization stage.

INSTALLATION Stage (E)

After having received $\mathrm{IP}_{\mathrm{enc},ID}$, the SI configures the FPGA with the personalized IP.

E1. SI configures the flash memory of the specific FPGA denoted by identifier ID with $\mathrm{IP}_{\mathrm{enc},ID}$ to operate the device.

Since K_{FPGA} is available in the FPGA, this step enables the FPGA to use the IPO's configuration bit file IP by computing $\mathsf{DecVer}_{K_{\mathrm{FPGA}}}(\mathrm{IP}_{\mathrm{enc},ID})$.

8.3 Security Aspects

We assumed the implementations of cryptographic elements integrated in the personalization module to be fault-tolerant and tamper-resistant. This is mandatory since an attacker might take physical influence on the device while security-relevant processes are performed. Hence, countermeasures against device tampering are required to check if the module is executed under unconventional conditions, e.g, over-voltage, increased temperature or clock frequency. A fairly easy approach to detect operational faults caused by such conditions is to use multiple, identical cryptographic components in the personalization module operated at different clocks or clock shifts. When all computations are finished, all results are compared which will detect any injected faults and operational irregularities. With the option to integrate a multitude of countermeasures [ABCS06, KK99], we are willing to rely on the FPGA behaving as specified. Then, the protocol is a rather straightforward application of well-known cryptographic mechanisms: symmetric encryption in the sense of authenticated encryption, and a key establishment scheme.

Only for authenticated encryption, we require a hardwired implementation within the FPGA device. As outlined in Section 8.2.2, this can be done based on a single block cipher such as AES or 3DES. Authenticated encryption ensures integrity of configuration bitstreams, i.e., a user cannot modify the encrypted configuration to obtain another (related) configuration that would be accepted by FPGA device and modify part of the intended behavior. A similar solution to provide authenticated encryption for FPGAs (based on the existing CBC-mode decryption of FPGAs) was proposed by Saar Drimer in [Dri07].

Finally, we want to emphasize that special care should be taken for the realization of the personalization module because of it is exposed to an untrusted domain. Luckily, such a fault-tolerant and tamper-resilient design has to be developed only once and should be easily portable between different device classes.

8.4 Implementation Aspects

We will give some brief suggestions and implementation details how to realize the participating components, and discuss their expected cost with respect to a recent FPGA device.

8.4.1 Implementing the Personalization Module

In the following we will demonstrate the feasibility of the personalization module that incorporates the implementation of a key establishment scheme as specified by the protection scheme in Section 8.2.

Several public-key algorithms have successfully been implemented in FPGAs. Low footprint RSA/Diffie-Hellman implementations have been proposed in [FL05, Koç95, MRSM03] and are already available in ready-to-use IP cores by FPGA vendors like Altera.

Taking the smallest Virtex-4 XC4VFX12 FPGA with an integrated AES-256 bitstream decryption core as a reference device, we have implemented the key establishment scheme for the

personalization module. For this proof-of-concept implementation, we realized an ECC core over prime fields, which forms the basis for an Elliptic Curve Diffie-Hellman (ECDH). Parameters for this implementation have been chosen by a trade-off of long-term security and efficiency: We designed an ECDH component over $GF(p)$ and 160 bit parameters, which is sufficient for mid-term security nowadays. For details, Table 8.1 presents requirements of logical elements and data throughput of our implementations.

Component	**Lookup-Tables**	**Flip-flops**	**BRAM**	**Data Throughput**
ECDH Core	5,674 LUT	767 FF	5	186Kbit/s
SHA-1 Core	1,102 LUT	689 FF	0	730Mbit/s
ROM	0	0	1	—
JTAG Register	0	320	0	—
SelectMAP Core	318	291	0	—

Table 8.1: Data throughput and logic requirement of personalization components on a Xilinx Virtex-4 FPGA.

For the implementation of the KDF according to [Nat06], we integrated an implementation of the SHA-1 as cryptographic hash function $h(x)$ with an output bit length of 160 bits. During key generation, the SHA-1 hash function is executed twice using three different inputs: Given two 32-bit counter values (effectively, two constants) c_0, c_1, a 128-bit FPGA ID and the ECDH result E, a 256-bit AES key K_{FPGA} can be derived as follows:

$$\begin{aligned} H_i &= h(c_i \parallel E \parallel ID) \text{ for } i \in \{0,1\} \\ K_{\text{FPGA}} &= S(H_0 \parallel H_1), \end{aligned}$$

where $S(x)$ is a selection function choosing the first 256 out of 320 input bits and where $\parallel$ denotes concatenation. The data exchange between the personalization module and an external party (SI) was realized using a shift register which is writable from the JTAG interface. Beside a ROM for storing the secret key of the HM and constants c_0, c_1, a SelectMAP controller is required to program the key storage of the FPGA, which was provided by Berkeley's Bee2 project [Ber06]. The schematic of the implemented personalization module is sketched in Figure 8.2.

For this proof-of-concept work, the SelectMAP core needs to be externally connected with the FPGA's SelectMAP interface since a direct configuration from the fabric is not yet available. It should be remarked that all implementations have been developed for an optimal area-time product so that reductions in logical elements can still be achieved if data throughput is not a primary issue. Concluding, the implementations at hand are small enough to fit even the smallest Virtex-4 XC4VFX12 device (providing a total of 5472 slices of which less than 4000 are required) with some remaining logical resources to add functionality providing tamper resistance and fault tolerance.

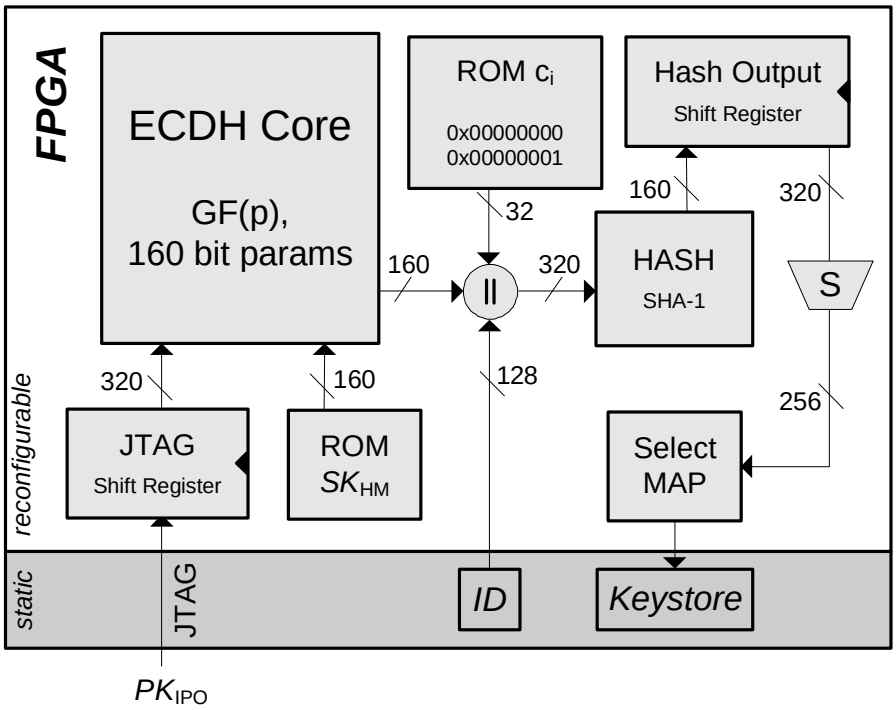


Figure 8.2: Simplified schematic of a personalization module for Xilinx Virtex-4 FPGAs with a 256-bit decryption key.

8.4.2 Additional FPGA Features

To use our proposed key scheme on common FPGA devices, an additional key storage with a fixed key needs to be added to the static design. This is no technical and big financial issue since it can be integrated by the HM either directly in the circuit layout, or using antifuses, or by using similar techniques in a post-production step.

Such a strategy based on laser inscriptions or antifuses can also be used to provide each FPGA with a unique identification number. Alternatively, so called Physically Unclonable Functions (PUF) implemented using a coating or delay technique might be an option to create a unique identification of each chip [Lim04].

A further additional requirement for our scheme is access from the fabric to the key store via an internal (write-only) interface. Since it only requires some internal component repositioning and few dedicated internal I/O pins, this can be cost-efficiently implemented.

Moreover, for advanced bitstream protection in the FPGA, we require the availability of authenticated encryption within the FPGA as explained in Section 8.2.2. A single block cipher engine can be used both for authentication and for decryption, and such reuse means that only little modification are needed for current FPGA architectures already containing a symmetric block cipher engine.

To achieve tamper resistance against invasive attackers, most HMs have already taken efforts to hide away critical parts from observation and manipulation, i.e., by distracting those components over multiple layers and locations. Moreover, strategies known from smart-cards [KK99] could be applied to strengthen FPGAs against physical and invasive attacks.

8.5 Conclusions and Outlook

We proposed a new protection scheme for IP of circuits for FPGAs that provides a significant improvement to the recent unsatisfying situation, with only very few modifications to current FPGA devices and corresponding infrastructures. Due to the use of a personalization module that can be loaded into the reconfigurable logic for a one-time key-establishment, the new protection scheme is suitable for nearly every modern FPGA type with just minor modifications to the function set and architecture.

An open issue is the protection of partial designs, e.g., functional cores that are used as a subcomponent in a configuration of an FPGA. One could make use of a protection scheme with multiple keys K^*_{FPGA} per FPGA, where protected subdesigns are loaded into well-defined fabric areas of the FPGA using the partial reconfiguration feature.

Instead of integrating a static ID into each FPGA, the HM might use a Random Number Generator (RNG) that is additionally included in the personalization module. This RNG can generate a random value taken as input to the KDF and used to bind an encrypted configuration $\text{IP}_{\text{enc},RN}$ to a specific FPGA. Implementations for cryptographically secure RNGs in FPGAs are already available [SMS07, KG04]. The RNG implemented in the reconfigurable logic might be an additional option to reduce the number of modifications with respect to current FPGA architectures to implement our protection scheme.

Chapter 9

Trusted Computing in Reconfigurable Hardware

Trusted Computing (TC) is an emerging technology used to build trustworthy computing platforms. The Trusted Computing Group (TCG) proposed several specifications to implement TC functionalities and security function. Their proposal included a hardware extension to common computing platforms which is known as Trusted Platform Module (TPM). In this chapter we propose a reconfigurable (hardware) architecture which is capable to support TC functionalities as well as trustworthy security components. With respect to the implemented set of security functions, we focus on TPM-similar functionality as proposed by the TCG, but specifically designed for embedded platforms.

Contents of this Chapter

9.1 Motivation

Trusted Computing (TC) is a promising technology used to build trustworthy computing platforms. A recent initiative to implement TC by extending common computing platforms with hardware and software components is due to the Trusted Computing Group (TCG), a consortium of IT enterprises [Tru08]. The TCG specified the Trusted Platform Module (TPM) which provides a small set of cryptographic and security functions, and is assumed to be the trust anchor in a computing platform. Currently, TPMs are implemented as dedicated crypto chip mounted on the main board of computing devices, and many vendors already ship their platforms equipped with TPM chips. The functionalities provided by the TPM allow to securely bind (sensitive) data to a specific platform meaning that the data is only accessible when the underlying platform has the valid and desired configuration.

However, there are several issues to deal with: first, existing TPM chips are currently mainly available for workstations and servers and rather for specific domain applications, in particular not for embedded systems[1]. Second, TPM specifications are continuously growing in size and complexity, and there is still no published analysis on the minimal security functionalities that are practically needed. In addition to this, TPM users have to completely trust implementations of TPM manufacturers, e.g., regarding the compliance to the TCG specification. This also demands the user to trust the TPM implementation that no malicious functionalities have been integrated (like trapdoors or Trojans). Finally, the TCG adversary model considers software attacks only, but manipulations on the *underlying* hardware can circumvent any sophisticated software security measures. Currently, TPM chips are connected to the I/O system with an unprotected interface that can be eavesdropped and manipulated easily [KSP05].

We address most of these issues by proposing a reconfigurable architecture for FPGAs that allows a scalable and flexible usage of trusted computing functionalities. To our knowledge, there has been no proposal for building TC capabilities (e.g., TPM functionalities and corresponding trustworthy security components) in reconfigurable hardware architectures. Our approach allows to bind a reconfigurable application to the underlying TPM and even to bind any higher software layer to the whole reconfigurable architecture. Based on the asymmetric means of an TCG-conform TPM, this can be used as an effective and flexible protection of IP to provide device-specific application software.

We believe that FPGA devices can provide a promising basis for a variety of TC applications in embedded system environments. On the contrary, for enabling TC functionality on these devices, today's FPGA architectures must be extended with additional features but the technologies for the required modifications are already available. Note that we do not primarily focus on the integration of large microprocessors (μP) like commercial Intel Core 2 Duo or AMD Opteron into an FPGA. In fact, our approach assumes embedded applications running on small μPs like ARM known from mobile phones and PDAs.

In this chapter we propose solutions to extend reconfigurable hardware architectures with Trusted Computing functionalities, e.g., for use in embedded systems. In particular, our architecture allows to include the TPM implementation itself into the so-called *chain of trust*. Although we aim at solutions compliant to the TCG proposed TPM specification, our architecture can be deployed independently from TCG approach for future developments of TC technology. Besides a vendor-independent and flexible integration of a TPM in embedded systems, our approach provides the advantage to reduce the trusted computing base to the minimum according to the application's needs. This includes specific functionalities to allow for effective protection scenarios with hardware and software IP (on FPGAs). Parts of this chapter are based on collaborations with Thomas Eisenbarth and Marko Wolf and were published in [EGP+07a, EGP+07b].

[1] At least there exist proposals from Brizek et al. [BKSW05] and the TCG [Tru08] for a specific TPM to also support mobile devices.

9.2 Previous Work

In [KKL+05] the authors already identified various challenges with regard to TPM maintenance in case of hardware or software updates. However, it remains open how they manage it to let the platform initialization begin at the CRTM-enabled TPM (unlike platform initializations starting at the BIOS or CPU), the presented solution is still vulnerable to attacks on the communication link between TPM and the system. Simpson and Schaumont [SS06] provide an IP protection scheme for FPGAs where hardware and software components authenticate each other based on a Physically Unclonable Function (PUF). While [SS06] considers PUF as blackbox, Guajardo *et al.* [GKST07] give an implementation for the PUF and also extend the FPGA-based IP protection protocol. However, both proposals require the availability of an external Trusted Party as well as the integration of a static PUF in the FPGA. This significantly cuts the flexibility with respect to what we are able to provide with a TC-based solution. Drimer [Dri07] proposes an FPGA bitstream authentication mechanism based on two parallel AES engines. However, his approach does not provide any further functionality except AES-based bit stream decryption and authentication. Zambreno *et al.* [ZHC+06] provide a further proposal for a software protection system using reconfigurable hardware but the protection of IP contained in bitstreams is still an active field of research (see, e.g., [BGB06]).

However, to our knowledge, a holistic approach to transfer the extensive capabilities of Trusted Computing systems to reconfigurable hardware has not been published yet.

9.3 TCG based Trusted Computing

This section gives a brief review of the main aspects of Trusted Computing technology as proposed by the TCG [Tru08].

The main TCG specifications are: a component providing cryptographic functions called *Trusted Platform Module* (TPM), a (immutable) part of BIOS (Basic I/O System) called the *Core Root of Trust for Measurement* (CRTM), and the *Trusted Software Stack* (TSS) which is the software interface to provide TC functionalities to the operating system. Many vendors already ship their computing devices with a TPM on the main board and TPM support is also already integrated into commercial operating systems, e.g., to enable hard disk encryption [Mic06], or to measure platform configuration [SZJvD04]. The TCG issues only functional specifications while implementations are left to vendors.

9.3.1 Trusted Platform Module (TPM)

Currently, the TPM is a dedicated hardware chip[2] similar to a smart-card that is assumed to be securely bound to the computing device. Figure 9.1 illustrates the components of a TPM according to the most recent specification version 1.2 [Tru06]. According to [Tru08], a

[2]TPM chips are already available, e.g., from Atmel, Broadcom, Infineon, Sinosun, STMicroelectronics, and Winbond.

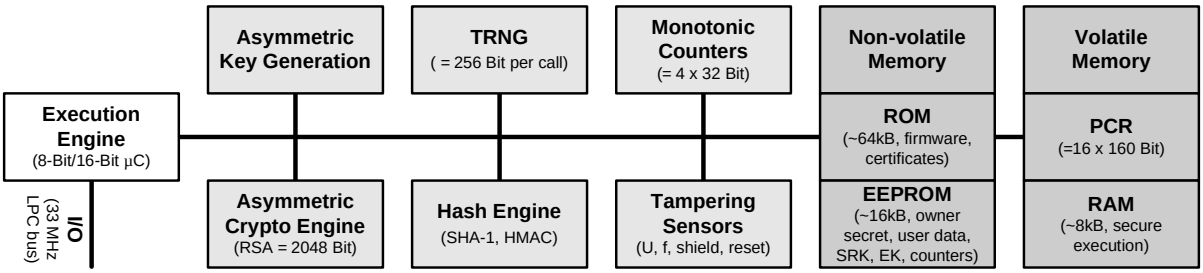


Figure 9.1: Structure and components of a TPM chip version 1.2.

TPM version 1.2 provides the following features: a hardware-based random number generator (RNG), a cryptographic engine for encryption and signing (RSA) as well as a cryptographic hash function (SHA-1, HMAC), read-only memory (ROM) for firmware and certificates, volatile memory (RAM), non-volatile memory (EEPROM) for internal keys, monotonic counter values and authorization secrets, and optionally, sensors for tampering detection. Common TPM chips use a synchronous Low Pin Count-I/O-Interface (LPC-I/O) to communicate with their host system. Based on protected information and hardware functionality the security critical operations like key generation and decryption are performed on-chip. Note that secret keys never leave the TPM device unencrypted. The TPM provides a minimum of 24 registers called *Platform Configuration Registers* (PCR) that can be used to store hash values. A PCR normally can be *extended* only, i.e., the value of a PCR can only be modified as follows: $PCR_{i+1} \leftarrow \mathsf{Hash}(PCR_i|x)$, with the old register value PCR_i, the new register value PCR_{i+1}, and the input x (e.g. a SHA-1 hash value over the instructions of an operating system level or an application). PCR_0 to PCR_{15} are reset only at system startup and thus can only be extended afterwards.

The PCR values are used to establish the *chain of trust*. Roughly speaking, at power-on of the platform, the CRTM[3] computes a hash value of the code and parameters of the boot loader. Computing the hash value is called *measurement* in the TCG terminology. Then the CRTM extends the corresponding PCR and hands over the control to the boot loader that measures the next component, e.g., operating system kernel, and so forth. The security of the chain relies strongly on explicit security assumptions about the CRTM.

9.3.2 Weaknesses of TPM Implementations

The main hardware-based components CRTM and TPM are assumed to be trusted by all involved parties. According to the TCG specification protection for these components against software attacks are required. However, devices are deployed in a potentially hostile environment where an adversary has full access to the underlying hardware. Thus, certain hardware attacks may undermine the security of the TCG approach. As mentioned before, some TPM manufacturers have already started a third party certification of their implementation with respect to security standards (Common Criteria [Com07]) to assure a certain level of tamper-

[3]Currently CRTM is a protected part of TCG-enhanced BIOS.

resistance (as TPM technology stems from the smart-card technology). However, recent TPM still show the following security weaknesses:

- **Unprotected Communication Link.** Currently, in most TCG-enabled platforms communication channels (buses) between TPM, RAM and microprocessor are unprotected. Hence, even if internal information (cryptographic keys, certificates, etc.) within shielded storage of the TPM cannot be compromised, the communication link can be subject to attacks [KSP05]. Moreover, it is unlikely that future processors itself will include required features (key sharing or exchange) for a establishing a secure channel to the TPM.

- **Potential Problems with TPM integration.** Although an integration of the TPM functionality into chipsets makes the manipulation of the Low-Pin Count (LPC) bus between TPM and microprocessor significantly more difficult and costly, the integration also introduces new challenges: on the one hand an external validation and certification of the TPM functionalities (e.g., required by some governments) will be much more difficult, and on the other hand, users may require computing platforms without TPM functionalities which becomes impossible in case of a static integration.

- **Insecure External Memories.** Moreover, standard microprocessors use an external boot ROM to store their initial boot code as well as the CRTM. Thus, an attacker could switch ROM modules to inject malicious boot code and compromise the chain of trust. Although sophisticated mechanisms can be used to build highly secure TPMs and CRTM, the vast majority of TPMs will only provide a limited protection against hardware based attacks, due to the trade-off between costs and tamper-resistance. Nevertheless, at least rudimentary tamper precautions and other countermeasures for memory protection have not been considered in the design and manufacturing process. Hence, system designers and developers should be aware of the adversary model and the assumptions underlying the trusted computing architecture and its instantiation. This is also valid for FPGA designs.

9.4 Trusted Reconfigurable Hardware Architecture

In this section, we outline solutions for realizing trusted computing functionalities in reconfigurable hardware and discuss possible implementations where we follow the TCG approach. However, our architecture can also be used for other possible developments in TC technology.

9.4.1 Underlying Model

Similar as in Chapter 8 the main parties involved are FPGA *manufacturers*, *hardware IP developers* (e.g., developing the application which is synthesized to a configuration file), *software IP developers* who implement software that runs on the loaded bitstream on the FPGA, *system integrators* who integrate hardware and software IP onto an FPGA platform and the *user* who

employs the device. All parties need to trust the FPGA hardware manufacturer. However, IP developers have only limited trust in systems developers, and users have only limited trust in IP and system developers. It is obvious that the entity issuing the update (usually the TPM designer) needs to be trustworthy, or the TPM implementation is subject to certification by some trusted organization.

We assume an *adversary* who can eavesdrop and modify all FPGA-external communication lines, eavesdrop and modify all FPGA-external memories, arbitrarily reconfigure the FPGA, but *cannot* eavesdrop or modify FPGA-internal states. Particularly, we exclude invasive attacks such as glitch attacks, microprobing attacks or attacks using laser or Focused Ion Beam (FIB) to gain or modify FPGA internals. Precautions against other physical attacks such as side channel attack or non-invasive tampering must be taken when implementing the TPM. Furthermore, we do not consider any destructive adversaries which are focusing on denial-of-service attacks, destroying components or the entire system.

9.4.2 Basic Idea and Design

The basic idea is to include the hardware configuration bitstream(s) of the FPGA in the chain of trust. The main security issue, besides protection of the application logic, is to protect the TPM against manipulations, replays and cloning. Hence, appropriate measures are required to securely store and access the sensitive (TPM) state $\mathcal{T}$.

In the following we denote a hardware configuration bitstream as B_X with $X \in \{TPM, App\}$ such that B_{TPM} denotes a TPM bitstream and B_{App} an application bitstream. We further define E_X as the encryption of B_X using a symmetric encryption algorithm and a symmetric encryption key $k_{\mathsf{Enc},X}$. We define A_X as an authenticator of a bitstream B_X with $A_X \leftarrow \mathsf{Auth}_{k_X}(B_X)$ where Auth_{k_X} could be for instance a Message Authentication Code (MAC) based on the authentication key k_X. We denote the corresponding verification algorithm of an authenticator A_X with $\mathsf{Verify}_{k_X}(B_X, A_X)$. In case that a bitstream has been encrypted to preserve design confidentiality, B_X is replaced by E_X. Thus, the corresponding authenticator A_X becomes $A_X \leftarrow \mathsf{Auth}_{k_X}(E_X)$. Unlike Chapter 8, we use here a separate operator for encryption and authentication since this provides direct access to their individual properties (e.g., to the authenticator A_X). According to [BN00], an *Encrypt-then-MAC* authenticated encryption scheme provides the strongest security (with respect to the two other possible schemes *MAC-then-Encrypt* and *Encrypt-and-MAC*). We finally define C_X as an unique representative of B_X's configuration. This value C_X can be, for example, directly obtained from *computed* authenticator A'_X when the FPGA verifies the authenticity of the configuration B_X (i.e., the FPGA checks that $A'_X = A_X$ for B_X).

Figure 9.2 shows our high-level reconfigurable architecture. The bitstreams B_{App}, B_{TPM} of the application and the TPM core without any state $\mathcal{T}$ are stored authenticated (and encrypted) in the *external (untrusted) memory* EM.

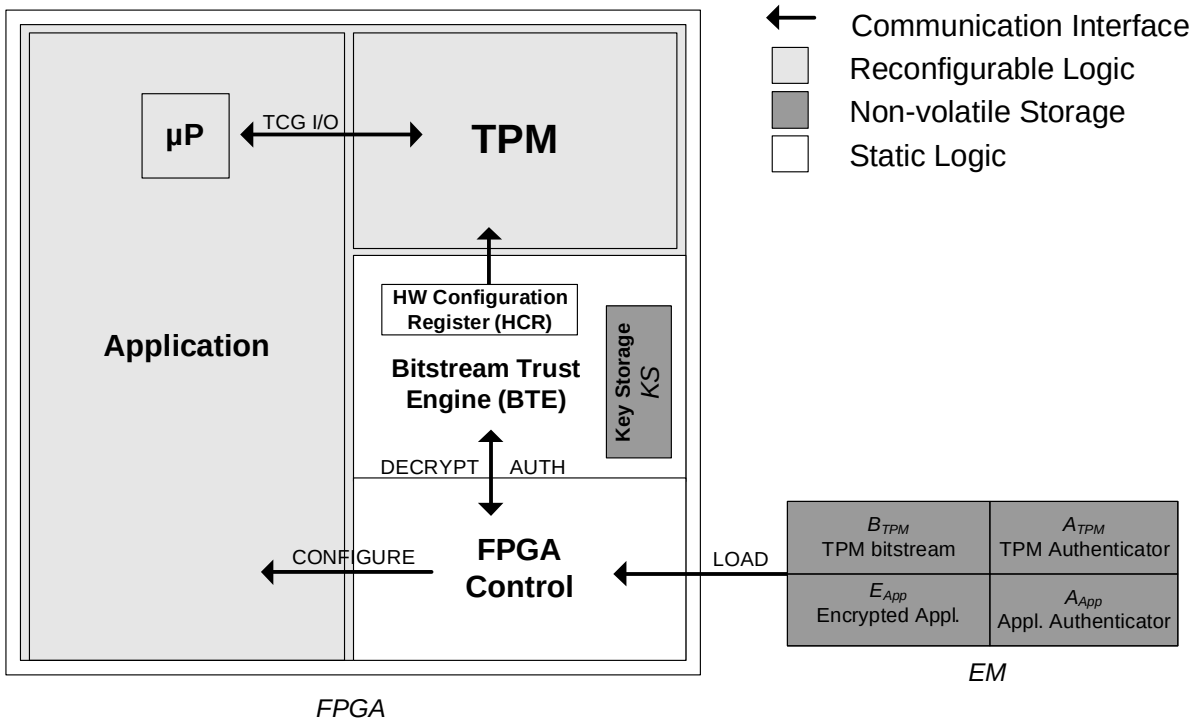


Figure 9.2: Proposed architecture of a trusted FPGA.

The *FPGA Control* logic allows partial hardware configuration[4] of the FPGA fabric to load the TPM and the application independently using the LOAD and CONFIGURE interfaces.

The *Bitstream Trust Engine* (BTE) provides means to decrypt and verify the authenticity and integrity of bitstreams using the DECRYPT and AUTH interfaces[5]. Furthermore, the BTE includes a protected and non-volatile key storage (KS) to store the keys for bitstream decryption and authentication. Finally, the BTE provides a volatile memory location called Hardware Configuration Registers (HCR) to store the configuration information of loaded bitstreams. These registers are used later on by the TPM to set up its internal Platform Configuration Registers (PCR).

In the following we define two stages in our protocol, the *setup* and the *operational* phase.

9.4.3 Setup Phase

To enable an FPGA with TC functionality, a TPM issuer designs a TPM and synthesizes it to a (partial) bitstream B_{TPM} for use on an FPGA. Furthermore, we assume an application designer to provide a TC-enabled FPGA application delivered as partial bitstream B_{App} which can interact with the TPM architecture using a dedicated interface. Particularly with an *open*

[4]This is a feature already available for some recent FPGA devices, e.g., available from Xilinx [Xil08b].

[5]Except for authentication, recent FPGAs already provide LOAD and DECRYPT interfaces (cf. Section 9.5) [Alt06, Xil08b].

TPM implementation, it is possible that both components are developed by a single party, e.g., by the system developer itself.

S1. The system developer verifies the authenticity of B_{TPM} and B_{App}, encrypts B_{App} to E_{App} and then creates bitstream authenticators A_{TPM} and A_{App} using the authentication keys k_{TPM} and k_{App}, respectively.[6]

S2. The TPM bitstream B_{TPM}, its authenticator A_{TPM}, the encrypted application bitstream E_{App}, and its authenticator A_{App} are stored in the external memory EM.

S3. The system developer writes the appropriate authentication keys k_{TPM} and k_{App} (and the encryption key k_{App}) to the key store KS of the BTE.

9.4.4 Operational Phase

Remember that on each power-up the FPGA needs to reload its hardware configuration from the external memory EM. Hence, for loading a TC-enabled application, the following steps need to be accomplished:

O1. On device startup, the FPGA controller reads the TPM bitstream B_{TPM} and the corresponding authentication information A_{TPM} from the external memory EM. BTE verifies the authenticity and integrity of B_{TPM} based on the authenticator A_{TPM} by using $\mathsf{Verify}_{k_{TPM}}(B_{TPM}, A_{TPM})$.

With successful verification of the bitstream, BTE computes the configuration value C_{TPM} of the TPM bitstream and writes C_{TPM} into the first Hardware Configuration Register (HCR) before the FPGA's fabric is finally configured with B_{TPM}.

O2. The TPM requires exclusive access to a non-volatile memory location to store its sensitive state $\mathcal{T} = (EK, SRK, TD)$ where EK denotes an asymmetric key that uniquely identifies each TPM (Endorsement Key), SRK an asymmetric key used to encrypt all other keys created by the TPM (Storage Root Key) and TPM data TD includes further security-critical non-volatile data of the TPM. This requires an extension of recent SRAM-FPGA devices with on-chip non-volatile storage which is discussed in more detail in Section 9.5. Furthermore, the access to this storage location is protected by an Access Control Function (ACF) integrated in the FPGA's static logic which provides access to sensitive data only when a specific bitstream (i.e., the TPM) is loaded. For full flexibility, the ACF implements an interface with which a currently configured bitstream can request a reset (and implicitly, a clear) of the non-volatile memory to reassign the access to the storage for its own exclusive use. The access authorization to the memory for a loaded bitstream X can easily be performed by BTE by checking its C_X stored in the first HCR.

O3. After the TPM has been loaded into the fabric, the application bitstream E_{App} and its authenticator A_{App} are read from EM, verified and decrypted in the same way. The

[6] If TPM bit stream B_{TPM} is also provided by the system integrator itself, he can choose $k_{TPM} = k_{App}$.

BTE stores the configuration value C_{App} of the verified application in the second HCR register. After the application bitstream has been configured in the FPGA, the first call of the application initializes the TPM as follows: Based on the content of the HCR (C_{TPM}, C_{App}), the TPM initializes its own PCRs: $PCR_1 \leftarrow \mathsf{Hash}(PCR_0|C_{TPM})$ and $PCR_2 \leftarrow \mathsf{Hash}(PCR_1|C_{App})$ where $\mathsf{Hash}(x)$ denotes the internal hash function of the TPM and PCR_0 is some constant (root) value. In this way the (unique) configurations of all bit streams can be included in the chain of trust[7].

After loading the hardware configuration of TPM and application into the FPGA, the chain of trust can be extended by the measurements of other specific system components like the operating system and high-level application software. This allows to bind any higher level application (of the IP provider) to the underlying FPGA by binding the application (or its data) using the subset of the PCR registers that contain the corresponding measurements of the underlying FPGA.

9.4.5 TPM Updates

The update of the current TPM_1 to another TPM_2 on an FPGA is quite easy when the sensitive state $\mathcal{T}$ does not need to be migrated. The TPM_2 needs to be loaded and will obviously not be able to access the ACF controlled memory containing $\mathcal{T}$ of TPM_1 (since TPM_2 cannot provide C_{TPM_1}). Hence, TPM_2 reassigns the ACF to be able to create and store its own $\mathcal{T}$. With the reset of the ACF, the previous $\mathcal{T}$ in the non-volatile memory is cleared[8] so that no confidential information of TPM_1 will be accessible for TPM_2.

However, for migrating $\mathcal{T}$ from TPM_1 to TPM_2 without loss of $\mathcal{T}$, we propose to extend existing TPM implementations by an migration function[9] $\mathsf{Migrate}(\mathsf{UA}, C_{TPM_2})$ where UA is an *Update Authenticator* and C_{TPM_2} a unique reference to the corresponding TPM_2. For a TPM update, a system developer (who has set k_{TPM} for the corresponding FPGA) generates an update authenticator $\mathsf{UA} \leftarrow \mathsf{Sign}_{SK_{UPD}}(C_{TPM_2}, P_{TPM})$ with the following parameters: SK_{UPD} denotes an update signing key where TPM_1 trusts the corresponding update verification key PK_{UPD}, e.g., preinstalled in TPM_1. Thus, TPM_1 knows a set of trusted update authorities (the system developer, etc.) who are allowed to perform the migration of $\mathcal{T}$ for use with TPM_2. P_{TPM} denotes a reference to the class of TPMs that provides a certain (minimum) set of security properties. Note, P_{TPM} can also be replaced by individual update signing keys each representing a single security property.

When the user requests a TPM update (e.g., as a feature of an application), he invokes the migration function of TPM_1 using the parameters UA and C_{TPM_2} received from the

[7] This is similar to the initialization of a desktop TPM via CRTM. However, now the PCR includes the hardware measurement results of the TPM itself.

[8] To prevent denial-of-service attacks against $\mathcal{T}$, BTE can additionally implement a mechanism such that TPM_1 has to clear its $\mathcal{T}$ before TPM_2 is able to reassigns the ACF for its own $\mathcal{T}$.

[9] Note, our migration functionality does *not* replace TCG mechanisms for migrating internals called `TPM_Migration` respectively `TPM_Maintenance`.

corresponding system developer (over an untrusted channel). Then, the migration function $\mathsf{Migrate}(\mathsf{UA}, C_{TPM_2})$ performs the following steps:

U1. The migration function of TPM_1 verifies UA using the update verification key PK_{UPD} and checks whether P_{TPM_2} provides the same (minimum) set of security properties as P_{TPM_1}.

U2. After successful verification, the migration function of TPM_1 reassigns the ACF (containing $\mathcal{T}$) for use with TPM_2. The ACF needs to grant access to TPM_2 without erasing the non-volatile memory. More precisely, the BTE provides a further interface so that only TPM_1 with access to the ACF memory can associate the memory with C_{TPM_2}. After reassignment of the ACF memory, only the new TPM_2 is able to access $\mathcal{T}$.

After the migration function has terminated, the application (or manually, the user) overwrites TPM_1 stored in the external memory EM with B_{TPM_2} and the corresponding authenticator A_{TPM_2}. Now, the user restarts the FPGA to reload the updated TPM and application (cf. Section 9.4.4).

9.4.6 Discussion and Advantages

Enhancing an FPGA with TC mechanisms in reconfigurable logic can provide the following benefits.

- **Enhancing Chain of Trust.** As mentioned in Section 9.3.1, recent TPM enabled systems establish the chain of trust by starting from the CRTM, which is currently part of the BIOS. For TPMs hosted in FPGAs, the BTE can begin with the hardware configuration of the application and even with the TPM itself. Therefore, the chain of trust can include the underlying hardware as well as the TPM hardware configuration, i.e., the chain of trust paradigm can be moved to the hardware level.

- **Flexible Usage of TPM Functionality.** The developer may also utilize the basic functionality of the TPM in his application which can make the development of additional cryptographic units obsolete. This includes the generation of true random numbers, the asymmetric cryptographic engine as well as protected non-volatile memory. Furthermore, a flexible FPGA design allows to use only that TPM functionality which is required for the application.

- **Flexible Update of TPM Functionality.** A TPM implemented in reconfigurable logic of an FPGA can easily be adapted to new requirements or versions. For example, if the default hash function SHA-1 turns out to be not secure enough [CR06], an FPGA hosted TPM could include a self-modification feature which updates the corresponding hash function component. Moreover, patches fixing potential implementation errors or changes/updates enhancing interoperability could be applied quickly and easily. The current TCG specification defines the binding/sealing functionality based on binary hashes

and hence any changes to the chain of trust can render sealed data inaccessible, even when keeping the same level of security. This is a general limitation of the TCG solution and holds for our chain of trust model as well. However, in [KKL+05] the authors propose the concept of *property-based sealing* that provide a mapping between security properties provided by a platform configuration and its binary measurements making updates very efficient, since as long as properties are preserved, changes during update to binary measurements have no impact on sealed data. In this context, the authors also propose to use a new TPM command called `TPM_UpdateSeal` that allows a TPM to verify a certificate issued (by a trusted third party) on a new configuration, and hence reseal the data under the new configuration.

- **Improved Communication Security.** The integration of CPU, ROM, RAM and TPM into a single chip enhances protection of communication links between these security-critical components from being intercepted or manipulated. With the boot ROM and RAM integrated in the FPGA, the injection of malicious boot code or RAM manipulations becomes much more difficult.

- **Vendor Independence.** Platform owners can select which TPM implementation is operated on their platforms. This allows even the usage of fully vendor independent *open TPM implementations* providing more assurance regarding trapdoors and Trojans. Moreover, since we can easily implement a TPM soft core into hardware, a multitude of vendors can offer a variety of TPM implementations. Thus, users are not only restricted to a few TPM ASIC manufacturers as today, they even can implement their own TPM instances[10] and thus do not have to trust any external manufacturer.

9.5 Implementation Aspects

Incorporating TC functionality into FPGAs for enabling trusted embedded computing platforms requires some modifications to current FPGA architectures. Hence, we will only present implementation proposals to demonstrate the feasibility of TPMs on reconfigurable devices. As a starting point, we assume a recent FPGA device with integrated protection mechanisms, i.e., an SRAM-based FPGA that provides symmetric bitstream decryption, partial hardware configuration and a small amount of non-volatile (key) storage. Such FPGA architecture implement the non-volatile key store in different ways, e.g., Altera's Stratix II devices use an (single-write) anti-fuse technique [Alt06]. On Virtex-II, 4 and 5 FPGAs, Xilinx stores the key during power-down using rewritable memory-buffered by a battery [Xil08b]. The advantage of being able to replace the key comes at the price of an additional external battery element, implying a limited lifetime of the system. Other approaches might use logic with built-in flash memory but this is only available with a less advanced manufacturing technology resulting in smaller devices (cf. Actel IGLOO or ProAsic3 [Act08]). Instead of using battery-buffered memory locations for

[10] These own implementations can be certified for TCG compliance by a trustworthy authority.

preserving the TPM state $\mathcal{T}$, it is possible to integrate a non-volatile memory directly on the FPGA chip. Newer FPGA devices like the Xilinx Spartan 3AN [Xil08b] already offer SRAM-based circuits combined with a non-volatile Flash memory layer in the same package. The Flash memory in those devices allows for storing up to 11 MBit of user-defined data, perfectly suited for storing $\mathcal{T}$ (with an additional ACF implementation).

To realize the BTE and bitstream authentication of B_{App} and B_{TPM} we require minor modifications to the existing (AES) decryption cores. For integrity verification and authentication of bit streams in the BTE, one option is to use a Message Authentication Code (MAC) which ideally uses the same cryptographic engine used for bitstream decryption (cf. the authenticated encryption feature discussed in Chapter 8).

TPM Function	**Reference**	**Logic**	**Memory**
RSA-2048 Core	ARSA-128 [FL05]	900 LE	13k
SHA-1 Core	Helion SHA-1 [Hel08]	1000 LE	0
TRNG	Fabric TRNG [KG04]	$<$ 50 LE	256
TPM Firmware	Internal RAM	-	16k
Volatile Memory	Internal RAM	-	16k
Controller	Picoblaze [Xil08b]	192 LE	18k
Total (+overhead)	-	3000	75k

Table 9.1: Estimated number of Logical Elements (LE) and RAM bits for the TPM functionality.

For all other (reconfigurable) cryptographic components of the TPM itself, a multitude of proposals for efficient implementation are available in the open literature. Table 9.1 shows estimates for the required resources of a TPM which is realized in an FPGA. Please note that the implementations have been selected to be area-optimal. We have converted the resource requirements to a universal metric based on Logical Elements (LE) which was chosen to maintain platform independence among different classes of FPGAs[11]. Translated to a low-cost Xilinx Spartan-3AN XC3S1400AN with a total system complexity of 25,344 LEs, the TC enhancements will take about 3,000 LEs and require about 12% of the device capacity. Hence, we can conclude that a TPM implementation will obviously be efficient with recent devices.

9.6 Conclusions

In this chapter we proposed solutions for reconfigurable hardware architectures based on FPGAs providing Trusted Computing (TC) mechanisms as proposed by the TCG specifications.

Integrating TC mechanisms into reconfigurable hardware allows to integrate application hardware and TC functionalities into the chain of trust, which helps building trustworthy embedded platforms for securing various security-critical embedded applications.

[11] In our notion, an LE consists of a single 4-input LUT connected to an optional, single-bit flip-flop

The flexible, efficient and scalable realization of TC mechanisms, allows manufacturer independent TC designs and flexible updates of TC functionalities. Due to complete integration, the given solutions also improve the protection against certain hardware attacks.

Future work can include a realization of our architecture as a proof-of-concept implementation and detailed analyses on the tamper resistance of FPGA chip packages.

Part IV

Appendix

Additional Tables

We here provide supplementary tables and information for the work presented in this thesis.

	Cycle	Key	C0	C1	C2	C3		Key	C0	C1	C2	C3	Out
	1		a_{00}	–	–	–			L				
	2		a_{03}	a_{11}	–	–			R	L			
	3		a_{02}	a_{13}	a_{22}	–	P		T	R	L		
	4		a_{01}	a_{10}	a_{21}	a_{33}	I		D	T	R	L	
I	5	$K_{0[0]}$	–	a_{12}	a_{20}	a_{32}	P	$K_{0[0]}$	$\oplus\searrow$	D	T	R	
N	6	$K_{0[1]}$	–	–	a_{23}	a_{31}	E	$K_{0[1]}$	$\oplus\searrow$	$\oplus\searrow$	D	T	
P	7	$K_{0[2]}$	–	–	–	a_{30}	L	$K_{0[2]}$	$\oplus\searrow$	$\oplus\searrow$	$\oplus\searrow$	D	
U	8	$K_{0[3]}$	–	–	–	–	I	$K_{0[3]}$	$\oplus\searrow$	$\oplus\searrow$	$\oplus\searrow$	$\oplus\rightarrow$	E_0
T	9		a'_{00}	–	–	–	N		L	$\oplus\searrow$	$\oplus\searrow$	$\oplus\rightarrow$	E_1
	10		a'_{03}	a'_{11}	–	–	E		R	L	$\oplus\searrow$	$\oplus\rightarrow$	E_2
	11		a'_{02}	a'_{13}	a'_{22}	–			T	R	L	$\oplus\rightarrow$	E_3
	12		a'_{01}	a'_{10}	a'_{21}	a'_{33}			D	T	R	L	
	13	$K_{1[0]}$	–	a'_{12}	a'_{23}	a'_{32}		$K_{1[0]}$	$\oplus\searrow$	D	T	R	

Table A.1: Supplementary information to Chapter 2: initial 13 clock cycles of the eight pipeline stages computing a plaintext input. Steps are RAM lookup L, RAM output register R; transform T, DSP input register D and DSP XOR $\oplus$. After eight cycles the output column E'_0 is used as input to the next round, etc.

1st Base Ext.	**2nd Base Ext.**	**1024 bit range**	**2048 bit range**
Bajard *et al.*	Shenoy *et al.*	981	2003
	Others	1013	2035
Others	Shenoy *et al.*	990	2014
	Others	1022	2046

Table A.2: Supplementary information to Chapter 4: modulus sizes for modular multiplication using RNS.

Step	Multiplier			Adder/Subtracter		
	Target	Operation	Description	Target	Operation	Description
1	$T0$	$Y_1 \cdot Z_2^3$	$\mathsf{Y_1 \cdot Z_2^3 = S_1}$			
2	$T1$	$Y_2 \cdot Z_1^3$	$\mathsf{Y_2 \cdot Z_1^3 = S_2}$			
3	$T2$	$X_1 \cdot Z_2^2$	$\mathsf{X_1 \cdot Z_2^2 = U_1}$	$T1$	$T1 - T0$	$\mathsf{S_2 - S_1 = R}$
4	$T3$	$X_2 \cdot Z_1^2$	$\mathsf{X_2 \cdot Z_1^2 = U_2}$			
5	$T4$	$T1^2$	$\mathsf{R^2}$	$T3$	$T3 - T2$	$\mathsf{U_2 - U_1 = H}$
6	$T5$	$T3^2$	$\mathsf{H^2}$			
7	$T6$	$T5 \cdot T2$	$\mathsf{U_1 H^2}$			
8	$T2$	$T5 \cdot T3$	$\mathsf{H^3}$	$T5$	$T6 + T6$	$\mathsf{2 \cdot U_1 H^2}$
9	$T7$	$T2 \cdot T0$	$\mathsf{S_1 \cdot H^3}$	$T4$	$T4 - T5$	$\mathsf{R^2 - 2 \cdot U_1 H^2}$
10	$T3$	$T3 \cdot Z_1$	$\mathsf{Z_1 \cdot H}$	X_3	$T4 - T2$	$\mathsf{R^2 - 2 \cdot U_1 H^2 - H^3}$
11	$Z3$	$T3 \cdot Z_2$	$\mathsf{Z_1 \cdot Z_2 \cdot H = Z_3}$	$T4$	$T6 - X3$	$\mathsf{U_1 H^2 - X_3}$
12	$T6$	$T1 \cdot T4$	$\mathsf{R(U_1 H^2 - X_3)}$			
13	$Z3^2$	$Z_3 \cdot Z_3$	$\mathsf{Z_3^2}$	Y_3	$T6 - T7$	$\mathsf{R(U_1 H^2 - X_3) - S_1 \cdot H^3}$
14	$Z3^3$	$Z_3^2 \cdot Z_3$	$\mathsf{Z_3^3}$			

Table A.3: Supplementary information to Chapter 3: instruction sequence for point addition using projective Chudnovsky coordinates based on a parallel adder and multiplier.

Step	Multiplier			Adder/Subtracter		
	Target	Operation	Description	Target	Operation	Description
1	$T0$	$Z_1^2 \cdot Z_1^2$	$\mathsf{Z_1^4}$	$T2$	$Y_1 + Y_1$	$\mathsf{2 \cdot Y_1 = B}$
2	$T1$	$X_1 \cdot X_1$	$\mathsf{X_1^2}$			
3	$T3$	$T2 \cdot T2$	$\mathsf{B \cdot B = C}$	$T0$	$T0 - T1$	$\mathsf{(X_1^2 - Z_1^4)}$
4	$Z3$	$T2 \cdot Z1$	$\mathsf{B \cdot Z_1 = Z3}$	$T1$	$T0 + T0$	$\mathsf{2 \cdot (X_1^2 - Z1^4)}$
5	$T4$	$T3 \cdot X1$	$\mathsf{C \cdot X_1 = D}$	$T1$	$T1 + T0$	$\mathsf{3 \cdot (X_1^2 - Z_1^4) = A}$
6	$T5$	$T1 \cdot T1$	$\mathsf{A^2}$	$T0$	$T4 + T4$	$\mathsf{2D}$
7	Z_3^2	$Z_3 \cdot Z_3$	$\mathsf{Z_3^2}$	X_3	$T5 - T0$	$\mathsf{A^2 - 2D = X_3}$
8	T	$T3 \cdot T3$	$\mathsf{C^2}$	$T5$	$T - X_3$	$\mathsf{D - X_3}$
9	$T2$	$T1 \cdot T5$	$\mathsf{A \cdot (D - X_3)}$	6	$T0\ DIV\ 2$	$\mathsf{C^2/2}$
10	Z_3^3	$Z_3^2 \cdot Z_3$	$\mathsf{Z_3^3}$	Y_3	$T2 - T6$	$\mathsf{A \cdot (D - X_3) - C^2/2}$

Table A.4: Supplementary information to Chapter 3: instruction sequence for point doubling using projective Chudnovsky coordinates based on a parallel adder and multiplier.

Bibliography

[ABCS06] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov. Cryptographic Processors – A Survey. *Proceedings of the IEEE*, 94(2):357–369, Feb 2006.

[ACD+05] R. M. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Chapman & Hall/CRC, 2005.

[Act08] Actel Corporation. IGLOO and ProASIC Flash-based FPGAs, 2008. Available at `http://www.actel.com/products/`.

[Alg07] Algotronix Ltd. AES G3 data sheet: Xilinx edition, October 2007. Available at `http://www.algotronix-store.com/kb_results.asp?ID=7`.

[Alt] Altera Corporation. FPGA design security using MAX II reference design. Available at `http://www.altera.com/support/refdesigns/sys-sol/indust_mil/ref-des-secur.html`.

[Alt06] Altera Corporation. Stratix II GX and Stratix III FPGAs, 2006. Available at `http://www.altera.com/products/devices/`.

[AM93] A. O. L. Atkin and F. Morain. Finding suitable curves for the elliptic curve method of factorization. *Mathematics of Computation*, 60:399–405, 1993.

[AMV93] G. B. Agnew, R. C. Mullin, and S. A. Vanstone. An Implementation of Elliptic Curve Cryptosystems Over $f_{2^{155}}$. *IEEE Journal on Selected Areas in Communications*, 11(5):804–813, 1993.

[ANS05] ANSI X9.62-2005. American National Standard X9.62: The Elliptic Curve Digital Signature Algorithm (ECDSA). Technical report, Accredited Standards Committee X9, `http://www.x9.org`, 2005.

[ATI06] Advanced Micro Devices, Inc. (AMD), Sunnyvale, CA, USA. *ATI CTM Guide, Release 1.01*, 2006. Available at `http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf`.

[Bar05] T. Barraza. How to Protect Intellectual Property in FPGA Devices Part II. *Design and Reuse Online: Industry Articles*, 2005. Available at `http://www.us.design-reuse.com/articles/article11240.html`.

[BBJ+08] D. J. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters. Twisted Edwards Curves. In S. Vaudenay, editor, *Proceedings of First International Conference on Cryptology in Africa – AFRICACRYPT 2008*, volume 5023 of *LNCS*, pages 389–405. Springer-Verlag, 2008. Document ID: c798703ae3ecfdc375112f19dd0787e4.

[BBLP08] D J. Bernstein, P. Birkner, T. Lange, and C. Peters. ECM using Edwards curves. Cryptology ePrint Archive, Report 2008/016, January 2008. Document ID: cb39208064693232e4751ec8f3494c43.

[BCC+09] D. J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang. ECM on Graphics Cards. Cryptology ePrint Archive, Report 2008/480, January 2009. Document ID: 6904068c52463d70486c9c68ba045839.

[BDK01] J.-C. Bajard, L.-S. Didier, and P. Kornerup. Modular Multiplication and Base Extension in Residue Number Systems. In N. Burgess, editor, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic ARITH15*, pages 59–65, Vail, Colorado, USA, June 2001.

[BDR+96] M. Blaze, W. Diffie, R. L. Rivest, B. Schneier, T. Shimomura, E. Thompson, and M. Wiener. Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security. Technical report, Security Protocols Workshop, Cambridge, UK, January 1996. Available at `http://www.counterpane.com/keylength.html`.

[Ber01] D. J. Bernstein. A software implementation of NIST P-224. In *5th Workshop on Elliptic Curve Cryptography (ECC 2001), University of Waterloo, October 29-31*, 2001. Available at `http://cr.yp.to/nistp224/timings.html`.

[Ber06] Berkeley University. BEE2 Project, 2006. Available at `http://bee2.eecs.berkeley.edu`.

[BGB06] L. Bossuet, G. Gogniat, and W. Burleson. Dynamically configurable Security for SRAM FPGA Bitstreams. *International Journal of Embedded Systems*, 2(1-2):73–85, 2006.

[BKSW05] J. P. Brizek, M. Khan, J. Seifert, and D. M. Wheeler. A platform-level trust-architecture for hand-held devices. In *Presented at Cryptographic Advances in Secure Hardware (CRASH 2005)*, September 2005.

[BMP05] J.-C. Bajard, N. Meloni, and T. Plantard. Efficient RNS Bases for Cryptography. In *Proceedings of IMACS 2005 World Congress*, Paris, France, July 2005.

[BN00] M. Bellare and C. Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In T. Okamoto, editor, *Advances in Cryptology – Proceedings of ASIACRYPT 2000*, volume 1976 of *LNCS*, pages 531–545. Springer-Verlag, 2000.

[Bos85] W. Bosma. Primality testing using elliptic curves. Technical Report 85-12, Universiteit van Amsterdam, 1985.

[BP01] T. Blum and C. Paar. High Radix Montgomery Modular Exponentiation on Reconfigurable Hardware. *IEEE Transactions on Computers*, 50(7):759–764, 2001.

[BP04] J.-C. Bajard and T. Plantard. RNS bases and conversions. *Advanced Signal Processing Algorithms, Architectures, and Implementations XIV*, 5559:1:60–69, 2004.

[Bre86] R. P. Brent. Some Integer Factorization Algorithms Using Elliptic Curves. *Australian Computer Science Communications*, 8:149–163, 1986.

[BSQ+08] P. Bulens, F.X. Standaert, J.-J. Quisquater, P. Pellegrin, and G. Rouvroy. Implementation of the AES-128 on Virtex-5 FPGAs. In S. Vaudenay, editor, *Proceedings of First International Conference on Cryptology in Africa – AFRICACRYPT 2008*, volume 5023 of *LNCS Series*, pages 16–26. Springer-Verlag, 2008.

[CC86] D. V. Chudnovsky and G. V. Chudnovsky. Sequences of numbers generated by addition in formal groups and new primality and factorization tests. *Advances in Applied Mathematics*, 7(4):385–434, 1986.

[Cer97] Certicom. The Certicom ECC Challenge, 1997. Available at `http://www.certicom.com/index.php/the-certicom-ecc-challenge`.

[Cer00a] Certicom Research. Standards for Efficient Cryptography – SEC 1: Elliptic Curve Cryptography. Available at `http://www.secg.org/secg_docs.htm`, September 2000. Version 1.0.

[Cer00b] Certicom Research. Standards for Efficient Cryptography – SEC 1: Recommended Elliptic Curve Domain Parameters. Available at `http://www.secg.org/secg_docs.htm`, September 2000. Version 1.0.

[CG03] P. Chodowiec and K. Gaj. Very compact FPGA implementation of the AES algorithm. In C. D. Walter, Ç. K. Koç, and C. Paar, editors, *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)*, volume 2779 of *LNCS*, pages 319–333. Springer-Verlag, 2003.

[CIKL05] D. L. Cook, J. Ioannidis, A. D. Keromytis, and J. Luck. CryptoGraphics: Secret key cryptography using graphics cards. In *RSA Conference, Cryptographer's Track (CT-RSA)*, February 2005.

[cKKAK96] Ç. K. Koç, T. Acar, and B. S. Kaliski, Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996.

[CKM00] D. Coppersmith, L. R. Knudsen, , and C. J. Mitchell. Key Recovery and Forgery Attacks on the Mac DES MAC Algorithm. In M. Bellare, editor, *Advances in Cryptology – Proceedings of CRYPTO 2000*, volume 1880 of *LNCS*, pages 184–196. Springer-Verlag, 2000.

[CKVS06] R. Chaves, G. Kuzmanov, S. Vassiliadis, and L. Sousa. Reconfigurable memory based AES co-processor. In *Proceedings of the Workshop on Reconfigurable Architectures (RAW 2006)*, page 192, 2006.

[Com90] P. G Comba. Exponentiation Cryptosystems on the IBM PC. *IBM Systems Journal*, Vol. 29(4):526–538, 1990.

[Com07] Common Criteria Project. Common Criteria and Common Evaluation Methodology Version 3.1R2, September 2007. Available at `http://www.commoncriteriaportal.org`.

[CR06] C. De Cannière and C. Rechberger. Finding SHA-1 Characteristics. In *Advances in Cryptology – Proceedings of ASIACRYPT 2006*, volume 4284 of *LNCS Series*, pages 1–20. Springer-Verlag, 2006.

[CS07] N. Costigan and M. Scott. Accelerating SSL using the vector processors in IBM's Cell broadband engine for Sony's Playstation 3. *Workshop on Software Performance Enhancement for Encryption and Decryption (SPEED 2007)*, 2007.

[CUD07] NVIDIA Corporation, Santa Clara, CA, USA. *Compute Unified Device Architecture (CUDA) Programming Guide, Version 1.0*, 2007.

[dDBQ07] G.M. de Dormale, P. Bulens, and J.J. Quisquater. Collision Search for Elliptic Curve Discrete Logarithm over GF (2^m) with FPGA. In P. Paillier and I. Verbauwhede, editors, *Proceedings of the Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2007)*, volume 4727 of *LNCS*, page 378. Springer-Verlag, 2007.

[dDQ07] G. M. de Dormale and J.-J. Quisquater. High-speed hardware implementations of Elliptic Curve Cryptography: A survey. *Journal of Systems Architecture*, 53(2-3):72–84, 2007.

[Den82] D. E. R. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[DGP08] S. Drimer, T. Güneysu, and C. Paar. DSPs, BRAMs and a pinch of logic: new recipes for AES on FPGAs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2008)*, pages 99–108. IEEE Computer Society, April 2008.

[DH76] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, 1976.

[DH77] W. Diffie and M. E. Hellman. Exhaustive cryptanalysis of the NBS Data Encryption Standard. *COMPUTER*, 10(6):74–84, June 1977.

[dMGdDQ07] G. de Meulenaer, F. Gosset, M. M. de Dormale, and J.-J. Quisqater. Integer Factorization Based on Elliptic Curve Method: Towards Better Exploitation of Reconfigurable Hardware. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, pages 197–206. IEEE Computer Society, 2007.

[DMKP04] A. Daly, W. Marnane, T. Kerins, and E. Popovici. An FPGA implementation of a GF(p) ALU for encryption processors. *Elsevier - Microprocessors and Microsystems*, 28(5–6):253–260, 2004.

[DR02] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

[Dri07] S. Drimer. Authentication of FPGA bitstreams: Why and how. In *Proceedings of the International Workshop on Applied Reconfigurable Computing (ARC 2007)*, 2007.

[Dri09] S. Drimer. *Security for volatile FPGAs*. PhD thesis, Cambridge University, 2009. to appear.

[ECR07] ECRYPT. *eBATS: ECRYPT Benchmarking of Asymmetric Systems*, March 2007. Available at `http://www.ecrypt.eu.org/ebats/`.

[Edw07] H. M. Edwards. A normal form for elliptic curves. *Bulletin-American Mathematical Society*, 44(3):393, 2007. `http://www.ams.org/bull/2007-44-03/S0273-0979-07-01153-6/S0273-0979-07-01153-6.pdf`.

[EGCS03] H. Eberle, N. Gura, and S. Chang-Shantz. A cryptographic processor for arbitrary elliptic curves over GF(2^m). In *Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 444–454, 2003.

[EGP+07a] T. Eisenbarth, T. Güneysu, C. Paar, A.-R. Sadeghi, D. Schellekens, and M. Wolf. Reconfigurable trusted computing in hardware. In *Proceedings of the ACM Workshop on Scalable Trusted Computing (STC '07)*, pages 15–20, New York, NY, USA, 2007. ACM Press.

[EGP+07b] T. Eisenbarth, T. Güneysu, C. Paar, A.-R. Sadeghi, M. Wolf, and R. Tessier. Establishing Chain of Trust in Reconfigurable Hardware. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, pages 289–290, Washington, DC, USA, 2007. IEEE Computer Society.

[Ele98] Electronic Frontier Foundation. *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design*. O'Reilly & Associates Inc., July 1998.

[Elg85] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[EYCP01] A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists. *IEEE Transactions on Very Large Scale Integration Systems (VLSI)*, 9(4):545–557, 2001.

[FD01] V. Fischer and M. Drutarovský. Two methods of Rijndael implementation in reconfigurable hardware. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Proceedings of the Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2001)*, volume 2162 of *LNCS*, pages 77–92. Springer-Verlag, 2001.

[FFK+09] J. Fougeron, L. Fousse, A. Kruppa, D. Newman, and P. Zimmermann. GMP-ECM Website, February 2009. Available at `http://www.komite.net/laurent/soft/ecm/ecm-6.0.1.html`.

[FKP+05] J. Franke, T. Kleinjung, C. Paar, J. Pelzl, and C. Priplataand C. Stahlke. SHARK – A Realizable Special Hardware Sieving Device for Factoring 1024-bit Integers. In J. R. Rao and B. Sunar, editors, *Proceedings of the Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2005)*, volume 3659 of *LNCS*, pages 119–130. Springer-Verlag, August 2005.

[FL05] J. Fry and M. Langhammer. RSA & Public Key Cryptography in FPGAs. Technical report, Altera Corporation, 2005.

[Fle07] S. Fleissner. GPU-accelerated Montgomery exponentiation. In Y. Shi, G. D. Albada, J. Dongarra, and P. M. A. Sloot, editors, *Proceedings of the International Conference on Computational Science (ICCS 2007)*, volume 4487 of *LNCS*, pages 213–220. Springer-Verlag, 2007.

[FO90] P. Flajolet and A. M. Odlyzko. Random mapping statistics. *Lecture Notes in Computer Science*, 434:329–354, 1990.

[FSV07] J. Fan, K. Sakiyama, and I. Verbauwhede. Montgomery Modular Multiplication Algorithm on Multi-Core Systems. In *Proceedings of the Workshop on Signal Processing Systems (SPS 2007)*, pages 261–266, 2007.

[GB05] T. Good and M. Benaissa. AES on FPGA from the fastest to the smallest. In J. R. Rao and B. Sunar, editors, *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES 2005)*, volume 3659 of *LNCS*, pages 427–440. Springer-Verlag, 2005.

[GKB+06a] K. Gaj, S. Kwon, P Baier, P. Kohlbrenner, H. Le, M. Khaleeluddin, and R. Bachimanchi. Implementing the Elliptic Curve Method of Factoring in Reconfigurable Hardware. In L. Goubin and M. Matsui, editors, *Proceedings of the Workshop*

on Cryptograpic Hardware and Embedded Systems (CHES 2006), volume 4249 of *LNCS*, pages 119–133. Springer-Verlag, 2006.

[GKB+06b] K. Gaj, S. Kwon, P. Baier, P. Kohlbrenner H. Le, M. Khaleeluddin, and R. Bachimanchi. Implementing the Elliptic Curve Method of Factoring in Reconfigurable Hardware. In *Presented at the Workshop on Special Purpose Hardware for Attacking Cryptographic Systems (SHARCS'06)*, 2006.

[GKN+08] T. Güneysu, T. Kasper, M. Novotný, C. Paar, and A. Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers*, 57(11):1498–1513, November 2008.

[GKST07] J. Guajardo, S. Kumar, G.-J. Schrijen, and P. Tuyls. FPGA intrinsic PUFs and their use for IP protection. In P. Paillier and I. Verbauwhede, editors, *Proceedings of the Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2007)*, volume 4727 of *LNCS*, pages 63–80. Springer-Verlag, 2007.

[GMP07a] T. Güneysu, B. Möller, and C. Paar. Dynamic Intellectual Property Protection for Reconfigurable Devices. In *Proceedings of the IEEE International Conference on Field-Programmable Technology (ICFPT 2007)*, pages 169–176. IEEE Computer Society, 2007.

[GMP07b] T. Güneysu, B. Möller, and C. Paar. New Protection Mechanisms for Intellectual Property in Reconfigurable Logic. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, pages 287–288, Washington, DC, USA, 2007. IEEE Computer Society.

[GNR08] T. Gendrullis, M. Novotný, and A. Rupp. A Real-World Attack Breaking A5/1 within Hours. In E. Oswald and P. Rohatgi, editors, *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES 2008)*, volume 5154 of *LNCS*, pages 266–282. Springer-Verlag, 2008.

[GP08] T. Güneysu and C. Paar. Ultra High Performance ECC over NIST Primes on Commercial FPGAs. In E. Oswald and P. Rohatgi, editors, *Proceedings of the Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2008)*, volume 5154 of *LNCS*, pages 62–78. Springer-Verlag, 2008.

[GPP07a] T. Güneysu, C. Paar, and J. Pelzl. Attacking elliptic curve cryptosystems with special-purpose hardware. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA 2007)*, pages 207–215. ACM Press, 2007.

[GPP+07b] T. Güneysu, C. Paar, J. Pelzl, G. Pfeiffer, M. Schimmler, and C. Schleiffer. Parallel Computing with Low-Cost FPGAs: A Framework for COPACOBANA. In *Proceedings of the Symposioum on Parallel Computing with FPGAs (ParaFPGA 2007)*, LNI, Jülich, Germany, September 2007. Springer-Verlag.

[GPP08] T. Güneysu, C. Paar, and J. Pelzl. Special-Purpose Hardware for Solving the Elliptic Curve Discrete Logarithm Problem. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 1(2):1–21, 2008.

[GPPS08] T. Güneysu, C. Paar, G. Pfeiffer, and M. Schimmler. Enhancing COPACOBANA for advanced applications in cryptography and cryptanalysis. In *Proceedings of the Conference on Field Programmable Logic and Applications (FPL 2008)*, pages 675–678, 2008.

[GRS07] T. Güneysu, A. Rupp, and S. Spitz. Cryptanalytic Time-Memory Tradeoffs on COPACOBANA. In *Proceedings of the Workshop on "Kryptologie in Theorie und Praxis" (INFORMATIK 2007)*, LNI, pages 205–209, Bremen, Germany, September 2007. Springer-Verlag.

[GT07] P. Gaudry and E. Thomé. The mp$\mathbb{F}q$ library and implementing curve-based key exchanges. *Workshop on Software Performance Enhancement for Encryption and Decryption (SPEED 2007)*, 2007.

[Gün06] T. Güneysu. Efficient Hardware Architectures for Solving the Discrete Logarithm Problem on Elliptic Curves. Master's thesis, Horst Görtz Institute, Ruhr University of Bochum, February 2006.

[Har07] M. Harris. Optimizing CUDA. In *Supercomputing 2007 Tutorial*, Reno, NV, USA, November 2007.

[HCD07] H. Hisil, G. Carter, and E. Dawson. Faster Group Operations on Special Elliptic Curves. Cryptology ePrint Archive, Report 2007/441, 2007. `http://eprint.iacr.org/`.

[Hel80] M. E. Hellman. A Cryptanalytic Time-Memory Trade-Off. In *IEEE Transactions on Information Theory*, volume 26, pages 401–406, 1980.

[Hel07] Helion Technology. High performance AES (Rijndael) cores for Xilinx FPGAs, 2007. `http://www.heliontech.com/downloads/aes_xilinx_helioncore.pdf`.

[Hel08] Helion Technology. SHA-1 IP Cores, 2008. Available at `www.heliontech.com`.

[HMV04] D. R. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, New York, 2004.

[HV04] A. Hodjat and I. Verbauwhede. A 21.54 Gbits/s fully pipelined AES processor on FPGA. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2004)*, pages 308–309. IEEE Computer Society, 2004.

[HW07] O. Harrison and J. Waldron. AES encryption implementation and analysis on commodity graphics processing unit. In P. Paillier and I. Verbauwhede, editors, *Proceedings of the Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2007)*, volume 4727 of *LNCS*, pages 209–226. Springer-Verlag, 2007.

[IKM00] T. Ichikawa, T. Kasuya, and M. Matsui. Hardware evaluation of the AES finalists. *AES Candidate Conference*, pages 13–14, 2000.

[Inc08a] Cray Inc. Cray XD1 Supercomputer, 2008. Available at `http://www.cray.com/downloads/FPGADatasheet.pdf`.

[Inc08b] Silicon Graphics Incorporated. SGI RASC Technology, 2008. `http://www.sgi.com/products/rasc/`.

[Int08] International Business Machines Inc. IBM Research: BlueGene, 2008. Available at `http://www.research.ibm.com/bluegene/`.

[Jär08] K. Järvinen. *Studies on High-Speed Hardware Implementations of Cryptographic Algorithms*. PhD thesis, Helsinki University of Technology, 2008.

[JTS03] K. U. Järvinen, M. T. Tommiska, and J. O. Skyttä. A fully pipelined memoryless 17.8 Gbps AES-128 encryptor. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA 2003)*, pages 207–215, New York, NY, USA, 2003. ACM Press.

[JY03] M. Joye and S.M. Yen. The Montgomery Powering Ladder. In B.S. Kaliski, Ç. K. Koç, and C. Paar, editors, *Proceedings of the Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2002)*, volume 2523 of *LNCS*, pages 291–302. Springer-Verlag, 2003.

[Kal95] B. S. (Jr.) Kaliski. The Montgomery Inverse and its Applications. *IEEE Transactions on Computers*, 44:1064–1065, 1995.

[Kea01a] T. Kean. Secure configuration of field programmable gate arrays. In *Proceedings of the Conference on Field-Programmable Logic and Applications (FPL 2001)*, Belfast, United Kingdom, 2001.

[Kea01b] T. Kean. Secure Configuration of Field Programmable Gate Arrays. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM 2001)*, Rohnert Park CA, 2001. IEEE Computer Society.

[Kea02] T. Kean. Cryptographic Rights Management of FPGA Intellectual Property Cores. In *Proceedings ACM Conference on FPGAs*, Monterey, CA, 2002.

[KG04] P. Kohlbrenner and K. Gaj. An embedded true random number generator for FPGAs. In R. Tessier and H. Schmit, editors, *Proceedings of the International*

Symposium on Field Programmable Gate Arrays (FPGA 2004), pages 71–78. ACM Press, 2004.

[KK99] O. Kömmerling and M. G. Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. In *Proceedings of the USENIX Workshop on Smartcard Technology, Chicago, USA*, 1999.

[KKL+05] U. Kühn, K. Kursawe, S. Lucks, A.-R. Sadeghi, and C. Stüble. Secure Data Management in Trusted Computing. In J. R. Rao and B. Sunar, editors, *Proceedings of the Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2005)*, volume 3659 of *LNCS*, pages 324–338. Springer-Verlag, 2005.

[KKSS00] S. Kawamura, M. Koike, F. Sano, and A. Shimbo. Cox-rower architecture for fast parallel Montgomery multiplication. In B. Preneel, editor, *Advances in Cryptology – Proceedings of EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 523–538. Springer-Verlag, 2000.

[KO63] A. Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics – Doklady*, 7(7):595–596, 1963.

[Kob87] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[Koç95] Ç. K. Koç. RSA Hardware Implementation. Technical report TR801, RSA Data Security, Inc., 1995. Available at `http://islab.oregonstate.edu/koc/docs/r02.pdf`.

[KPP+06a] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, A. Rupp, and M. Schimmler. How to Break DES for € 8,980. In *Presented at the Workshop on Special Purpose Hardware for Attacking Cryptographic Systems (SHARCS'06)*, 2006.

[KPP+06b] S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler. Breaking Ciphers with COPACOBANA - A Cost-Optimized Parallel Code Breaker. In L. Goubin and M. Matsui, editors, *Proceedings of the Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2006)*, volume 4249 of *LNCS*, pages 101–118. Springer-Verlag, 2006.

[KSP05] K. Kursawe, D. Schellekens, and B. Preneel. Analyzing trusted platform communication. In *Presented at Cryptographic Advances in Secure Hardware (CRASH 2005)*, page 8, Leuven,Belgium, 2005.

[Len87] H. Lenstra. Factoring Integers with Elliptic Curves. *Annals of Mathematics*, 126:649–673, 1987.

[Lim04] D. Lim. Extracting secret keys from integrated circuits. Master's thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2004.

[LKLRP07] Y. Liu, T. Kasper, K. Lemke-Rust, and C. Paar. E-passport: Cracking basic access control keys. In R. Meersman and Z. Tari, editors, *Proceedings of On the Move to Meaningful Internet Systems Conferences (OTM 2007)*, volume 4804 of *LNCS*, pages 1531–1547. Springer-Verlag, 2007.

[LL93] A. K. Lenstra and H. W. Jr. Lenstra, editors. *The Development of the Number Field Sieve*, volume 1554 of *LNM*. Springer-Verlag, 1993.

[LV01] A. K. Lenstra and E. R. Verheul. Selecting Cryptographic Key Sizes. *Journal of Cryptology*, 14(4):255–293, 2001.

[Man07] S. A. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *Proceedings of IEEE's International Conference on Signal Processing and Communication (ICSPC 2007)*, pages 65–68. IEEE Computer Society, 2007.

[MBPV06] N. Mentens, L. Batina, B. Prenel, and I. Verbauwhede. Time-Memory Trade-Off Attack on FPGA Platforms: UNIX Password Cracking. In *Proc. of ARC 2006*, volume 3985 of *LNCS*, pages 323–334. Springer, 2006.

[Men07] N. Mentens. *Secure and Efficient Coprocessor Design for Cryptographic Applications on FPGAs*. PhD thesis, Katholieke Universiteit Leuven, Leuven-Heverlee, Belgium, June 2007.

[Mic06] Microsoft Corporation. Bitlocker drive encryption: Technical overview, April 2006. Available at `http://www.microsoft.com/technet/windowsvista/security/bittech.mspx`.

[Mil86] V. Miller. Uses of Elliptic Curves in Cryptography. In H. C. Williams, editor, *Advances in Cryptology – Proceedings of CRYPTO 1985*, volume 218 of *LNCS*, pages 417–426. Springer-Verlag, 1986.

[MM01] M. McLoone and J.V. McCanny. High performance single-chip FPGA Rijndael algorithm implementations. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Proceedings of the Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2001)*, volume 2162 of *LNCS*, pages 65–76. Springer-Verlag, 2001.

[MM03] M. McLoone and J.V. McCanny. Rijndael FPGA implementations utilising look-up tables. *The Journal of VLSI Signal Processing*, 34(3):261–275, 2003.

[MMM04] C. McIvor, M. McLoone, and J. McCanny. An FPGA elliptic curve cryptographic accelerator over GF(p). In *Irish Signals and Systems Conference (ISSC)*, pages 589–594, 2004.

[Mon85] P. L. Montgomery. Modular Multiplication without Trial Division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[Mon87] P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.

[MPS07] A. Moss, D. Page, and N. Smart. Toward acceleration of RSA using 3d graphics hardware. In *Cryptography and Coding*, volume 4887 of *LNCS*, pages 369–388. Springer, 2007.

[MRSM03] A. Mazzeo, L. Romano, G. P. Saggese, and N. Mazzocca. FPGA-based implementation of a serial RSA processor. *Proceedings of Conference and Exposition on Design, Automation and Test in Europe (DATE 2003)*, pages 10582–10589, 2003.

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

[Nat77] National Institute for Standards and Technology (NIST). FIPS PUB 46-3: Data Encryption Standard, January 1977.

[Nat99] National Institute of Standards and Technology (NIST). Recommended Elliptic Curves for Federal Government Use, July 1999.

[Nat00] National Institute of Standards and Technology (NIST). Digital signature standard (DSS) (FIPS 186-2), January 2000.

[Nat01] National Institute of Standards and Technology (NIST). FIPS PUB 197: Advanced Encryption Standard, 2001.

[Nat05] National Institute of Standards and Technology (NIST). Recommendation for block cipher modes of operation – the CMAC mode for authentication. NIST Special Publication SP 800-38B, 2005.

[Nat06] National Institute of Standards and Technology (NIST). Recommendation for pair-wise key establishment schemes using discrete logarithm cryptography. NIST Special Publication SP 800-56A, 2006.

[NMSK01] H. Nozaki, M. Motoyama, A. Shimbo, and S. Kawamura. Implementation of RSA algorithm based on RNS Montgomery multiplication. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Proceedings of the Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2001)*, volume 2162 of *LNCS*, pages 364–376. Springer-Verlag, 2001.

[OBPV03] S. B. Örs, L. Batina, B. Preneel, and J. Vandewalle. Hardware implementation of elliptic curve processor over $GF(p)$. In *Proceedings of the Application-Specific Systems, Architectures and Processors (ASAP 2003)*, pages 433–443, 2003.

[Oec03] P. Oechslin. Making a Faster Cryptanalytic Time-Memory Trade-Off. In *Proc. of CRYPTO 2003*, volume 2729 of *LNCS*, pages 617–630. Springer, 2003.

[ÖOP03] S. B. Örs, E. Oswald, and B. Preneel. Power-analysis attacks on an FPGA – first experimental results. In C. D. Walter, Ç. K. Koç, and C. Paar, editors, *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)*, volume 2779 of *LNCS*, pages 35–50. Springer-Verlag, 2003.

[OP00] G. Orlando and C. Paar. A High-Performance Reconfigurable Elliptic Curve Processor for GF(2^m). In Ç. K. Koç and C. Paar, editors, *Proceedings of the Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2000)*, volume 1965 of *LNCS*, pages 41–56. Springer-Verlag, 2000.

[OP01] G. Orlando and C. Paar. A Scalable $GF(p)$ Elliptic Curve Processor Architecture for Programmable Hardware. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Proceedings of the Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2001)*, volume 2162 of *LNCS*, pages 356–371. Springer-Verlag, 2001.

[Oru95] H. Orup. Simplifying quotient determination in high-radix modular multiplication. In *ARITH '95: Proceedings of the 12th Symposium on Computer Arithmetic*, page 193, Washington, DC, USA, 1995. IEEE Computer Society.

[P1300] Institute of Electrical and Electronics Engineers. *IEEE P1363 Standard Specifications for Public Key Cryptography*, 2000.

[PO96] B. Preneel and P.C. Van Oorschot. Key recovery attack on ANSI X9.19 retail MAC. In *Electronics Letters*, volume 32 of *17*, pages 1568–1569. IEEE, Dept. of Electr. Eng., Katholieke Univ., Leuven, 1996.

[Poe06] B. Poettering. SECCURE Elliptic Curve Crypto Utility for Reliable Encryption, version 0.3, August 2006. Available at `http://point-at-infinity.org/seccure/`.

[Pol74] J. M. Pollard. Theorems on factorization and primality testing. In *Proceedings of the Cambridge Philosophy Society*, pages 521–528, 1974.

[Pol75] J. M. Pollard. Monte Carlo method for factorization. *Nordisk Tidskrift for Informationsbehandlung (BIT)*, 15:331–334, 1975.

[Pol78] J. M. Pollard. Monte Carlo methods for index computation mod p. *Mathematics of Computation*, 32(143):918–924, July 1978.

[PTX07] NVIDIA Corporation, Santa Clara, CA, USA. *Parallel Thread Execution (PTX) ISA Release 1.0*, 2007.

[Ros07] U. Rosenberg. Using Graphic Processing Unit in Block Cipher Calculations. Master's thesis, University of Tartu, Tartu, Estonia, 2007. Available at `http://dspace.utlib.ee/dspace/bitstream/10062/2654/1/rosenberg_urmas.pdf`.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[RSA07] RSA - The Security Division of EMC^2. RSA SecurID, 2007. `http://www.rsa.com/`.

[RSQL03] G. Rouvroy, F.-X. Standaert, J.-J. Quisquater, and J.-D. Legat. Design Strategies and Modified Descriptions to Optimize Cipher FPGA Implementations: Fast and Compact Results for DES and Triple-DES. In *Field-Programmable Logic and Applications - FPL*, pages 181–193, 2003.

[RSQL04] G. Rouvroy, F.-X. Standaert, J.-J. Quisquater, and J.-D. Legat. Compact and Efficient Encryption/Decryption Module for FPGA Implementation of the AES Rijndael Very Well Suited for Small Embedded Applications. *International Conference on Information Technology: Coding and Computing*, 2:583, 2004.

[SBG+03] R. Schroeppel, C. Beaver, R. Gonzales, R. Miller, and T. Draelos. A Low-Power Design for an Elliptic Curve Digital Signature Chip. In B.S. Kaliski, Ç. K. Koç, and C. Paar, editors, *Proceedings of the Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2002)*, volume 2523 of *LNCS*, pages 53–64. Springer-Verlag, 2003.

[SFCK04] E. Savas, A. F.Tenca, M. E. Ciftcibasi, and C. K. Koc. Multiplier architectures for GF(p) and GF(2^n). *IEE Proceedings Computers & Digital Techniques*, 151(2):147–160, 2004.

[SG08] R. Szerwinski and T. Güneysu. Exploiting the Power of GPUs for Asymmetric Cryptography. In E. Oswald and P. Rohatgi, editors, *Proceedings of the Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2008)*, volume 5154 of *LNCS*, pages 79–99. Springer-Verlag, 2008.

[Sha71] D. Shanks. Class number, a theory of factorization and genera. *Proceedings of Symposia in Pure Mathematics*, 20:415–440, 1971.

[SK89] A. P. Shenoy and R. Kumaresan. Fast Base Extension Using a redundant Modulus in RNS. *IEEE Transactions on Computers*, 38(2):292–297, February 1989.

[Sma01] N. P. Smart. The Hessian form of an elliptic curve. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Proceedings of the Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2001)*, volume 2162 of *LNCS*, pages 118–125. Springer-Verlag, 2001.

[SMS07] B. Sunar, W. J. Martin, and D. R. Stinson. A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks. *IEEE Transactions on Computers*, 56(1):109–119, January 2007.

[Sol99] J. A. Solinas. Generalized Mersenne Numbers. Technical report, National Security Agency (NSA), September 1999. Available at `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.46.2133\&rep=rep1\&type=pdf`.

[SÖP04] F.-X. Standaert, S. B. Örs, and B. Preneel. Power analysis of an FPGA implementation of Rijndael: Is pipelining a DPA countermeasure? In M. Joye and J.-J. Quisquater, editors, *Proceedings of the Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2004)*, volume 3156 of *LNCS*, pages 30–44. Springer-Verlag, 2004.

[ŠPK+05] M. Šimka, J. Pelzl, T. Kleinjung, J. Franke, C. Priplata, C. Stahlke, M. Drutarovský, V. Fischer, and C. Paar. Hardware Factorization Based on Elliptic Curve Method. In J. Arnold and K. L. Pocek, editors, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2005)*, pages 107–116. IEEE Computer Society, April 18-20 2005.

[SRQL03a] F.-X. Standaert, G. Rouvroy, J. Quisquater, and J. Legat. A Time-Memory Tradeoff using Distinguished Points: New Analysis & FPGA Results. In B.S. Kaliski, Ç. K. Koç, and C. Paar, editors, *Proceedings of the Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2002)*, volume 2523 of *LNCS*, pages 596–611. Springer-Verlag, 2003.

[SRQL03b] F.-X. Standaert, G. Rouvroy, J.-J. Quisquater, and J.-D. Legat. Efficient implementation of Rijndael encryption in reconfigurable hardware: improvements and design tradeoffs. In C. D. Walter, Ç. K. Koç, and C. Paar, editors, *Proceedings of the Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2003)*, volume 2779 of *LNCS*, pages 334–350. Springer-Verlag, 2003.

[SS06] E. Simpson and P. Schaumont. Offline Hardware/Software Authentication for Reconfigurable Platforms. In L. Goubin and M. Matsui, editors, *Proceedings of the Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2006)*, volume 4249 of *LNCS*, pages 311–323. Springer-Verlag, 2006.

[ST67] N. S. Szabó and R. I. Tanaka. *Residue Arithmetic and its Applications to Computer Technology.* McGraw - Hill Inc., USA, 1967.

[ST03a] A. Satoh and K. Takano. A scalable dual-field elliptic curve cryptographic processor. *IEEE Transactions on Computers*, 52(4):449–460, 2003.

[ST03b] A. Shamir and E. Tromer. Factoring Large Numbers with the TWIRL Device. In D. Boneh, editor, *Advances in Cryptology – Proceedings of CRYPTO 2003*, volume 2729 of *LNCS*, pages 1–26. Springer-Verlag, 2003.

[Sti05] D. R. Stinson. *Cryptography. Theory and Practice.* Taylor & Francis, 3rd edition, 2005.

[Suz07] D. Suzuki. How to maximize the potential of FPGA Resources for Modular Exponentiation. In P. Paillier and I. Verbauwhede, editors, *Proceedings of the Workshop on Cryptograpic Hardware and Embedded Systems (CHES 2007)*, volume 4727 of *LNCS*, pages 272–288. Springer-Verlag, 2007.

[SZJvD04] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the USENIX Workshop on Smartcard Technology, Boston, USA*, pages 223–238, 2004.

[Tes98] E. Teske. Speeding Up Pollard's Rho Method for Computing Discrete Logarithms. *Algorithmic Number Theory Seminar ANTS-III*, 1423:541–554, 1998.

[TraAD] G. Suetonius Tranquillus. De vita Caesarum (*transl.: On the Life of the Caesars*). Volume 1: Life of Julius Caesar, 121 AD.

[Tru06] Trusted Computing Group (TCG). TPM specification, version 1.2 revision 94, March 2006. Available at `http://www.trustedcomputinggroup.org/specs/TPM/`.

[Tru08] Trusted Computing Group (TCG). About the TCG, 2008. Available at `http://www.trustedcomputinggroup.org/about/`.

[Uni05] University of California, Berkeley. Seti@Home Website, 2005. `http://setiathome.berkeley.edu/`.

[vOW99] P. C. van Oorschot and M. J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *Journal of Cryptology*, 12(1):1–28, 1999.

[WBV+96] E. De Win, A. Bosselaers, S. Vandenberghe, P. De Gersem, and J. Vandewalle. A fast software implementation for arithmetic operations in GF(2n). In *Advances in Cryptology – Proceedings of ASIACRYPT 1996*, volume 1163 of *LNCS*, pages 65–76. Springer-Verlag, 1996.

[Wie96] M. J. Wiener. Efficient DES Key Search. In William R. Stallings, editor, *Practical Cryptography for Data Internetworks*, pages 31–79. IEEE Computer Society Press, 1996.

[Wil82] H. C. Williams. A $p + 1$ method of factoring. *Mathematics of Computation*, 39:225–234, 1982.

[WP03] T. Wollinger and C. Paar. How secure are FPGAs in cryptographic applications, 2003.

[Xil06] Xilinx Inc. *UG190: Virtex-5 user guide*, 2006. Available at `http://www.xilinx.com/support/documentation/user_guides/ug190.pdf`.

[Xil07] Xilinx Inc. *UG193: Virtex-5 XtremeDSP design considerations user guide*, 2007. Available at `http://www.xilinx.com/support/documentation/user_guides/ug193.pdf`.

[Xil08a] Xilinx Inc. Xilinx' History of FPGA Development, 2008. Available at `http://www.xilinx.com/company/history.htm`.

[Xil08b] Xilinx Inc. Xilinx Spartan-3 and Virtex FPGA devices, 2008. Available at `www.xilinx.com/products/silicon_solutions/`.

[ZD06] P. Zimmermann and B. Dodson. 20 years of ECM. In F. Hess, S. Pauli, and M. E. Pohst, editors, *Proceedings of the Symposium on Algorithmic Number Theory Symposium (ANTS 2006)*, volume 4076 of *LNCS*, pages 525–542. Springer, 2006.

[ZHC+06] J. Zambreno, D. Honbo, A. Choudhary, R. Simha, and B. Narahar. High-performance software protection using reconfigurable architectures. *Proceedings of the IEEE*, 94:419 – 431, Feb. 2006.

[Zim09] P. Zimmermann. 50 largest factors found by ECM, February 2009. Available at `http://www.loria.fr/~zimmerma/records/top50.html`.

List of Figures

List of Tables

List of Abbreviations

AES	Advanced Encryption Standard
ANSI	American National Standards Institute
ASIC	Application-Specific Integrated Circuit
AU	Arithmetic Unit
BRAM	Block Random-Access Memory
CBC	Cipher Block Chaining
CIOS	Coarsely Integrated Operand Scanning
CLB	Configurable Logic Block
CMAC	CBC-based Message Authentication Mode
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
CRT	Chinese Remainder Theorem
CRTM	Core Root of Trust for Measurement
CTR	Counter
CUDA	Compute Unified Device Architecture
DES	Data Encryption Standard
DH	Diffie-Hellman
DLP	Discrete Logarithm Problem
DSA	Digital Signature Algorithm
DSP	Digital Signal Processing
EC	Elliptic Curve
ECB	Electronic Code Book
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECM	Elliptic Curve Method
FF	Flip-Flop
FIPS	Federal Information Processing Standard
FPGA	Field Programmable Gate Array
GE	Gate Equivalent
GPU	Graphics Processing Unit
GSM	Global System for Mobile Communications
HM	Hardware Manufacturer
IP	Intellectual Property

IPO	Intellectual Property Owner
IT	Information Technology
KDF	Key Derivation Function
LAL	Look-Ahead Logic
LS	Logic Slice
LUT	Look-Up Table
MAC	Message Authentication Code
MPPR	Multi-Processor Pollard-Rho
NFS	Number Field Sieve
NIST	National Institute of Standards and Technology
OP	Operation
OTP	One-Time Password
PIN	Personal Identification Number
PK	Public Key
PKI	Public Key Infrastructure
PUF	Physical Unclonable Function
RFID	Radio Frequency Identification
RNG	Random Number Generator
RNS	Residue Number Systems
RSA	Rivest Shamir Adleman
SK	Secret Key
SECG	Standards for Efficient Cryptography Group
SI	System Integrator
SIMD	Single Instruction Multiple Data
SoC	System-on-a-Chip
SPPR	Single-Processor Pollard-Rho
TC	Trusted Computing
TCG	Trusted Computing Group
TMTO	Time-Memory Tradeoff
TPM	Trusted Platform Module
TTP	Trusted Third Party
XOR	Exclusive-OR

About the Author

Personal Data

■ Name	Tim Erhan Güneysu
■ Date of Birth	October 3rd, 1979
■ Place of Birth	Werne Westf., Germany

Short Resume*

■ Feb 2007 – *present*	Fellow of Research School at Ruhr-University Bochum
■ Mar 2006 – *present*	Research associate of Embedded Security Group (EMSEC) at Ruhr-University Bochum
■ Oct 2003 – Jan 2006	Study of "IT Security" at Ruhr University Bochum
■ Sep 2002 – Jan 2003	Semester abroad at Staffordshire University, UK
■ Jun 2002 – Sep 2002	Scholarship by Carl-Duisberg Association for internship at IBM Almaden Research Lab, San José , USA
■ Oct 2000 – Sep 2003	Study of "Information Technology International" at University of Coorporative Education Mannheim

Research and Industry Projects

■ 2007 – *present*	BSI Germany: Factorization of Large Integers
■ 2007 – *present*	Sciengines GmbH: Development/Improvement of Cryptanalytic Applications
■ 2006 – 2007	escrypt GmbH: ECDSA Hardware Accelerator on Xilinx Virtex-4 devices
■ 2003 – 2006	IBM Germany GmbH: Lotus Notes/Domino Development
■ 2000 – 2003	IBM Germany GmbH: Internships/apprenticeship

*As of February 2009.

Publications

The author of this thesis has worked in several research areas. The following contributions to the cryptographic and reconfigurable computing community were published (*January 2009*).

Book Chapters

- Tim Güneysu, Christof Paar. Modular Integer Arithmetic for Public-key Cryptography. Chapter in *Secure Integrated Circuits and Systems*, Editor I. Verbauwhede, to appear in Kluwer-Verlag, 2009.

Journals

- Tim Güneysu, Timo Kasper, Martin Novotny, Christof Paar, Andy Rupp. Cryptanalysis with COPACOBANA. In *IEEE Transactions on Computers, IEEE Computer Society*, volume 57, number 11, pp. 1498–1513 November 2008.

- Tim Güneysu, Christof Paar, Jan Pelzl. Special-Purpose Hardware for Solving the Elliptic Curve Discrete Logarithm Problem. In *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, volume 1, number 2, pp. 1–21, June 2008.

- Jorge Guajardo, Tim Güneysu, Sandeep Kumar, Christof Paar, Jan Pelzl. Efficient Hardware Implementation of Finite Fields with Applications to Cryptography. In *Acta Applicandae Mathematicae: An International Survey Journal on Applying Mathematics and Mathematical Applications*, volume 93, numbers 1-3, pp. 75–118, September 2006.

International Conferences & Workshops

- Tim Güneysu, Christof Paar, Gerd Pfeiffer, Manfred Schimmler. Enhancing COPACOBANA for Advanced Applications in Cryptography and Cryptanalysis. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), Heidelberg, Germany, IEEE Computer Society, September 2008.*

- Tim Güneysu, Christof Paar. Ultra High Performance ECC over NIST Primes on Commercial FPGAs. In *Proceedings of the Cryptographic Hardware and Embedded Systems, LNCS Series, Washington D.C., USA, Springer-Verlag, August 2008.*

- Robert Szerwinski, Tim Güneysu. Exploiting the Power of GPUs for Asymmetric Cryptography. In *Proceedings of the Cryptographic Hardware and Embedded Systems, LNCS Series, Washington D.C., USA, Springer-Verlag, August 2008.*

- Saar Drimer, Tim Güneysu and Christof Paar. DSPs, BRAMs and a Pinch of Logic: New recipes for AES on FPGAs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, Stanford University, CA, USA, IEEE Computer Society, April 2008.*

- Tim Güneysu, Christof Paar. Breaking Legacy Banking Standards with Special-Purpose Hardware. In *Proceedings of the Conference on Financial Cryptography and Data Security (FC08), Cozumel, Mexico, LNCS Series, Springer-Verlag, January 2008.*

- Tim Güneysu, Bodo Möller, Christof Paar. Dynamic Intellectual Property Protection for Reconfigurable Devices. In *Proceedings of the IEEE Conference on Field-Programmable Technology, (ICFPT), Kitakyushu, Japan, IEEE Computer Society, pp. 169-176, December 2007.*

- Thomas Eisenbarth, Tim Güneysu, Christof Paar, Ahmad-Reza Sadeghi, Dries Schellekens, Marko Wolf. Reconfigurable Trusted Computing in Hardware. In *Proceedings of the ACM Conference on Scalable Trusted Computing, (STC), Alexandria, VA, USA. ACM Press. November 2007.*

- Tim Güneysu, Christof Paar, Sven Schäge. Efficient Hash Collision Search Strategies on Special-Purpose Hardware. In *Proceedings of the Western European Workshop on Research in Cryptology (WeWORC), Bochum, Germany, LNCS Series, Springer-Verlag, July 2007.*

- Tim Güneysu, Bodo Möller, Christof Paar. New Protection Mechanisms for Intellectual Property in Reconfigurable Logic. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), IEEE Computer Society, Napa, CA, USA, April 2007.*

- Thomas Eisenbarth, Tim Güneysu, Christof Paar, Ahmad-Reza Sadeghi, Russell Tessier, Marko Wolf. Establishing Chain of Trust in Reconfigurable Hardware. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), IEEE Computer Society, Napa, CA, USA, April 2007.*

- Tim Güneysu, Christof Paar, Jan Pelzl. Attacking Elliptic Curve Cryptosystems with Special-Purpose Hardware. In *Proceedings of the International Symposium on Field Programmable Gate Arrays, ACM Press, Monterey, CA, USA, Februar 2007.*

National Conferences & Workshops

- Tim Güneysu, Andy Rupp, Stefan Spitz. Cryptanalytic Time-Memory Tradeoffs on COPACOBANA. In *INFORMATIK 2007 – Informatik trifft Logistik, Workshop: "Kryptologie in Theorie und Praxis", 37. Jahrestagung der Gesellschaft für Informatik e. V. (GI), LNI, Springer-Verlag, Bremen, Germany, September 2007.*

- Tim Güneysu, Christof Paar, Jan Pelzl, Gerd Pfeiffer, Manfred Schimmler and Christian Schleiffer. Parallel Computing with Low-Cost FPGAs: A Framework for COPACOBANA. In *Proceedings of the Symposium on Parallel FPGA Architecture (ParaFPGA), LNI, Springer-Verlag, Jülich, Germany, September 2007.*

Invited Talks

- Tim Güneysu. High Performance ECC over NIST Primes on Commercial FPGAs. *12. Workshop on Elliptic Curve Cryptography (ECC 2008), Trianon Zalen, Utrecht, Niederlande, 22.-24 September 2008.*

- Tim Güneysu. COPACOBANA - Cost-Optimized Parallel Code Breaker. *"Deutschland Land der Ideen", University of Siegen, 22. Juli 2008.*

- Tim Güneysu. Accelerating ECM with Special-Purpose Hardware. *Workshop "Factoring Large Numbers, Discrete Logarithms and Cryptanalytical Hardware", Institute for Experimental Mathematics, University Duisburg-Essen, Germany, 22.-23. April 2008.*

- Tim Güneysu. Efficient Hardware Architectures for Solving the Discrete Logarithm Problem on Elliptic Curves. *ECC-Brainpool, NXP Semiconductors, Hamburg, Germany, 07. Juni 2006.*

Technical Reports

- Saar Drimer, Tim Güneysu, Markus Kuhn, Christof Paar. Protecting multiple cores in a single FPGA design. Research Report, Horst Görtz Institute for IT security in cooperation with University of Cambridge, September 2008.