Integrating Passive RFID Devices into the Internet of Things

Johannes Samhaber
johannes.samhaber@student.tugraz.at

Institute for Applied Information
Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a
8010 Graz, Austria

Master's Thesis

Supervisor: Dr. Michael Hutter, TU Graz
Assessor: Dr. Karl-Christian Posch, TU Graz

May, 2013

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am …… ……..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…… ……..
date (signature)

Acknowledgements

First of all I would like to thank my supervisor Michael Hutter of Graz University of Technology for his outstanding support through the course of this thesis. He willingly provided answers to all my questions, even at extraordinary hours and kept pushing me forward.

Next, I would like to thank my Assessor Dr. Karl-Christian Posch, who contributed to meeting the deadline by permitting to hand in the thesis at the last minute.

I would also like to thank Michael Höfler, a colleague and friend, for the endless technical discussions in all these years, which were often inspiring.

A very special thanks goes to my girlfriend, who always stood by me and encouraged me. Thank you for your understanding. I know it was not always easy.

Last but not least, I would like to thank my parents and grandparents for their unprecedented support through the course of my academic studies.

Abstract

The Internet of Things (IoT) is understood as a concept where numerous items are able to acquire and store data as well as communicate over the Internet. This thesis deals with the realization of the last step towards the Internet of Things: Integrating passive RFID tags into the Internet. A secure Internet layer is established upon the RFID-communication layer, which allows for secure end-to-end connection between tags and clients.

This thesis is separated into the design and implementation of an elliptic-curve cryptography (ECC) core in hardware, as well as the implementation of the basic IPsec functionality on the IAIK UHF DemoTag in software. The Internet Key Exchange Protocol version 2 (IKEv2) is used for key management between the tags and the clients. IKEv2 is a relatively complex protocol and demanding on the memory when considering resource-constrained RFID tags. Therefore, complexity is offloaded to the reader, which acts as a router for the tags and preprocesses the payloads. Some constraints are defined regarding the IKEv2 protocol in order to make it feasible for use with RFID tags. An elliptic curve Diffie-Hellman (ECDH) key exchange is required the IKEv2 protocol.

A major part of this thesis is the design and implementation of an ECC core in hardware. The focus was on finding a suitable design, i.e., a fast, and area efficient implementation. In order to keep the required chip area within limits, the smallest prime curve defined for use in IKEv2 was chosen, which is the NIST curve P-192 over the prime field $\mathbb{F}_{p192}$. An extensive design-space exploration was carried out, to find a good trade off between area requirements and cycle count for the execution of a point multiplication $Q = d * P$. For robustness against side-channel attacks, the ECC core was designed to have constant execution time, independent of the scalar value d. The decision fell in favor of a design which is very fast while keeping the area requirements within limits. The final design of the ECC core was synthesized using UMC 130 nm CMOS technology. It consumes an area of 24 kGE while only needing 287 k clock cycles for an elliptic-curve point multiplication, which is faster than most related work on ECC over prime fields.

Furthermore, we pave the way for a first Internet-of-Things prototype by providing all basic components and concepts such as IoT-enabled RFID tags that are able to communicate with arbitrary hosts, RFID readers that act as routers for IoT-enabled tags, and the definition and implementation of a secure Internet layer on top of the RFID protocol using custom RFID commands.

Keywords: Internet of Things, Elliptic Curve Cryptography, Radio Frequency Identification, ECDH, IKEv2, IPsec, ASIC

Kurzfassung

Das Internet der Dinge wird als Konzept verstanden bei dem zahlreiche Gegenstände dazu im Stande sind, Daten zu aquirieren und zu speichern, sowie über das Internet zu kommunizieren. Diese Abschlussarbeit behandelt die Realisierung des letzten Schrittes in Richtung Internet der Dinge: Die Integration von passiven RFID-Tags in das Internet. Eine Schicht für sichere Internet Anbindung wird auf die RFID-Kommunikationsschicht aufgesetzt, um sichere End-zu-End Verbindungen zwischen Tags und Nutzern zu ermöglichen.

Die Arbeit ist aufgeteilt in das Design und die Hardware-Implementation eines Elliptische Kurven Kryptographie (ECC) Cores, sowie die Implementation der grundlegenden IPsec-Funktionalität auf dem IAIK UHF-DemoTag in Software. Das Internet Key Exchange Protokoll Version 2 (IKEv2) wird für das Schlüsselmanagement zwischen Tags und Nutzern verwendet. IKEv2 ist ein relativ komplexes Protokoll und stellt hohe Anforderungen an den Speicher, im Kontext von ressourcenbeschränkten RFID Tags. Aus diesem Grund wird Komplexität auf das RFID Lesegerät übertragen, welches als Router für die Tags fungiert und die Nutzdaten für den Tag aufbereitet. Einige Einschränkungen bezüglich des IKEv2-Protokolls wurden definiert, um es für die Verwendung auf RFID Tags praktikabel zu machen. Teil des IKEv2-Protokolls ist ein Elliptische Kurven Diffie-Hellman (ECDH) Schlüsselaustausch.

Ein großer Teil dieser Abschlussarbeit behandelt das Design und die Implementation eines ECC Cores zu diesem Zweck in Hardware. Der Fokus lag auf dem Finden eines geeigneten Designs, das heißt, eine schnelle Implementation mit wenig Chipfläche. Um die Chipfläche in Grenzen zu halten, wurde die kleinste Primkurve gewählt, welche für die Verwendung für IKEv2 definiert ist. Es handelt sich dabei um die NIST-Kurve P-192 über den Primkörper $\mathbb{F}_{p192}$. Eine umfassende Design-Space Exploration wurde durchgeführt, um einen guten Kompromiss zwischen der Fläche und der Anzahl der benötigten Taktzyklen für die Ausführung einer Punktmultiplikation $Q = d * P$ zu finden. Um Robustheit gegen Seitenkanalattacken zu erreichen, wurde der Core hinsichtlich konstanter Ausführungszeit, unabhängig vom Skalar d, designed. Die Entscheidung fiel zugunsten eines sehr schnellen Designs, welches die benötigte Chipfläche gleichzeitig in Grenze hält. Das finale Design des ECC-Core wurde unter Benutzung einer 130 nm CMOS Technologie synthetisiert. Es benötigt eine Fläche von 24 kGE während die Ausführung einer Punktmultiplikation nur 287 k Taktzyklen benötigt, was schneller ist als die meißten vergleichbaren Publikationen betreffend ECC über Primkörper.

Darüber hinaus bahnen wir den Weg für einen ersten Internet der Dinge Prototyp, indem wir alle grundlegenden Komponenten und Konzepte zur Verfügung stellen, wie RFID tags mit Internet der Dinge Funktionalität, die im Stande sind mit beliebigen Hosts zu kommunizieren, RFID-Lesegeräte, die als Router für solche RFID-Tags fungieren und die Definition und Implementation einer sicheren Internetschicht aufbauend auf das RFID Protokoll unter Verwendung von spezifischen Befehlen.

Stichwörter: Internet der Dinge, Elliptische Kurven Kryptographie, Radio Frequenz Identifikation, ECDH, IKEv2, IPsec, ASIC

Contents

Chapter 1

Introduction

1.1 Motivation

The Internet of Things (IoT) is understood as a concept where numerous items are able to acquire and store data as well as communicate with each other over the Internet. By connecting arbitrary objects to the Internet, *smart things* can be realized. This opens up new possibilities to facilitate work and everyday life. A key technology for realizing this vision is Radio Frequency Identification (RFID). So-called RFID *tags* consist of an antenna attached to a microprocessor, enabling contact-less communication with an RFID *reader* device by a radio link. Typically, RFID tags are implemented in such a way that they do not have a dedicated power supply but draw their power from the electromagnetic field of the reader. RFID tags can be extended by sensors and, enabled by their small form factor, easily attached to objects, thus making them great candidates for the realization of Internet of Things-enabled items. The Internet of Things has only been made feasible by the introduction of Internet Protocol Version 6 (IPv6): In relation to it's predecessor IPv4 which has an address space of 32 bits, IPv6 has an address space of 128 bits. Thus, when going by the rough estimate of $0.51 * 10^{15} m^2$ for earth's surface, IPv6 theoretically provides more than $6.67 * 10^{23}$ IP addresses per square meter. This leaves plenty of addresses for use in the Internet of Things, even when going by pessimistic estimates for future address allocation. For the full specification of IPv6 we refer to [5].

Applications for the integration of RFID-enabled objects into the Internet are manifold as the following use cases show:

Use Case 1: Washing In [4], Darianian et al. described various smart home service use cases, such as washing, enabled by smart things. In their vision, the dirty clothes container and the washing machine each contain an RFID reader. An application monitors the amount of clothes in the container and generates an alarm if a certain threshold level is reached. Further, the washing machine can execute just the right washing program for certain clothes. Both services are enabled by the clothes containing RFID tags with Information regarding their material and color. By using Internet-enabled RFID tags, a person could use a Web service, e.g., to check which clothes currently are to be washed.

Use Case 2: Food Industry Internet-enabled RFID tags could be attached to food like raw meat, monitoring parameters like the temperature during processing and transportation. This could be used for automated alarming or sorting purposes as

well as updating a database at each (intermediate) RFID reader such that customers would be able to track the temperature and current location in real time.

Use Case 3: Stock-taking Stock-taking in stores is associated with costs and other efforts. Now, for example in a book store, an Internet-enabled RFID tag could be attached to each book. Once a book is delivered to the store, it passes an RFID reader. During this pass, the tag interacts with the reader in such a way that a database is updated with a distinct entry for the passing book. Once the book is sold, it passes the reader again, leading to another database update. Therefore, the database always contains the books that are actually in the store. If a customer wants to return a book, another database update can be triggered. So, the use of Internet-enabled RFID tags could make manual stock-checking obsolete. A further advantage of this approach is, that the destination or availability of a certain book is always up to date and can be retrieved by a simple database query.

While Use Case 1 describes a controlled environment where the owner of the washing machine and thus of the RFID reader usually would be the same person as the owner of the clothes and RFID tags, in general this is not true and the RFID readers can not be trusted. This becomes more apparent when looking closer into Use Case 2: The food may travel through whole cities or even countries. On it's way, it might pass numerous RFID readers trying to gather or access the data stored or acquired by the tags. Now there are several requirements to the RFID tags, like the following:

- Some information about the item should remain private, like, for example, personal information stored in the memory of the tags.
- The configuration of the RFID tag, like, for example, the sensor calibration should only be possible by "authorized" persons.
- It should not be possible to alter certain data stored on the RFID tag, for example, to change the last temperature measurement.

These are a few examples for problems that could arise by deploying unsecured Internet of Things (IoT)-enabled RFID tags. Therefore, it is vital to take security considerations into account when designing IoT-enabled objects.

Furthermore, passive RFID tags are resource constrained in various ways [7]:

Energy: Since the power for passive RFID tags is gained from the electromagnetic field dissipated by the reader, the available power is very limited.

Area: In order to make the utilization of Internet of Things-enabled RFID tags in real-life scenarios feasible, the cost of such tags is a limiting factor for their size (i.e., the size of cryptographic hardware, memory, etc).

Timing: RFID tags have to respond to a request within a limited time frame. These are typically some few milliseconds in practice.

Hence, hardware components and protocols must be carefully selected.

1.2 Goals of this Thesis

In [7], Dominikus et al. introduced their concept of Mobile IPv6 enabled RFID tags. In this thesis, we want to go the last step towards realizing the Internet of Things, based on this concept: Integrating passive RFID devices into the Internet. This is accomplished by defining and implementing a protocol that allows accessing specific Internet of Things-enabled tags from an arbitrary client. Further, already in the design phase security aspects of the solution are considered. Sensitive data collected by Internet of Things-enabled tags must be protected against unauthorized access from users in the Internet. Thus, it is necessary to supply cryptography and network capabilities in order to provide Web-based services to RFID tags. The main goal of this thesis is to develop a solution which allows a secure end-to-end connection between RFID tags and clients over the Internet. In order to accomplish this, a secure Internet layer will be established upon the RFID-communication layer. In more detail, the IAIK UHF RFID tags will be used and enhanced in order to create Internet-enabled RFID tags. These enhancements include:

1. Implementation of a lightweight solution to provide the basic IPsec functionality to the tag. Higher-level protocols will be implemented in software to be executed on an ATmega128 microcontroller, which is part of the IAIK UHF DemoTag. This includes the implementation of custom RFID commands as well as the implementation of IKEv2 functionalities.

2. Implementation of a cryptographic hardware in HDL. The crypto core will provide Elliptic curve Diffie-Hellman (ECDH) and Advanced Encryption Standard (AES) functionality, which is required by IPsec [23] and IKEv2 [20]. The ECDH core must meet the requirements of the IPsec and IKEv2 standard while using as less resources as possible. The smallest NIST curve over a prime field, namely the curve P-192 over $\mathbb{F}_{p192}$, is used for that purpose.

1.3 Outline

In Chapter 2, we give an introduction into the Internet of Things and its enabling technologies. These include Radio Frequency Identification (RFID), an insight into cryptography, Internet Protocol Security (IPsec), the Internet Key Exchange protocol version 2 (IKEv2), and Mobile IPv6. Furthermore, we look into some related work.

In Chapter 3, we present the design and the development of a security layer based on IPsec for secure end-to-end communication between an RFID tag and a corresponding node. First, the IAIK DemoTag is introduced, followed by a description of the basic concept and the implementation of the IKEv2 exchanges, including necessary custom RFID commands.

The design exploration regarding the hardware implementation of the ECC core is presented in Chapter 4. First, the NIST reduction is introduced, followed by the exploration of possible hardware implementations and optimizations. Next, various hardware-multiplier designs are examined. The remainder of the chapter is comprised of the evaluation of diverse concepts for deployment of different hardware multipliers and reduction strategies in the datapath of the ECC core.

Using the knowledge gained by the design exploration, the final ECC core is designed in Chapter 5. Starting with the derivation of the final multiplier design, the architecture

of the datapath is described. Later in the chapter, the chosen algorithm for elliptic-curve point multiplication and its implementation are outlined. Furthermore, the results regarding area and cycle count as well as a comparison with related work are given.

Finally, we present the conclusions and give an outlook in Chapter 6.

Chapter 2

Internet of Things

This chapter is comprised of an introduction to the Internet of Things (IoT). First, a general description and definitions for the IoT is given. The later sections focus on some enabling technologies as well as basic concepts for the IoT, i.e., the Radio Frequency Identification (RFID) technology, followed by an introduction into cryptography including Elliptic Curve Cryptography (ECC) as well as a description of the Internet Key Exchange version 2 (IKEv2) protocol, which together with Internet Protocol Security (IPsec) is useful for establishing secure end-to-end connections between two entities in the Internet (of Things). The final section of this chapter shows some related work, i.e., the work on which this thesis is based on.

2.1 What is the Internet of Things?

The basic idea behind the Internet of Things is that everyday objects interact with each other over the Internet. In [3], Atzori et al. point out that currently there are many definitions of the Internet of Things and for that reason it can be difficult to understand what IoT really means. Informations provided in this section are taken from their survey.

The name Internet of Things itself offers three perspectives to look at it. First, an *Internet* oriented approach, second, a *Things* oriented approach, and third, a *Semantic* oriented approach. The latter is based on the idea that in the future a huge number of items will be involved in the Internet, which leads to challenges regarding how information is stored, represented, generated and interconnected. A certain definition belonging to the *Things*-oriented approach considers the connection of RFID tags to the Internet. Another definition concentrates on anyone and anything being connected at any time. A definition falling in the *Internet*-oriented category states that the IP stack is a light protocol which is already used to connect many devices. Those devices sometimes are tiny, embedded devices supplied by a battery. Therefore IP already has the quality to realize the Internet of Things.

2.1.1 Enabling Technologies

The realization of the Internet of Things has only become possible due to its enabling technologies. Some of these are listed here:

- Regarding hardware, Radio Frequency Identification (RFID) systems are a major contributing factor to the realization of the Internet of Things, since RFID tags can

be produced small and cheap. Further, most RFID tags are passive, which means they do not come with their own dedicated power supply but are supplied by energy harvesting technologies. Passive RFID tags can possibly be made smaller, which makes it easier to attach them to arbitrary items.

- As mentioned in [7] and [6], the introduction of IPv6 was very important for the realization of the Internet of Things, because of its large address space. With IPv4, there would be the problem of not having enough addresses for the many individual objects (e.g., RFID tags) over the Internet.
- By enabling *things* to communicate with each other, possibly without any interference by humans, the risk of surveillance, information theft and other privacy, security and trust related hazards grows strongly. Therefore, the availability of mechanisms for protection of data is an enabling factor for the Internet of Things.

2.1.2 Applications for the Internet of Things

Realizing the Internet of Things leads to a huge number of applications with the possibility of improving the quality of our lives. Some potential domains as of [3] are healthcare, logistics, or the social domain. Some use cases for applications of the Internet of Things were already provided in Chapter 1. Here, without going into detail, some scenarios taken from [3] are listed:

- Logistics: Assisted driving, mobile ticketing, (realtime) monitoring
- Healthcare: Tracking, sensing, identification
- Social: Networking, thefts, losses

These should just give a hint on the variety of possible applications that could benefit from the Internet of Things.

2.2 Radio Frequency Identification

Radio Frequency Identification (RFID) enables contactless identification of so-called RFID tags by an RFID reader using a radio link. An RFID tag is a small transponder, consisting of a microchip and an antenna. Most RFID tags are *passive* which means they use energy harvesting concepts in order to supply the microchip, but there also exist RFID tags with dedicated power supplies like batteries. Such RFID tags are referred to as *active*. An RFID reader is a device which emits an electromagnetic field for communication with RFID tags. Passive RFID tags are also power supplied by the field of the reader. Many different RFID systems and standards exist, with different construction formats, prices, operating ranges and functionalities.

As of [9], the price class of the chip is primarily determined by the size of the chip. So, low-cost RFID tags are constrained in memory and processing power, both of which consumes chip area. Another limiting factor is the power supply. In [8], Feldhofer et al. state that the mean current consumption is limited to $15\,\mu A$, for a higher current draw limits the operation range of the tag. They further state that suitability of cryptographic algorithms is also determined by the number of clock cycles the algorithm needs for completion of a computation.

In [18], Hutter et al. showed that RFID devices are prone to side-channel attacks like differential power analysis, thus necessitating countermeasures when implementing cryptographic algorithms both in hardware and in software. As of [28], such countermeasures include the strategies *hiding* and/or *masking*. Hiding deals with the removal of data dependencies in the power trace. A strategy for hiding would be for all operations to consume the same amount of power. Masking deals with randomization of intermediate values in cryptographic protocols.

2.3 Cryptography: A Brief Introduction

In [24], Menezes et al. state that a fundamental goal of cryptography is to address the four security objectives *confidentiality*, *data integrity*, *authentication*, and *non-repudiation*. A short description of these services is given as follows:

Data integrity deals with (unauthorized) alteration of data. Data manipulation including deletion, substitution, and insertion must be able to be detected in order to assure data integrity.

Confidentiality, secrecy, privacy are all synonymous, describing the same objective of only letting authorized entities access content of information.

Authentication can be split into two security objectives, namely *entity authentication* and *data origin authentication*. *Entity authentication*, also called *identification*, is a technique for verifying identities, whereas *data origin authentication* is a technique for verifying the identity of the originator of a message.

Non-repudiation deals with the ability to prevent an entity from denying previous actions and/or commitments.

There exist a number of *cryptographic primitives*, which are tools that can be used to address these four security objectives. In this section just a brief insight into the primitives *symmetric cryptography* and *asymmetric or public key cryptography* is provided, followed by an overview of the *Advanced Encryption Standard (AES)* which belongs to the field of symmetric cryptography as well as an introduction into *Elliptic Curve Cryptography (ECC)*, which belongs to the field of asymmetric cryptography.

2.3.1 Kerckhoffs's Principle

In 1883, Kerckhoff stated a list of requirements for cipher systems [24]. The second point on this list declares that a the security of a cipher system must not depend on obscuring the details of the system but only the secrecy of the key.

2.3.2 Symmetric Cryptography

A symmetric encryption scheme is symmetric with respect to the key. That is, the encryption key k_e which is used to encrypt a message m, and the decryption key k_d, which is used to decrypt a message that was encrypted using the encryption key k_e are the same ($k_e = k_d$). In [25], symmetric key cryptography is described using the following analogy: There are two parties, namely Alice and Bob, as well as a safe with a strong lock. Alice

and Bob both have a key to the safe. Putting a message into it and locking the safe is analogous to the encryption, and opening the safe to again retrieve the message can be seen as decryption. For this analogy, the keys of Alice and Bob must be the same. A more mathematical definition of symmetric cryptography as seen in [24] would be: Let $E_e : e \in \mathcal{K}$ and $D_d : d \in \mathcal{K}$ be the sets of encryption respectively decryption functions of an encryption scheme. E_e denotes the encryption function with e being an encryption key taken from the keyspace $\mathcal{K}$, and D_d denotes the decryption function with d being the decryption key taken from $\mathcal{K}$. An encryption scheme is then called *symmetric-key encryption* if it is easy to determine e when only knowing d and vice versa, for all encryption/decryption key pairs (e, d).

As of [25], there are some deficiencies associated with asymmetric cryptography. First, there is the problem of **key distribution**. If Alice and Bob have to use the same key in order to decrypt a message encrypted by the other, the key has to be somehow distributed over a secure channel. Second, there is the potential that a **large number of keys** has to be dealt with, since each pair of users needs distinct keys. If n users want to communicate with each other in secrecy, there are $\frac{n*(n-1)}{2}$ individual key pairs, thus, each of the n users must store $n-1$ keys. Further, the keys have to be stored in a secure way. Third, since Alice and Bob use the same key, there is **no protection against cheating** by the other party. Of course there are strategies to deal with these shortcomings or at least mitigate them, see for example [24] and [25].

2.3.3 Asymmetric Cryptography

Asymmetric cryptography exploits the idea that for secure communication between Alice and Bob it is not necessary that the encryption key is being kept secret, but that the message can only be decrypted again by use of a decryption key which is secret [25]. In [24], public key decryption is defined as follows: Let $E_e : e \in \mathcal{K}$ and $D_d : d \in \mathcal{K}$ be the sets of encryption and decryption functions of an encryption scheme, respectively. E_e denotes the encryption function with e being an encryption key taken from the keyspace $\mathcal{K}$, and D_d denotes the decryption function with d being the decryption key taken from $\mathcal{K}$. An encryption scheme is then called *asymmetric-key encryption* if one of the encryption/decryption keys (e, d) is made publicly available and the other key is kept secret. The encryption key e, which is publicly available is then called *public key* and the secret decryption key d is called *private key*. The security of such an encryption scheme is based on the computational infeasibility of computing the private key d from the public key e. In order to realize asymmetric cryptography, the principle of one-way functions can be used. In [25] a one-way function is defined as follows: Let $f()$ be a function. If the computation $y = f(x)$ is "easy", and if the computation $x = f^{-1}(y)$ is infeasible, the function $f()$ is called a one-way function. Today, there are two such one-way functions used frequently in practical asymmetric cryptography, namely the *integer factorization problem* and the *discrete logarithm problem*. Elliptic Curve Cryptography is based on the latter.

2.3.4 Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) is a symmetric block cipher based on the Rijndael block cipher [10]. The only difference between the original Rijndael algorithm and the AES is that the block size for AES is always 128 bits, whereas the Rijndael block cipher allowed for the block sizes 128, 192, and 156 bits. Key lengths for both Rijndael and AES can either be 128, 192, or 256 bits. The AES cipher is computed in rounds,

where the number of rounds n_r is dependent on the chosen key length. All rounds except for the last one are identical. For 128-bit keys $n_r = 10$, for 192-bit keys $n_r = 12$ and for 256-bit keys $n_r = 14$.

Internally, AES operates on a two-dimensional array consisting of $4 * 4$ bytes, called the state s. Initially, the message block to be encrypted (respectively decrypted) is copied into s. If the input data in is represented as a linear array, the copy operation is according to the following scheme (see [10]):

$$s[r, c] = in[r + 4c], \tag{2.1}$$

where r denotes the row index of s and c denotes the column index of s, thus $0 \leq r < 4$ and $0 \leq c < 4$. At the end of encryption (or decryption), s contains the encrypted (respectively plain text) message block and is copied to the linear output array *out* as follows:

$$out[r + 4c] = s[r, c]. \tag{2.2}$$

A basic pseudo code representation of the cipher is shown in Algorithm 1. It uses five transformations, namely *SubBytes()*, *ShiftRows()*, *MixColumns()*, and *AddRoundKey()*. Based on [10], a short description of the individual transformations is given as follows:

SubBytes() is a non-linear byte-wise substitution of the individual bytes in s using a substitution table called *S-box*. The S-box can either be stored in some form, or in order to save memory, be calculated dynamically. For details on how the S-box is created please refer to [10].

ShiftRows() applies a cyclical byte-wise left shift (rotation) to the last three rows of the state s. The first row is left unchanged, the second row is rotated by one byte, the third row is rotated by two bytes and the fourth row is rotated by three bytes.

MixColumns() threats the state s column-wise and applies the following matrix multiplication to each column:

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} * \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} = \begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} \tag{2.3}$$

AddRoundKey() performs a bitwise XOR operation of each column of the state s with the *round key*. The *round key* is derived from the cipher key K by an initial key expansion. For details on how the key expansion is calculated, we refer to [10].

Algorithm 1 The basic AES cipher schedule

Require: $in, roundkeys$
Ensure: $s = enc(in)$
1: $s \leftarrow in$;
2: $AddRoundKey(s, roundkey)$;
3: **for** $round = 1$ to $n_r - 1$ **do**
4: $SubBytes(s)$;
5: $ShiftRows(s)$;
6: $MixColumns(s)$;
7: $AddRoundKey(s, roundkey)$;
8: **end for**
{last round:}
9: $SubBytes(s)$;
10: $ShiftRows(s)$;
11: $AddRoundKey(s, roundkey)$;
12: **return** s;

2.3.5 Elliptic Curve Cryptography (ECC)

Elliptic Curve Cryptography (ECC) belongs to the family of asymmetric cryptography algorithms. It's security is based on the discrete logarithm problem. Compared to RSA, which is another public key algorithm, ECC provides the same level of security while using keys of considerably smaller sizes.

Elliptic Curves

The so-called *Weierstrass equation*

$$E : y^2 + a_1 * x + a_3 * y = x^3 + a_2x^2 + a_4x + a_6, \tag{2.4}$$

where $a_1, a_2, a_3, a_4, a_6 \in K$ and the discriminant of E, $\Delta \neq 0$ defines an elliptic curve E over a field K. Hankerson et al. [15] further state, that the condition $\Delta \neq 0$ ensures that no points exist where the curve has $\geq$ two distinct tangent lines. A *Weierstrass equation*, as shown in Equation2.4 can be simplified to

$$E : y^2 = x^3 + ax + b \tag{2.5}$$

given that the characteristic of K is not equal to 2 or 3. The discriminant of the curve then is $\Delta = -16(4a^3 + 27b^2)$.

Point Addition and Doubling

Algebraic formulas for the addition of two points P and Q and for doubling of the point P, where P and Q are points on the elliptic curve E over the field K are given below. The formulas and definitions are taken from [15] and are valid for the simplified Weierstrass curve $E : y^2 = x^3 + ax + b$ where the characteristic of K is neither 2 nor 3. The point at infinity, denoted ∞, serves as *identity*.

1. *Identity:* $P + \infty = \infty + P \ \forall \ P \in E(K)$.

2. *Negatives:* If $P = (x, y) \in E(K)$, then $(x, y) + (x, -y) = \infty$. The point $(x, -y)$ which also is a point in $E(K)$ is called the *negative* of P and is denoted by $-P$. The point at infinity, $\infty = -\infty$.

3. *Point addition:* Let $P = (x_1, y_1) \in E(K)$ and $Q = (x_2, y_2) \in E(K)$, where $P \pm Q$. Then $P + Q = (x_3, y_3)$, where

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \text{ and } y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 (x_1 - x_3) - y_1. \tag{2.6}$$

4. *Point doubling:* Let $P = (x_1, y_1) \in E(K)$, where $P \pm -P$. Then $2P = (x_3, y_3)$, where

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1 \text{ and } y_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 (x_1 - x_3) - y_1. \tag{2.7}$$

Point Representation

By observation of the equations for point addition and point doubling given in Equation 2.6 respectively Equation 2.7, it can be seen that a field inversion is required for each addition and doubling operation. Field inversion can be significantly more expensive than field multiplication. Thus, using alternative point representations can be advantageous, i.e., by transforming affine coordinates into *projective* coordinates. As stated in [29], a point $P = (x, y)$ on an elliptic curve E can be represented by the triplet (X, Y, Z), where $(X/Z^c, Y/Z^d) = (x, y)$ for some positive integers c and d. This representation is called *projective* representation. There are as many projective representations of a point P as there are different Z. The projective coordinates where $c = 2$ and $d = 3$ are called *Jacobian coordinates*.

By using *Jacobian coordinates*, the projective form of the simplified Weierstrass curve is $Y^2 = X^3 + aXZ^4 + bZ^6$. Possible formulae for computing a point doubling in *Jacobian coordinates* are (see [15]):

$$X_3 = (3X_1^2 + a * Z_1^4) - 8X_1Y_1^2 \tag{2.8}$$

$$Y_3 = (3X_1^2 + a * Z_1^4)(4X_1Y_1^2 - X_3) - 8Y_1^4 \tag{2.9}$$

$$Z_3 = 2Y_1Z_1 \tag{2.10}$$

Formulae for computing a point addition in *Jacobian coordinates* taken from [29] are:

$$X_3 = F^2 - E^3 - 2BE^2 \tag{2.11}$$

$$Y_3 = F(BE^2 - X_3) - DE^3 \tag{2.12}$$

$$Z_3 = Y_1Z_1 \tag{2.13}$$

where $A = X_1Z_2^2$, $B = X_2Z_1^2$, $C = Y_1Z_2^3$, $D = Y_2Z_1^3$, $E = A - B$, and $F = C - D$. There is also the possibility to perform the point addition $P + Q$ with *mixed Jacobian-affine coordinates*, where P is represented in *Jacobian coordinates* and Q is represented in affine coordinates ($Z = 1$), which can speed up the computation [29].

Domain Parameters

As stated in [15], an elliptic curve E can be described by a set of so-called *domain parameters* $D = (q, FR, S, a, b, P, n, h)$, where q is the field order, FR is the representation used for the elements of $\mathbb{F}_q$, $a, b \in \mathbb{F}_q$ are the coefficients as seen in 2.5, $P \in E(\mathbb{F}_q)$ denotes the base point of the curve, n its order and h is the *cofactor*.

Key Pair Generation

Algorithm 2 (taken from [15]) depicts the generation of an elliptic curve key pair. It is also part of the Elliptic-Curve Diffie-Hellman algorithm. Prior to the key pair generation, a set of domain parameters D must be determined. The algorithm then randomly selects a private key d and computes the public key Q by the elliptic curve point multiplication $Q = d * P$. The key pair then is (Q, d).

Algorithm 2 Elliptic curve key pair generation

Require: Domain parameters $D = (q, FR, S, a, b, P, n, h)$
Ensure: Puplic key Q, private key d
1: select $d \in_r [1, n-1]$;
2: compute $Q = d * P$;
3: **return** (Q, d);

Elliptic Curve Diffie-Hellman (ECDH)

Elliptic Curve Diffie-Hellman (ECDH) is a key agreement protocol using elliptic curve cryptography. It is based on the basic Diffie-Hellman key agreement which is a practical solution for key distribution. The basic Diffie-Hellman key agreement in [24] is described as follows:

1. Select a prime p and a generator g of $\mathbb{Z}_p^* (2 \leq g \leq p-2)$
2. Alice chooses a random secret x where $1 \leq x \leq p-2$, computes $A = g^x \mod p$ and transmits A to Bob.
3. Bob chooses a random secret y where $1 \leq y \leq p-2$, computes $B = g^y \mod p$ and transmits B to Alice.
4. Bob receives g^x and computes the shared key $K = (g^x)^y \mod p$.
5. Alice receives g^y and computes the shared key $K = (g^y)^x \mod p$.

As stated in [24], the Diffie-Hellman protocol can be carried out in any group with efficient exponentiation and and where the discrete logarithm problem on which the security of the algorithm is based is hard, e.g., the group of points defined by an elliptic curve over a finite field. Thus, ECDH can be described as follows:

1. Select the elliptic curve domain parameters $D = (q, FR, S, a, b, P, n, h)$
2. Alice uses Algorithm 2 to generate the private key d_A and the public key Q_A and transmits the public key to Bob.
3. Bob uses Algorithm 2 to generate the private key d_B and the public key Q_B and transmits the public key to Alice.
4. Bob receives Q_A and computes the elliptic curve point multiplication $(x_k, y_k) = d_B * Q_A$.
5. Alice receives Q_B and computes the elliptic curve point multiplication $(x_k, y_k) = d_A * Q_B$.

The shared secret then is the x-coordinate x_k of the point.

2.4 Internet Protocol Security (IPsec)

The goal of Internet Protocol Security (IPsec) is to provide security for the Internet Protocol (i.e., IPv4 and IPv6). IPsec offers a set of services and protocols for integrity, data-origin authentication, confidentiality, access control and others. It is defined in RFC 4301 (see [23]). IPsec is mandatory in IPv6. There are three major protocols involved with IPsec. The Internet Key Exchange protocol (see Figure2.5) is used to exchange keys in a secure way. The two traffic security protocols, namely Authentication Header (AH), defined in RFC 4302 [21], and Encapsulating Security Payload (ESP), defined in RFC 4303 [22] provide security services. ESP and AH are briefly introduced below.

2.4.1 Authentication Header

The authentication Header protocol provides data origin authentication, integrity, access control, and optional anti-replay features, but does not provide confidentiality. The cryptographic algorithms used for these security services are defined by the Security Association negotiated via IKEv2. Integrity is provided using an integrity check value generated by the algorithm specified in the Security Association. Further, a sequence number is included in each AH packet which is used for the (optional) anti-replay features.

2.4.2 Encapsulating Security Payload

The Encapsulating Security Payload provides data origin authentication, integrity, access control, and optional anti-replay features. As opposed to AH, ESP also provides confidentiality. The ESP packet contains a sequence number and an integrity check value, for anti-replay features and integrity. In addition, the whole payload sent with ESP is encrypted with the encryption algorithm specified in the negotiated SA.

2.4.3 Security Associations

Security Associations (SA) are a fundamental concept of IPsec. IPsec can either work in *tunnel mode* or in *transport mode*, for this purpose two separate SAs are defined. As described in Section 2.5, the SAs for IPsec are constructed by the IKEv2 protocol. In [23], the SA is defined as a simplex connection (thus two SAs are required in order to secure a bi-directional connection) with security services provided to it. A SA specifies attributes like which cryptographic algorithm should be used for a particular purpose (e.g., authentication, encryption, etc.).

Transport Mode

Transport Mode offers end-to-end protection of the traffic between two hosts which implement the IPsec protocol. It can only be used for host-to-host communication. The source and destination IP addresses are not changed or encrypted.

Tunnel Mode

In Tunnel Mode the entire IP packet is authenticated and (in case of ESP) encrypted. An additional IP datagram encapsulates the original packet. Tunnel Mode can be used for host-to-host, network-to-network, and host-to-network connections.

Table 2.1: Abbreviations of the different payload types which appear in the IKEv2 exchanges (taken from [20]).

AUTH	Authentication
CERT	Certificate
CERTREQ	Certificate Request
CP	Configuration
D	Delete
EAP	Extensible Authentication
HDR	IKE header (not a payload)
IDi	Identification - Initiator
IDr	Identification - Responder
KE	Key Exchange
Ni, Nr	Nonce
N	Notify
SA	Security Association
SK	Encrypted and Authenticated
SPIi	Security Parameter Index - Initiator
SPIr	Security Parameter Index - Responder
TSi	Traffic Selector - Initiator
TSr	Traffic Selector - Responder
V	Vendor ID

2.5 Internet Key Exchange Version 2 (IKEv2)

The Internet Key Exchange Version 2 (IKEv2) protocol was first defined in the (now obsolete) RFC 4306 (see [19]) and is an update to the Internet Key Exchange (IKE) protocol. The current version of IKEv2 is given in RFC 5996 [20] and the update RFC 5998[31]. IKEv2 is a component of IPsec, dealing with the key exchange, performing mutual authentication between two parties, as well as establishing Security Associations (SAs). IKE SAs include secrets that can be used for establishing IPsec SAs (for *Encapsulating Security Payloads* (ESP) or *Authentication Header* (AH)). The term *Child SA* refers to a security association which was set up for ESP or AH through IKE.

IKE communication is based on a request/response type protocol. A pair of messages, namely a request message and its response message are called *exchange* for the remainder of this section. The descriptions given in this section are based on RFC 5996 [20].

2.5.1 Exchanges

There are four individual exchanges defined in IKEv2, namely IKE_SA_INIT, IKE_SA_AUTH, CREATE_CHILD_SA, and INFORMATIONAL exchanges. Usually, there is a single IKE_SA_INIT exchange followed by a single IKE_SA_AUTH exchange used for establishing an IKE SA and a first Child SA. There can be any number of CREATE_CHILD_SA and INFORMATIONAL exchanges. This section gives a short description of the IKEv2 exchanges, without claiming to be exhaustive. Figure 2.1 depicts the exchanges between two peers Alice and Bob, including the payloads as described in [20]. The meaning of the abbreviations is shown in Table 2.1. Optional payloads are denoted as [...].

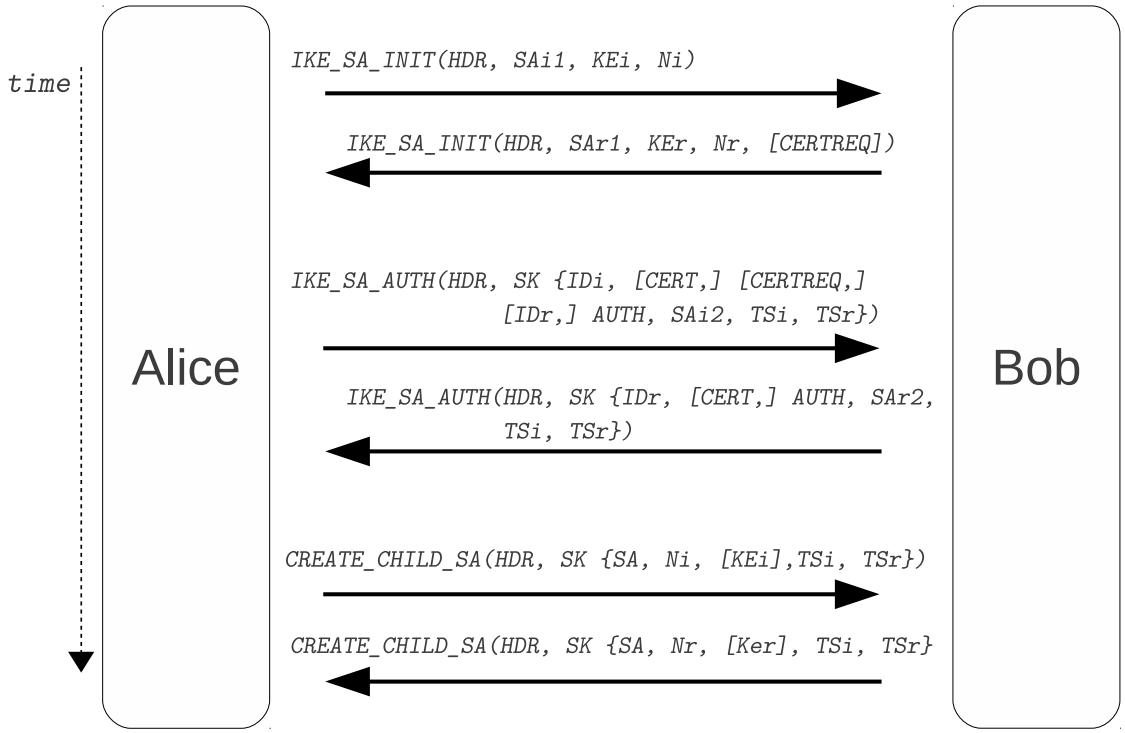

Figure 2.1: IKEv2 exchanges between the two peers Alice and Bob

IKE_SA_INIT Exchange

IKE_SA_INIT is the first exchange of an IKE session, used to negotiate the security parameters (like cryptographic suites to be used for the IKE SA). It also performs a Diffie-Hellman key agreement as described in 2.3.5. The exchange including the payloads can be seen in Figure 2.1. The header HDR contains the SPIi, which is a number chosen by the initiator to identify the SA. SAi1 indicates a set of cryptographic algorithms that are supported by the initiator for use in the IKE SA. The Diffie-Hellman values of the initiator are provided within the KEi payload. Further, a nonce Ni is provided. In IKEv2, nonces are used, e.g., for the calculation of SKEYSEED.

After receiving the init message, the responder chooses cryptographic algorithms from those provided by the initiator in the SAi1 payload and creates the SAr1 payload containing the choices. The responder can request a certificate using the CERTREQ payload (which is optional). The Diffie-Hellman key exchange is completed by the inclusion of the KEr payload in the init response.

After the IKE_SA_INIT exchange, both peers can calculate SKEYSEED from the exchanged payloads, which is used to derive all keys for integrity protection and encryption of future messages (see Section 2.5.3).

Remark: Since the IKE_SA_INIT request already contains the DH payload with the initiators Diffie-Hellman values, but the Diffie-Hellman group to be used has not been negotiated yet, the initiator has to guess which group the responder will select from the initiators proposed groups. If the guess was wrong, the responder will use a Notify payload to inform the initiator.

IKE_SA_AUTH Exchange

The IKE_SA_AUTH exchange is used to authenticate the messages previously sent by the IKE_SA_INIT exchange. Except for the header, the IKE_SA_AUTH exchange messages are already encrypted and integrity protected using keys established through the initial exchange. The encrypted payload SK{...} contains the IDi to assert the identity. The AUTH payload is used to authenticate the IDi as well as for integrity protection of the message. The traffic selectors TSi and TSr are used to define which traffic should be protected by the Child SA. The responder is allowed to choose a subset of the initiators proposed traffic. SAi2 is already part of the Child SA negotiation. The responder also asserts its identity (IDr), uses AUTH to authenticate the identity and integrity protect the message, and completes the Child SA negotiation.

CREATE_CHILD_SA Exchange

The CREATE_CHILD_SA exchange is used to establish a security association for IPsec by using the algorithms negotiated by SAi2 respectively SAr2 during the IKE_SA_AUTH exchange. Again, except for the header, the whole message is encrypted. Within the encrypted payload, a SA offer is provided by the initiator, as well as a new nonce, new traffic selectors, and possibly a new Diffie-Hellman value. The response message is created similar, with the chosen algorithms in the SA response, a nonce, traffic selectors which possibly only select a subset of those proposed and the optional Diffie-Hellman value. The CREATE_CHILD_SA exchange can utilize a Notify payload in order to use IPsec in transport mode for the negotiated SA. Per default, tunnel mode is used.

INFORMATIONAL Exchanges

INFORMATIONAL exchanges are used in order to transmit control messages as well as for error handling and notification of certain events. The payloads of INFORMATIONAL exchanges are encrypted with the negotiated keys. Therefore they must only occur after the IKE_SA_INIT and IKE_SA_AUTH exchanges. Since with IKEv2, all exchanges consist of a request/response pair and must be completed before another exchange can take place, an empty INFORMATIONAL request triggers an INFORMATIONAL response nonetheless and thus can be used for checking the liveness of the peer.

2.5.2 Payloads

IKEv2 payloads do not have to be sent in the same order as seen in Figure 2.1. Each payload contains the so-called *Generic Payload Header* which (amongst others) indicates the next payload type in the message. This section contains a non-exhaustive description of some payloads that are particularly interesting for the purpose of this work.

Nonces

The initiator's nonce used in IKEv2 must be at least as long as half the key size of **all** of the proposed pseudo-random functions. The responder's nonce must be at least half as long as the key size of the chosen pseudo-random function.

The Authentication Payload

The authentication payload AUTH includes the authentication data of variable length. It is part of the IKE_SA_AUTH message exchange and authenticates the peers, e.g., by signing a block of data. The initiator signs the IKE_SA_INIT request. The signed octets start with the SPI in the HDR. Appended to this are the nonce of the responder and the result of the pseudo-random function with SK_pi used as the key K and IDi' used as the seed. IDi' denotes the ID payload excluding the header. The responder signs the IKE_SA_INIT response. The signed octets again start with the first SPI in the HDR. Appended to this are the nonce of the initiator and the result of the pseudo-random function with SK_pr used as the key K and IDr' used as the seed. IDr' denotes the ID payload excluding the header. In case pre-shared keys are used, the AUTH value is then computed as:

$$AUTH = prf(prf(\text{shared secret, “Key Pad for IKEv2”}), < signed\_octets >)$$

where the string “Key Pad for IKEv2” consists of 17 ASCII characters.

The Encrypted Payload

The encrypted payload SK{...} is part of all exchanges except for the IKE_SA_INIT exchange. It contains all other payloads of a message (except for the header HDR) in encrypted form and must be the last payload of a message. Besides the generic payload header, SK{...} is comprised of the following payloads:

- An initialization vector IV for the encryption algorithm. The length of IV depends on the block size of the encryption algorithm for cipher block chaining mode ciphers.
- The encrypted IKE payload, which consists of the encrypted concatenation of IKE payloads (e.g., nonces, SAs, etc.).
- Padding of length 0 to 255 byte. The length of the padding must be such that the combination of the encrypted IKE payloads, the padding itself and the pad length are a multiple of the encryption block size.
- The pad length field, which indicates the length of the padding
- Integrity checksum data, which is the cryptographic checksum of the entire message, including the HDR and the encrypted payload.

2.5.3 Generating Key Material

Four cryptographic algorithms are necessary for IKEv2 SA negotiation. These are the encryption algorithm, the integrity protection algorithm, the Diffie-Hellman group, as well as a pseudo-random function (denoted PRF). Naturally, these algorithms need cryptographic keys. The PRF is utilized for generation of keying material. A list of IKEv2 parameters including supported algorithms is provided by the Internet Assigned Numbers Authority (IANA) (see [2]).

PRF+

The negotiated PRF is used to generate **all** keying material. Since the bit length of its output is dependent on the chosen PRF and might not be long enough to provide the whole amount of keying material, the PRF is used in an iterative fashion: Let

$$\begin{aligned} T1 &= prf(K, S|0x01) \\ T2 &= prf(K, T1|S|0x02) \\ T3 &= prf(K, T2|S|0x03) \\ T4 &= prf(K, T3|S|0x04) \\ &... \end{aligned}$$

where | denotes concatenation. K is the key and S is the seed. The PRF+ is then defined as:

$$\text{prf+}(K, S) = T1|T2|T3|T4|...$$

The output of the PRF+ is a bitstream from which all keys are taken consecutively.

SKEYSEED

Several keys have to be generated for the IKE SA. These are:

- SK_d is used for deriving keys for the Child SA
- SK_ai is the initiators key for integrity protection
- SK_ar is the responders key for integrity protection
- SK_ei and SK_di are used for encryption respectively decryption of subsequent messages
- SK_pi and SK_pr are needed for generating the AUTH payload

For the generation of these keys, first SKEYSEED is computed by utilization of the PRF with the initiator's nonce Ni concatenated with the responders nonce Nr as the key[1] and the shared secret of the Diffie-Hellman exchange (denoted g^{ir}) as the seed:

$$SKEYSEED = prf(Ni|Nr, g^{ir})$$

SKEYSEED is then used as the key K for the PRF+. The seed is $S = Ni|Nr|SPIi|SPIr$. The final keying material thus is created by the PRF+ as follows:

$$\begin{aligned} &\{SK\_d|SK\_ai|SK\_ar|SK\_ei|SK\_er|SK\_pi|SK\_pr\} \\ &= \text{prf+}(SKEYSEED, Ni|Nr|SPIi|SPIr) \end{aligned}$$

[1] For some pseudo-random functions only the first 64 bits of Ni respectively Nr are concatenated and used as the key.

KEYMAT

Keying material for Child SAs is created differently from the method described above. Instead of SKEYSEED, the key SK_d (which was previously derived by the use of SKEYSEED and the PRF+) is used as the key K of PRF+. Keying material for CREATE_CHILD_SA exchanges which include the optional Diffie-Hellman exchange is created as follows:

$$KEYMAT = \text{prf+}(SK_d, g^{ir}(new)|Ni|Nr)$$

Keying material for CREATE_CHILD_SA exchanges without the optional Diffie-Hellman exchange is created as follows:

$$KEYMAT = \text{prf+}(SK_d, Ni|Nr)$$

2.6 Mobile IPv6

IPv6 is specified in RFC 2460 [5]. It is the successor to Internet Protocol version 4 (IPv4). The main reason for deploying IPv6 was to expand the addressing capabilities. IPv6 has an address space of 128 bits, as opposed to the address space of 32 bits of IPv4. Further changes include a "simpler" header format (some header fields have been made optional), data integrity and confidentiality capabilities, as well as support for greater flexibility.

Mobile IPv6 is specified in RFC 6275 [27] and provides a way for ensuring that mobile nodes can remain reachable while moving[2]. In Mobile IPv6, *home address* refers to the IP address of a node, where the network part of the IP address points to the network containing the home link. The *care-of address* refers to the IP address of a node, where the subnet prefix of the IP address points to the network containing the foreign link where the node currently is located. *Home Agent* refers to a Mobile IPv6 capable router on the home link of the mobile node. When the mobile node is connected to a foreign network, the Home Agent handles forwarding of packets that are addressed to the home address of the mobile node to the current care-of address. When a mobile node moves to a foreign network it has to generate a new care-of address and transmit it to the Home Agent which in terms updates its routing table.

2.7 Related Work

2.7.1 On Passive RFID Technology for the Internet of Things

In [6], Dominikus et al. showed a way of modifying RFID readers and low-cost tags for future Internet of Things use. In this section, a short description of some of their ideas is given.

Their concept for integration of passive RFID tags into the Internet is based on Mobile IPv6. The tags are treated as mobile nodes, each holding a unique IPv6 address but without actually implementing the IPv6 stack. Instead, the readers have to forward messages from a corresponding node to the tag or vice versa. The following enhancements were introduced for *Home Agents*, RFID readers, and RFID tags:

- *Home Agents* hold a database for managing their assigned RFID tags. When a tag is assigned, a database entry containing the unique identifier of the tag as well as

[2] In this context the term *movement* refers to a change of the link to the Internet.

its home address is created. Analogue to standard Mobile IPv6, the home address consists of a 64-bit subnet prefix of the home agent and a 64-bit unique identifier of the tag. When a tag enters the field of a reader, a care-of address is created, consisting of the reader's subnet prefix and the 64-bit unique identifier of the tag. The corresponding node stores this care-of address in its database. Hence, it can route messages which are sent to the tag's home address to the subnet where the tag is currently located and therefore the tag itself.

- The Internet of Things enabled RFID readers are connected to the Internet and communicate with the tags via the RFID protocol. They must be able to handle IPv6 packets and act as a router for the tags that are currently located in their field. Internet of Things enabled tags are recognized by the reader (e.g., via a flag that is transmitted to the reader) and their IPv6 address is obtained in order to create a new care-of address. The care-of address is then sent to the corresponding node. In order to retrieve the IPv6 address from the tag, custom RFID commands could be used.
- Internet of Things enabled RFID tags have to implement some dedicated custom RFID commands like *get_IP()* and *set_IP()* for retrieving and setting their IPv6 address by the reader. In addition, some memory (i.e., 128 bits) is required to store the IPv6 address. The tag itself does not implement the full IPv6 stack. Instead the reader is used as a "translator".

2.7.2 On Secure Communication with RFID Tags in the Internet

In [7], Dominikus and Kraxberger introduced a concept, where special RFID tags are enabled to communicate via IPv6. Since this thesis describes an approach for realizing their vision, this section gives a short recapitulation of their ideas and concepts as described in [7].

Preliminaries

The idea of integrating RFID tags into the Internet (of Things) by extending the tags with (Mobile) IPv6 capabilities is based on the observation that low-cost tags already provide basic functionalities like extended memory and additional useful commands like the kill command. It is therefore assumed that the features provided by RFID tags in the near future will become even more practical for the implementation of such an idea (e.g., more memory while still keeping the tags cheap).

The whole concept was only made possible by the introduction of IPv6, which provides an address space large enough to address many individual tags per square meter and therefore enable addressing an expected huge number of tagged items. Concepts like Mobile IPv6 are beneficial since it considers the moving of mobile nodes between subnets.

Secure communication over the Internet is an enabling technique for many applications. In order to become a trusted media, communication over the Internet of Things must be able to be secured. In the "traditional" Internet, IPsec is a well known way for securing IPv6 connections.

Passive RFID tags already provide cryptographic functions such as performing symmetric and asymmetric encryption which can be utilized for securing the communication between an Internet of Things-enabled RFID tag and a corresponding node somewhere in the Internet.

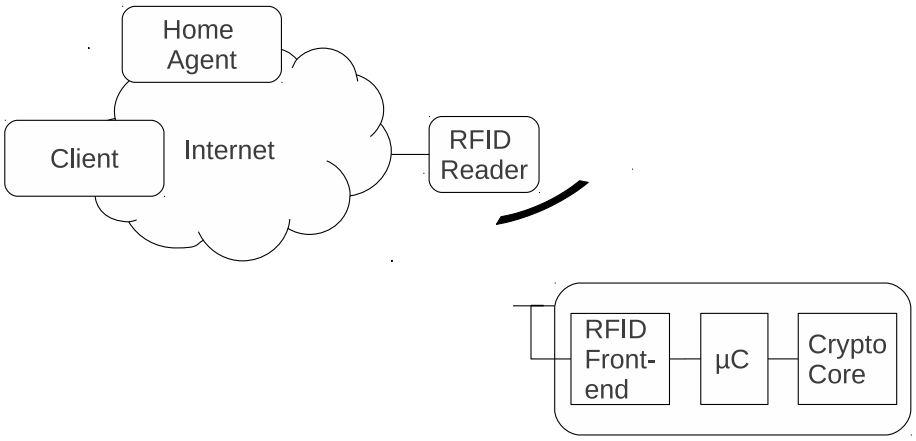

Figure 2.2: System overview

The Basic Concept for Secure End-to-End Communication

Four communicating parties are involved in the proposed system:

- Tagged items (i.e., Internet of Things enabled RFID tags attached to arbitrary items), that, in the context of Mobile IPv6 can be seen as mobile nodes
- RFID readers
- A tag manager very similar to the *Home Agent* in Mobile IPv6
- Corresponding nodes, which are arbitrary nodes in the Internet that communicate with the tags.

As depicted in Figure 2.2, RFID readers, corresponding nodes, and the tag manager are connected to the Internet, whereas the tagged items are assumed to be mobile and can possibly move between various RFID-reader fields. Because of the limited capabilities of RFID tags, they are not capable of doing IPv6 on their own. Thus, part of the protocol handling is transfered to the Reader. Implementation of the whole IPv6 stack as well as the whole functionality of IPsec on the RFID tag is not feasible due to memory requirements not meeting the constraints of the tag. By simply shifting the IPv6 functionality to the reader, a basic integration of the tags into the Internet of Things could be established. This could by accomplished by letting the reader "translate" the packages and forward their content to the tag via RFID commands, or vice versa, packing the content sent by the tag into IPv6 packages and sending them over the Internet. The Mobile IPv6 protocol could be used to take care of the addressing of the tag and handle moving of the tag through different subnets. However, for such an approach, the RFID tag would have to trust the reader which in general is not possible. Further, the approach would not allow for secure end-to-end connection between an RFID tag and a corresponding node since a possible secured connection via IPsec would only be possible between the corresponding node and the (not trusted) reader. The reader can be seen as a router for the tags. For these reasons, Dominikus et al. suggested to establish a dedicated security layer to make secure end-to-end communication between an RFID tag and a corresponding node possible over the Internet (of Things).

Their concept for this security layer is strongly based on IPsec, which provides the security services *authentication* and *confidentiality*, as well as key agreement for the session

keys. These services are also provided by their dedicated security layer. To make this achievable for resource-constrained RFID tags, some assumptions and constraints have to be made. In [7], the following constraints were proposed:

Certificates: For key agreement, public-key cryptography is used. The *Home Agent* is used as the *Certificate Authority* to sign the public key of the tag and stores it (the certificate) in a database together with the Care-Of and Home address of the respective RFID tag. The *Home Agent* is also responsible for deciding which corresponding node is permitted to talk to the respective tag.

IPv6 datagram size: Since IPv6 datagrams can have a size of 64 kB and more if so called Jumbo datagrams are used, and RFID tags are memory constrained, the maximum size of the data sent to the tag has to be constrained.

Limiting connections Only one active connection to a certain RFID tag is allowed at a time due to memory and computational power limitations of the tags. This means, only one *Corresponding Node* can talk to a certain tag at at time, other connections are refused.

Mode of operation: Dominikus et al. suggested to only use one mode of operation respectively protocol for the secure connection between a *Corresponding Node* and an RFID tag using IPsec, namely *Encapsulating Security Payload* in *Transport Mode*.

Key Agreement based on IKEv2

Dominikus et al. propose a slightly modified respectively restricted key agreement protocol between the tag and a corresponding node, strongly based on IKEv2. The restrictions and assumptions are necessary due to the resource constrained nature of RFID tags. For compatibility with the standard IKEv2 protocol, the protocol described here is basically a subset of standard IKEv2. As we see later on in Chapter 5.6, the constraints suggested in [7] which are described here unfortunately are not sufficient and further assumptions have to be made when using low-cost RFID tags currently available.

The key agreement is initiated by the *Corresponding Node* using the *IKE_SA_INIT* request. The *security association* parameter is not necessary because of the constraint of only one security association at a time is allowed due to the tag's limits. Via Mobile IPv6, the request is sent to the *Care-of* address of the tag (i.e., the particular RFID reader where the tag currently is located). The reader then extracts the required payloads and translates the request to a custom RFID command which is sent to the tag. The tag responds with a custom command, including the tag's Diffie-Hellman values and nonce values. The payloads are extracted by the reader and translated into the *IKE_SA_INIT* response which is then sent back to the corresponding node via Mobile IPv6.

After the *IKE_SA_INIT* exchange, both parties calculate the *SKEYSEED* value from which keys for further communications are derived. Dominikus et al. suggest that from the suit of possible algorithms, AES is chosen for encryption and HMAC-SHA-1 is chosen for authentication but also mention that it is possible to solely use AES for encryption, authentication and random number generation which is the approach used for this work. The following exchanges, namely *IKE_SA_AUTH* and *CREATE_CHILD_SA* are encrypted and authenticated using the keys derived from *SKEYSEED* and the algorithms chosen from the suit (see [20] for a list of possible cryptographic algorithms for IKEv2).

The *IKE_SA_AUTH* request is sent to the reader with its payloads encrypted and authenticated. The reader again translates the command to a custom RFID command in order to forward the contents of *IKE_SA_AUTH* to the tag. The tag sends back a response to the reader, containing the data for the *IKE_SA_AUTH* response, again encrypted and authenticated. The reader then assembles the *IKE_SA_AUTH* response message from the data sent by the tag and transmits it to the *Corresponding Node*.

Finally the *Corresponding Node* sends the *CREATE_CHILD_SA* request to the reader to establish the security association for IPsec. Again, the payloads are encrypted and authenticated. The reader transmits the contents of *CREATE_CHILD_SA* to the tag via another custom RFID command, which responds with its respective encrypted and authenticated *CREATE_CHILD_SA* payloads. These are again sent to the *Corresponding Node* via the *CREATE_CHILD_SA* response by the reader. The *CREATE_CHILD_SA* exchange contains new Diffie-Hellman values which are used by both parties to calculate new encryption and authentication keys which are then used for further communication.

At this point, the key agreement procedure is completed. The new keys derived from the Diffie-Hellman values transmitted by the *CREATE_CHILD_SA* exchange can now be used for secure communication between the RFID tag and the *Corresponding Node*.

Chapter 3

The IAIK UHF DemoTag in the IoT

The IAIK UHF DemoTag is an emulator for Ultra High Frequency (UHF) RFID tags that work according to the ISO18000-6C standard [1]. The DemoTag was used for development of a security layer based on IPsec that should enable secure end-to-end communication between a corresponding node and the RFID tag over the Internet (of Things). This chapter describes the underlying concept and implementation for the practical realization.

3.1 The DemoTag

The IAIK UHF DemoTag consists of an analog front-end connected to a dipole antenna (printed on the Printed Circuit Board (PCB)), an interface to a host computer, and daughter boards, which either provide the functionality of an RFID tag via a microcontroller or an FPGA. It operates at a carrier frequency of 868 MHz. The UHF DemoTag is power supplied by either a battery, or USB. For the purpose of developing the security layer, a daughter board with an ATmega128 microcontroller operating at 16 MHz CPU frequency was used. The DemoTag includes an ISO18000-6C example application which can handle the EPC Generation 2 standards. Based on this application, the security layer was implemented. The example application is written in C and partially in assembler code, however, the security layer was purely written in C, using the Rowley Associates CrossWorks development system.

3.2 IPsec-Protocol Design

While the design of the security layer is strongly based on the ideas of Dominikus et al. proposed in [7], some changes and enhancements have been applied. First of all, their concept for secure end-to-end communication between an RFID tag and a corresponding node over the Internet of Things does not go into the very details (which was not their intention anyways). Second, due to RFID tags being resource constrained, some assumptions have had to be made regarding the used protocols, in order for the implementation to become applicable in real systems.

The basic concept can be described as follows: Internet of Things-enabled RFID tags and readers implement some features to make a *secure* end-to-end connection between *any* corresponding node and a tag possible. For end-to-end security features, IPsec and

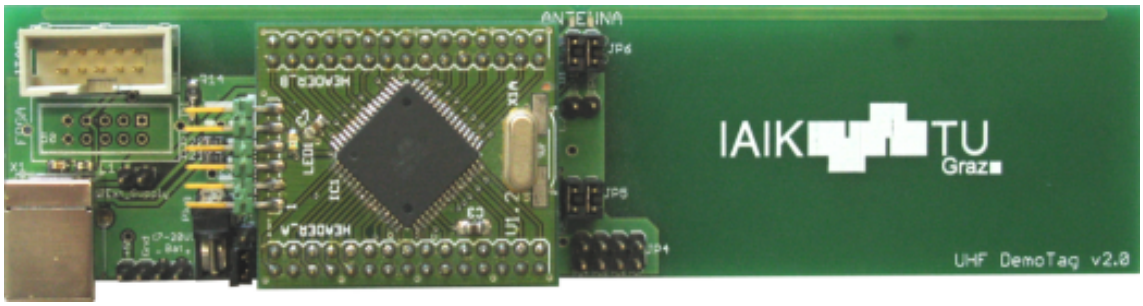

Figure 3.1: Overview of the IAIK UHF DemoTag

IKEv2 are used. When the IPsec security association has been set up, the protocol is relatively easy, so the real effort is the key exchange between a corresponding node and the tag via IKEv2 (cf. Section 2.7). Since an RFID reader can not be trusted in general, the connection must be protected between the corresponding node and the tag, protection between the node and the reader is not sufficient. Unfortunately, IPsec and IKEv2 are demanding protocols in terms of memory requirements, at least in the context of low-resource RFID systems. IPv6 packets alone can have a size of 64 kB (and more). Hence, the RFID reader must perform as much work as possible while security relevant parts have to be handled by the tag. Naturally, for the requirement that *any* corresponding node is able to communicate with IoT-enabled RFID tags in a secure way, it is important to stay within compliance with the IKEv2 respectively IPsec protocols. For this first prototype some restrictions had to be made. That is, the underlying protocols have been implemented in compliance with the specifications of IPsec and IKEv2. Only algorithms that are listed by the IANA (see [2]) for use with IPsec and IKEv2 are used. However, only a subset of the possible algorithms is available on the tag. IKEv2 defines a set of algorithms that *must* be available in all implementations (cf. RFC 4307 [30]), this requirement can not be met as of today. Further, due to severe memory limitations on the tag, some constraints had to be made regarding the various payloads. Some payloads in IKEv2 can be of variable length. Nonces, for example, can have a length of up to 256 bytes. Thus, payloads are restricted to their minimum length which is still in compliance with the protocol specification as for now. The RFID reader acts as a router for the tags and prepares IKEv2 and IPsec packets for the tag. Tags can be seen as Mobile IPv6 nodes, each having its own IPv6 address as well as some dedicated memory that purpose. For setting and retrieving the IP address by the reader, two custom commands as proposed in [6] and [7] were implemented. These are

- setIP(), which is used by the RFID reader, e.g., to change the the IP address of the tag when its owner changes, and
- getIP(), which is used by the RFID reader in order to retrieve the tag's IP address in order to generate a care-of address.

The whole complexity of the Mobile IPv6 protocol is shifted to the reader. The following sections provide a description of how the IKEv2 transforms were implemented, including how the workload is shared between the tag and the reader. For the purpose of these descriptions it is assumed that no Mobile IPv6 specific actions have to be performed, i.e. that the tag is in the field of a reader that is located in the tags home network.

3.2.1 The Init Exchange

If the reader receives an IKE_SA_INIT request for the tag from a corresponding node, it uses a custom command *IPsec_KE_InitCmd()* to transmit the required payloads to the tag. The tag then extracts the payloads and calculates a Diffie-Hellman keypair for transmission to the corresponding node. Elliptic Curve operations are handled by the Elliptic Curve Cryptography module which is described in Chapters 4 and 5. From the Diffie-Hellman public key received through the init request, the tag can also calculate the shared secret. Then SKEYSEED is calculated and the cryptographic keys are derived, using the pseudo-random function. A response payload is assembled and stored until the reader requests it by using another custom command *transmitInitPayloads()*. From these payloads the reader constructs the actual IKE_SA_INIT response and sends it to the corresponding node.

3.2.2 The Auth Exchange

The principle workflow of the auth exchange is the same as of the init exchange. Again, the reader gets a request, extracts the required payloads and transmits them to the tag, where they are processed and the response payload is assembled. The response payload are then requested by the reader and upon receiving them they are assembled to a real IKE_SA_AUTH response and transmitted to the corresponding node. However, all payloads of the auth exchange (except for the header) are encrypted, therefore the reader can not take off work from the tag.

In IPsec, the integrity checksum is calculated over the init message and some concatenated data (cf. 2.5.2). For the authentication performed by the tag this means, that the message authentication code has to be calculated over the init request payload, concatenated with the nonce the tag sent in the init response, as well as the output of the negotiated pseudo-random function with the ID payload of the corresponding node (excluding the fixed header) used as the seed. The ID payload is part of the encrypted payload, thus, first the decryption has to be performed. Due to the memory constraints of the tag, the procedure looks as follows:

1. The reader transmits the encrypted auth-request payload.

2. The tag performs the integrity check by calculation of the integrity checksum using the negotiated integrity check algorithm and comparison of the outcome to the integrity check value which was transmitted within the encrypted payload.

3. If the check succeeds, tag decrypts the payload and stores ID payload (excluding the header) and the AUTH payload.

4. The reader transmits the init-request payload of the corresponding node, which was sent during the init exchange as well as the nonce payload of the tag. In order to be able to do so, during the init exchange, the reader has to store the tag's nonce as well as the whole init-request message of the corresponding node.

5. The tag calculates the pseudo-random function over the ID payload and concatenates the outcome to the nonce and the init request payload. These are the *signed octets*, over which the AUTH value is calculated as described in Section 2.5.2.

6. The AUTH value just calculated is compared to the AUTH value that was transmitted by the corresponding node within the auth request. If the check fails, a notification is sent to the reader.

7. Otherwise, the reader again transmits the encrypted auth request payloads to the tag (the DemoTag does not have enough memory to hold both the auth request and the init request in its memory).

8. The tag again decrypts the successfully MAC'd payload and processes the actual payloads.

9. The tag then assembles the auth-response payloads by first calculating the AUTH payload. In order to do so, the reader has to send the tag the previously sent init-response message (which has had to be stored on the reader). After all payloads have been assembled, the tag has to encrypt them using the negotiated algorithm, calculate the integrity checksum and assemble the encrypted payload which is then transmitted to the reader

10. From the encrypted payload, the reader constructs the real IKE_AUTH_RESPONSE message and transmits it to the corresponding node.

From this list it can be seen that due to the small memory of RFID tags, a lot of communication overhead is produced. The protocol can be adapted based on the available memory.

3.2.3 The Create Child SA Exchange

The CREATE_CHILD_SA transform again has its payloads encrypted. The reader again forwards the encrypted payload to the tag, which performs the integrity check and decrypts the payloads. The CREATE_CHILD_SA request contains new Diffie-Hellman values. The Diffie-Hellman exchange is performed the same way as it was done with the init transform, using the elliptic curve cryptography core. Using the new shared secret, new keying material is created on the tag by calculation of KEYMAT (see Section 2.5.3). Depending on the memory available, the tag might again have to request the nonces which were exchanged during the init transform from the reader for this purpose. The tag assembles the response payload, calculates the integrity check value, encrypts the payloads and transmits it to the reader. The reader constructs the actual CREATE_CHILD_SA response and sends it to the corresponding node. At this point the security association for IPsec is established.

Using the just negotiated security association including the keys, a secure end-to-end communication between the corresponding node and the tag is now possible. ESP in transport mode is used for that purpose.

3.2.4 Cryptographic Algorithms

From the cryptographic algorithms for IKEv2/IPsec listed by the IANA (see [2]), one of each group is implemented. Except for the Diffie-Hellman exchange all chosen algorithms can be executed using the AES. Hardware and software implementations for the AES were provided by IAIK.

Encryption: AES in Cipher Block Chaining Mode

For encryption, the AES in cipher block chaining (CBC) mode is used. RFC 3602 (see [11]) defines the AES-CBC for use with IPsec. In CBC mode the ciphertext block c_i is dependent on the current plaintext block m_i and all previous plaintext blocks. In [25], the CBC mode is defined as follows: Let b be the block size, c_i the i-th ciphertext block of length b and m_i the i-th plaintext block of length b, e_k the encryption function with key k, and IV the initialization vector of length b. The first block then is encrypted by: $c_1 = e_k(m_1 \oplus IV)$, and consecutive blocks are encrypted by: $c_i = e_k(m_i \oplus c_{i-1})$.

Integrity: AES-XCBC-MAC-96

Using AES for message authentication with IPsec is described in RFC 3566 (see [12]). The AES-XCBC-MAC-96 algorithm uses AES with a 128-bit key. The algorithm is calculated as follows, where K denotes the 128-bit key and M denotes the message:

- Generate key k_1 by encrypting 0x01010101010101010101010101010101 with K.
- Generate key k_2 by encrypting 0x02020202020202020202020202020202 with K.
- Generate key k_3 by encrypting 0x03030303030303030303030303030303 with K.

E is then defined as:

$$E[i] = e_{k1}(M[i] \oplus E[i-1]) \text{ with } 0 < i < n, \tag{3.1}$$

where $E[i]$ is the i-th block of E and $M[i]$ the i-th block of M, $E[0] = 0$, e_{k1} denotes encryption with key k_1. The n-th block of M is then calculated as follows: If the size of the last block of M is 128 bits, $E[n] = e_{k1}(M[n] \oplus E[n-1] \oplus k2)$. If the size of the last block of M is less than 128 bits, $M[n]$ is padded to the length 128-bits with a single "1" bit followed by zeros. Thereafter, the last block of E is calculated as $E[n] = e_{k1}(M[n] \oplus E[n-1] \oplus k3)$. The 96 most significant bits of E then are the MAC value.

Pseudo-random Function: AES-XCBC-128

The AES algorithm can also be used as a pseudo-random function as described in RFC 4434 (see [17]). The AES-XCBC-128 algorithm is almost identical to the AES-XCBC-MAC algorithm described above. The key is created by either using an 128-bit key, or, if the key is shorter, it has to be padded with zeros to 128 bits, or, if it is longer, shortened to 128 bits by using the AES-XCBC-128 algorithm.

Diffie-Hellman Group: 192-bit Random ECP Group

For the Diffie-Hellman key agreement the 192-bit random elliptic-curve group over $\mathbb{F}_p$ was chosen (which is equivalent to the NIST P-192 curve).

Chapter 4

Design Exploration

In this chapter, we present the design exploration regarding the hardware implementation of the ECC core. After a brief introduction and a general outline, the NIST reduction is introduced in Section 4.3, including possible hardware implementations and optimizations. In Section 4.4, we explore various multiplier designs. In Section 4.5, we present a possible hardware design for a multi-precision Comba multiplier. The remainder of this chapter deals with the evaluation of diverse concepts for hardware multiplication as well as reduction. In Sections 4.6, 4.7, and 4.8, a Comba multiplier is used in combination with different reduction strategies. In Section 4.9, we consider the usage of a 16-bit multiplier along with a 32-bit adder. In Section 4.10, we look into the combination of a Comba multiplier and a Karatsuba-Ofman multiplier. In Section 4.11, a Karatsuba-Ofman multiplier with serial multiplication is investigated. Finally, in Section 4.12, we explore various shift-add multiplier designs.

4.1 Introduction

Due to the resource constrained nature of RFID tags, the suitability of an elliptic curve cryptography core design for use on the tag must be carefully evaluated. Ideally, such an elliptic-curve cryptography core should be fast, should have a low area, and should consume only low power. For deployment on an RFID tag, a reasonable trade off between these characteristics must be found.

This Chapter, as well as Chapter 5, includes various estimations regarding the chip area. For these estimations the UMC-L130 CMOS technology was used. Table 4.1 shows some few basic standard cells and their respective area requirements that were used for the following estimations.

Table 4.1: Area requirements of the standard cells used for estimations of the chip area in this chapter.

Cell name	**AND**	**MUX3**	**MUX4**	**FA**
Area [GE]	1.25	3.1	4.9	5.5

4.2 Searching for an Efficient Datapath Design

This chapter focuses on the evaluation of various designs of the crypto core's datapath that were considered on the way to the final (chosen) design described in Chapter 5. Besides efficiency, attention was paid to designing the crypto core in such a way that it is robust to *side-channel attacks*. The term *side-channel attack* describes a family of attacks on cryptographic algorithms, where not the algorithm itself, but the implementation is attacked. Thereby, such attacks exploit information leakage of various forms. For example, the power trace of a cryptographic system, or the duration of certain cryptographic operations can reveal secret information (cf. [15]). A important factor contributing to robustness against side-channel attacks is constant execution time leading to two somewhat conflicting design methodologies. Thus, the intention was to implement the crypto core with aim on constant execution time, while making it use as view clock cycles as possible although keeping the chip area small.

Elliptic curves over binary fields can be implemented very efficiently in hardware: No carry propagation must be considered for addition operations, since addition is performed bitwise. So, instead of full adders simple XOR gates can be used for that purpose. Multipliers also benefit from this property, resulting in a much smaller chip area as can be seen in [32]. That is why, elliptic curves over binary fields are often preferred in hardware realizations. However, upon closer examination of the Internet Key Exchange Version 2 protocol and its parameters (see [2]), it turned out that only elliptic curves over prime fields are supported. Since this would generally lead to a slower and larger crypto core, the curve P-192 was selected due to its beneficial characteristics for implementation in hardware as described in Section 4.3. On the one hand, it allows for very fast reduction algorithms, on the other hand it is relatively small, what makes it a good fit for implementation on resource constrained devices.

4.3 Exploiting the NIST Reduction

4.3.1 Terminology

Finite Fields

A set $\mathbb{F}$ containing a finite number of elements, together with the operations addition and multiplication is called a *finite field*.

Prime Fields

The finite field $\mathbb{F}_p$, where p is a prime number, is called a *prime field*.

Reduction Modulo Prime

The arithmetic operations addition and multiplication in prime fields can be performed by first computing their corresponding integer arithmetic operations, followed by a *reduction* (mod p). This reduction modulo p of an integer i denotes the remainder r that is obtained by the integer division i/p, such that $0 \leq r < p$ [15].

4.3.2 Fundamentals

Five elliptic curves over prime fields are recommended by the National Institute of Standards and Technology (NIST) in the FIPS 186-3 standard (see [14]), namely the curves P-192, P-224, P-256, P-384, and P-521. The pseudo-random elliptic curve used is denoted as follows:

$$E : y^2 \equiv x^3 + ax + b \pmod{p}. \tag{4.1}$$

For efficiency reasons, the coefficient a was selected to be -3 (as seen in [14]) leading to the curve:

$$E : y^2 \equiv x^3 - 3x + b \pmod{p}. \tag{4.2}$$

The moduli p of the five recommended prime fields are:

$$p = 2^{192} - 2^{64} - 1 \quad \text{P-192}$$
$$p = 2^{224} - 2^{96} + 1 \quad \text{P-224}$$
$$p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 \quad \text{P-256}$$
$$p = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1 \quad \text{P-384}$$
$$p = 2^{521} - 1 \quad \text{P-521}$$

These five NIST primes have in common that they are Mersenne-like primes. This means, as depicted above that each of them can be written as a sum of differences $\pmod{p}$ of powers of two, which leads to fast reduction algorithms as follows (see [15]): First, the integer i to be reduced, where $0 \leq i < p^2$ and p is the prime (e.g., for the curve P-192: $p = 2^{192} - 2^{64} - 1$), is represented as a series of equal-sized chunks. For our example prime, the base-2^{64} representation would be suitable. Second, higher powers of two, namely those that exceed the bit width of the prime, are then substituted by their congruences, e.g., $2^{192} \equiv 2^{64} + 1 \mod p$. Thus, by rewriting i using the congruences, it can be seen that the majority of the reduction modulo prime can be accomplished by simply shifting higher powered coefficients to their respective lower positions followed by the calculation of the sum (and possibly difference). Finally, the fast reduction modulo prime is completed by (repeated) subtraction of the modulus until $i < p$. The principle of the fast reduction modulo prime is depicted in Figure 4.1 for our example prime (excluding the final subtraction): The 384-bit integer i is divided into six 64- bit chunks. According to the used congruences the three most-significant chunks A, B, and C are shifted to lower significant positions. The sum of the four depicted words modulo p_{192} is congruent to i and thus final repeated subtraction of p_{192} yields the reduced integer. For further details we refer to [15].

4.3.3 Possible Hardware Implementations of the NIST Reduction

In the previous section we have seen that the NIST fast reduction can be accomplished by adding or subtracting chunks of the *high_word* of i (i.e., $(2^{383}, 2^{382}, ..., 2^{192})$ when using the P-192 curve) at certain positions to the *low_word* of i (i.e., $(2^{191}, 2^{190}, ..., 2^0)$), followed by subtraction of the modulus until $i < p$. Thus, a hardware implementation of the NIST fast reduction needs an addition/subtraction circuit as well as some sort of stop condition for the final repeated subtraction of the modulus. There are two obvious approaches for implementation of the addition/subtraction circuit: a high-speed design and a low-area design.

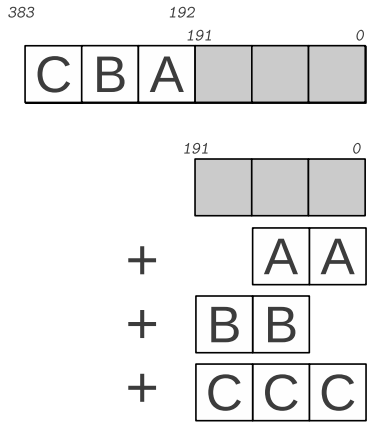

Figure 4.1: Principle of the fast reduction modulo p_{192}: The three most-significant 64-bit chunks of the 384-bit integer i are shifted to lower positions according to their modulo congruences. The sum of the four depicted 192-bit words (mod p) is congruent to i.

The high-speed approach would be to use one individual adder for each addition (e.g., three 192-bit adders when disregarding any carry propagation for the P-192 curve, see Figure 4.1), in order to calculate the reduction (except for the final subtractions) in one clock cycle. However, experience has shown that an adder needs approximately 5.5 GE/bit when synthesized, which leads to an area of $\approx 1\,056\,\text{GE} * 3 = 3\,168\,\text{GE}$ just for the reduction. Another disadvantage of this approach is the big propagation delay through the adders, leading to a reduced maximum clock frequency and therefore reducing the speed advantage.

The low-area approach would be to repeatedly use one adder of width w and an accumulator register of with $m = n * w$, where m is the bit width of the modulus. Since only one w-bit addition is computed per clock cycle, the speed of this approach is highly dependent on the factor n. For example, when using the curve P-192 (thus $m = 192$), and let $n = 3$, the bit width of the adder must be 64. Thus, with this configuration the NIST fast reduction (minus the final subtractions) can be computed in $3 * 3 = 9$ clock cycles (again, disregarding carry propagation). Even though this approach needs a slightly more complicated control circuit in practice, the synthesized circuit should yield a much lower area than the high-speed approach. Both the estimation of the required area and the estimation of the number of clock cycles are highly dependent on the width of the adder.

4.3.4 Further Optimizations

When elliptic-curve operation formulae are implemented in a straight-forward manner, reduction can severely contribute to the number of clock cycles needed. Thus, reduction is a rather expensive operation. The reason for this is, that each field operation (i.e., addition modulo p, subtraction modulo p, multiplication modulo p and field inversion) requires a reduction step. However, since RFID tags are resource constrained, taking the low area approach as a basis and optimizing it seems more appropriate. For the reason of a reduced chip area, since the datapath of the elliptic curve cryptography core needs an adder for field addition anyway, this same adder can be reused for reduction. In the following, some strategies for optimizations of the NIST reduction are given:

- By closer investigating the fast reduction principle for the curve P-192 in Figure 4.1, it can be seen that in case of a 64-bit datapath one addition can be saved, since there are only two additions necessary for generation of the least-significant 64-bit word. When using a 32-bit datapath two additions can be saved, and when using a 16-bit datapath, 4 additions can be saved.

- Further, the repeated subtraction of the modulus at the end of the reduction step can lead to a varying execution time: Suppose we again use a 64-bit datapath and the P-192 curve. After performing the fast reduction until the final subtractions, the intermediate result can still take up to 193-bit due to carry propagation. Thus, for full reduction of i there can be either two, one or zero subtraction steps necessary. However, we need to achieve constant execution time in order to prevent side channel attacks.

  One way to achieve constant execution time is to perform a second reduction round: First, an ordinary NIST reduction is executed as described above, excluding the final subtractions. Now, independent of the actual value of the *carry* (the most significant bit), the procedure is repeated (with some simplifications). If the carry bit is zero (the integer already only uses 192 bit), this second iteration does not change anything because only zeros are added. Only after the second iteration was completed, the final subtraction step is performed. Due to the second iteration, the intermediate result must be $< 2^{192}$ and thus the modulus has to be subtracted once at a maximum. The execution time difference now can be cleared by subtracting *zero* in case $i < p$ already.

  The simplifications for the second reduction iteration are as follows: Since after the first iteration the intermediate result can only take up one bit more than the modulus (up to 193-bit when using the curve P-192), only part of the reduction has to be carried out. In case of the curve P-192, this means that only the chunk A (which in turn is zero at this point with the possible exception of the least significant bit) has to be added to the *low_word* (see Figure 4.1) and thus only three additions (one for adding A to the lowest 64 bits of the *low_word*, one for adding A to the next 64 bits of the *low_word* and a final one to add a possible carry to the most significant 64 bits of the *low_word*) are necessary when using a 64-bit datapath. The 192th bit at this point can never generate a carry: Let's assume the integer i to be reduced is $i = p^2 - 1$, which is the largest possible integer where the NIST reduction as described above still applies. Computing the reduction on i leads to:

$$A = FFFFFFFFFFFFFFFE_h$$
$$B = FFFFFFFFFFFFFFFD_h$$
$$C = FFFFFFFFFFFFFFFF_h$$
$$S_1 = A * 2^{64} + A$$
$$S_2 = (B * 2^{64} + B) * 2^{64}$$
$$S_3 = C * 2^{128} + C * 2^{64} + C$$
$$i_{reduced} = i \mod 2^{192} + S_1 + S_2 + S_3$$
$$i_{reduced} = 0x1FFFFFFFFFFFFFFFFFFFFFFF...$$
$$...FFFFFFFFDFFFFFFFFFFFFFFFD_h < 2^{193}$$

It can be seen that $i_{reduced}$ takes up 193 bits but not all bits are set, thus, in the second reduction step any occurring carry is absorbed before it propagates to the 193th bit:

$$A = 1$$
$$B = C = 0$$
$$S_1 = A * 2^{64} + A$$
$$S_2 = S_3 = 0$$
$$i_{reduced} = i \mod 2^{192} + S_1 + S_2 + S_3$$
$$i_{reduced} = 0xFFFFFFFFFFFFFFFFFFFFFFFF...$$
$$...FFFFFFFEFFFFFFFFFFFFFFFE_h < 2^{192}$$

Since this is true for the largest possible integer i for which the NIST reduction is still defined, this is generally true for all integers i where $0 \leq i < p^2 - 1$.

- Another big advantage of the multi-iteration reduction as described above is, that subtraction can be neglected until the elliptic curve point multiplication is completed. Only then, the final subtraction is carried out, leading to a huge improvement regarding the number of clock cycles necessary. This reduction of computational costs is possible due to the fact that by performing a multi-iteration reduction after each field operation, each term is reduced to its k-bit representation where $k = 2^{\lfloor \log_2 p \rfloor}$, e.g., 192-bit representation when using the curve P-192. Thus, the width of the components in the datapath (like registers, adders, and multiplexers) is sufficient for correct operation and no arithmetic overflow is produced, ensuring correct operation of the circuit.

  Table 4.2 shows a comparison of the number of final subtractions needed for a full point multiplication. The numbers were generated by addition of the number of field operations (thus operations that need a reduction step afterwards) necessary for a complete point multiplication. The used formulae was the Montgomery ladder with (X,Y)-only co-Z doubling-addition, which is shown in Algorithm 9, respectively Algorithms 10, 11, and 12, taken from [29]. A listing of the necessary field operations is depicted in Table 4.3.

- The approach can be refined even further when considering the following: Multiplication of two p-bit integers i_1 and i_2 yields a $(2 * p)$-bit integer j at a maximum, e.g., multiplication of two 192-bit integers yields an integer j that takes up up to 384-bit, and addition of two p-bit integers i_1 and i_2 yields a $(p + 1)$-bit integer j at a maximum, e.g., addition of two 192-bit integers yields an integer j that takes up up to 193-bit. This means, that after an addition the bit width of the datapath is exceeded by only one bit and thus only a reduced reduction step as described above is necessary. This leads to another improvement of the execution time.

Table 4.2: Final subtraction costs for various reduction techniques. The number of necessary reductions was estimated for a complete elliptic-curve point multiplication using the Montgomery ladder with (X,Y)-only co-Z doubling-addition (see Table 4.3 and [29])

Maximum # final subtraction operations				
	Curve P-192			
	192-bit DP	64-bit DP	32-bit DP	16-bit DP
Standard NIST reduction	17 608	52 824	105 648	211 296
Multi-iteration NIST reduction	8 804	26 412	52 824	105 648
Partial reduction	1	3	6	12

4.4 Exploring Various Multiplier Designs for use in the Datapath

4.4.1 Introduction

In order to compute prime field multiplications which are part of the elliptic curve point multiplication in hardware, a multiplier is needed. A hardware multiplier can be implemented in various ways. Three distinct designs were considered for the design of the crypto core. These are:

- Array Multiplier
- Karatsuba-Ofman Multiplier
- Shift-Add Multiplier

Each of those multiplier designs comes with certain advantages and disadvantages. For use in constrained environments, e.g., an RFID tag, an adequate trade off between speed and area has to be made. Although the area needed by the multiplication circuit should of course be kept as small as possible, multiplication must not need too many cycles: RFID tags are often just passing the electromagnetic field of the reader in a short amount of time, as for example a product with an RFID tag attached to it passing the reader in an assembly line, or an RFID-enabled passport being automatically checked while the owner is passing by an RFID reader. By investigation of Table 4.2, it can be seen that multiplication cost is a major factor contributing to the overall cost of elliptic-curve point multiplications. If the chosen multiplication circuit is too slow, the examples mentioned above would become infeasible.

4.4.2 Array Multiplier

The fastest (with respect to the number of clock cycles needed for completion of one multiplication operation) of the three examined multiplier architectures is the basic array

Table 4.3: Computational costs for Montgomery ladder with (X,Y)-only co-Z doubling-addition, see [29]

Cost per bit (n-2 times)			Additional cost			
MUL	SQU	ADD	INV	MUL	SQU	ADD
8	6	32	1	18	10	35

multiplier, which is depicted in Figure 4.2 for a word size of 5 bit. As described in [26], the propagation delay through such an adder structure is rather long, but its structure is highly regular which makes it an efficient layout for VLSI. Each level of the array multiplier computes the result of the multiplication $a_n * b$ (where a_n is the nth-bit of a), that is, each level adds a shifted version of the previous level to the term b or 0, depending on whether $a_n = 1$ or $a_n = 0$.

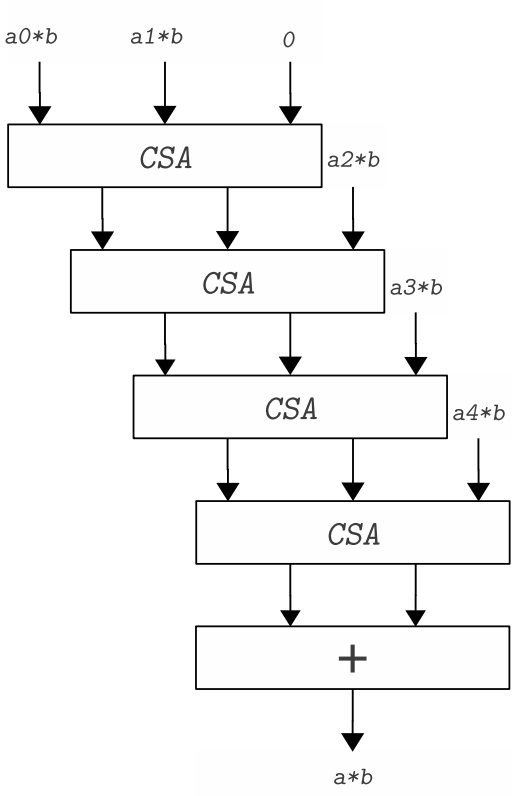

Figure 4.2: Block diagram of a basic 5x5 bit array multiplier, composed of four carry-save adders and one ripple-carry adder.

Such an array multiplier computes the result c of the multiplication $c = a*b$ in only one clock cycle. However, the short computation time comes with rather large area requirements. In [32], Wenger and Hutter compared the number of gate equivalents consumed by different hardware array-multiplier types at various datapath widths. An excerpt of their results is shown int Table 4.4. From these values a steep growth of the consumed area with the bit width can be observed. Going from a 16-bit array multiplier to one with 32-bit width, multiplication can be sped up, but with the drawback of a growth of the required area by a factor greater than four. Thus, for use in the constrained environment that is an RFID tag, usage of array multipliers of width greater than 32 bit seems not practical. Thereby the maximum datapath width for the crypto core is reduced to 32 bits. In order to compute a full 192-bit multiplication (required for the P-192 curve), some means of multi-precision multiplication is required, slowing down multiplication operation.

4.4.3 Shift-Add Multiplier

Multiplication of the binary representation of two integers a and b, where a is the multiplicand and b is the multiplier, can be decomposed to additions and bit shifts. For each bit of the multiplier b either zero or a shifted version of the multiplicand a is added to an accumulator (or the multiplicand is added to a shifted version of the accumulator). There are two very similar takes on this shift and add principle, namely the

1. Multiply and shift-left algorithm
2. Multiply and shift-right algorithm

An in-depth explanation of those algorithms can be found in [26]. In short, both algorithms add partial products to an accumulator (which is initialized to zero). The partial products are computed by the multiplication $a * b_n$, where b_n is the n-th bit of the multiplier b. The product therefore is either zero or a. Before it is added to the accumulator, the product is shifted by the appropriate number of bits (or the accumulator is shifted). The difference between the *Multiply and shift left* algorithm and the *Multiply and shift right* algorithm is the direction of the shift.

The shift-right version starts with the least significant bit of the multiplier b, adds the product $a * b_n$ to the accumulator and shifts the accumulator one bit to the right. The procedure is repeated until all partial products have been added to the accumulator which then contains the product $a * b$.

The shift-left version starts with the most significant bit of the multiplier b, adds the product to the accumulator and shifts the accumulator one bit to the left. Again, the procedure is repeated until all bits of b have been evaluated. Both algorithms are completed after the same number of shifts and additions, which means that both algorithms also need the same number of clock cycles to complete. However, as shown in [26], the shift-left algorithm might be less efficient due to the fact that by left-shifting the accumulator the partial products are added to the least significant bits of the accumulator and thus carry propagation might affect more bits.

The array multiplication described in Section 4.4.2 is also based on the shift and add principle. However, as opposed to the shift-add hardware multiplier design described in this section, with the array multiplier all partial products are generated and added within one clock cycle.

A possible hardware implementation of a shift-add multplier (shift-right version) is depicted in Figure 4.3. It consists of two k-bit operand registers, a k-bit adder (e.g., a ripple carry adder), a $2k$-bit accumulator register and a k-bit 2-input multiplexer. Since one of the k-bit multiplexer inputs is always zero, an and gate can be used instead, resulting in reduced area. The operand register containing the multiplier, as well as the accumulator register are implemented as shift registers with parallel load input. Thus, multiplication of two k-bit operands consumes k cycles (one cycle for each multiply-accumulate step). At each clock cycle the least significant bit of the multiplier register is used as the select

Table 4.4: Area costs for hardware array-multipliers of different bit widths

Finite Field	**Required adder cells per bit**	**8 bit [GE]**	**16 bit [GE]**	**32 bit [GE]**	**64 bit [GE]**
$GF(p)$	FA	376	1 616	6 688	26 336

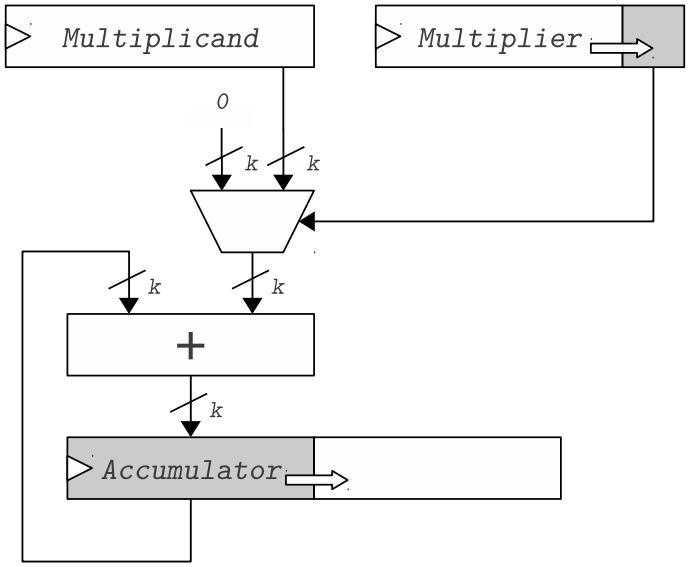

Figure 4.3: Simplified block diagram of a sequential hardware realization of a shift and add multiplier (shift right version).

input of the multiplexer (or as one of the inputs of the and gates), which in turn supplies either the multiplicand or zero to the adder. The second input of the adder is wired to the accumulator. Therefore, the most significant k-bits of the accumulator register must be initialized to zero. The addition result is stored within the most significant k-bits of the accumulator and both the accumulator and the multiplier register perform a right shift. Due to the right-shift, the MSB of the accumulator is always zero (when the addition is performed), therefore no carry bit is generated by the addition.

Area Requirements The estimated chip area for a shift-add multiplier as depicted in 4.3 mainly consists of the area consumed by the three registers and the adder. Experience has shown that a rough estimate for the area consumption of one bit of a ripple-carry adder is approximately 5.5 GE and for one bit of a register is approximately 6 GE. Since one of the two two-input multiplexer inputs is always zero in the depicted shift-add circuit, the multiplexers can be substituted by 2-input and gates. If we estimate the area of such an and gate by 1.25 GE, a coarse grained estimation for the area a of the whole shift-add multiplier would be (where w_x denotes the bit-width of component x):

$$a_{shift-add} \approx (k_{multiplicand} + w_{multiplier} + w_{accumulator}) * 6 + w_{RCA} * 5.5 + w_{and} * 1.25$$
$$a_{shift-add} \approx 4 * k * 6 + k * 5.5 + k * 1.25$$

For multiplication of two 192-bit operands with such a shift-add multiplier, applying the formulae above leads to an estimated area of 5 856 GE. The cycle count for one k-bit multiplication is k, so the 192-bit shift-add multiplication is completed within 192 clock cycles. Of course, for use of the shift-add multiplier in the datapath of an elliptic curve cryptography core, the operand registers are not solely belonging to the multiplier but are

Table 4.5: Comparison of cycle counts and required chip area of array multipliers and shift-add multipliers at various bit widths.

	Array Multiplier		Shift-Add Multiplier		Shift-Add Multiplier (incl. operand registers)
bit width	**# cycles**	**area [GE]**	**# cycles**	**area [GE]**	**area [GE]**
8	1	376	8	150	246
16	1	1 616	16	300	492
32	1	6 688	32	600	984
64	1	26 336	64	1 200	1 968
192	-	-	192	3 600	5 904

the same for the whole datapath. Thus, they are excluded from the area estimation below and the formulae is reduced to:

$$a_{shift-add} \approx (w_{accumulator}) * 6 + w_{RCA} * 5.5 + w_{and} * 1.25$$
$$a_{shift-add} \approx 2 * k * 6 + k * 5.5 + k * 1.25$$

Table 4.5 shows a comparison between the k-bit array multipliers and k-bit shift-add multipliers. The area costs for the array multiplier are taken from [32]. It can be seen, that for small bit widths an array multiplier offers the best trade off between the number of clock cycles required and the chip area. However, as the bit width grows, because of the array multiplier's steep growth in area, the shift-add multiplier becomes more attractive. This is due to the fact that, unlike the array multiplier, the shift-add multiplier grows linearly with the bit width. At a width of 192 bit for example, the shift-add multiplier has an estimated chip area of $\approx 3\,600$ GE (or $\approx 5\,904$ GE when the operand registers are included), which is smaller than the area of an array multiplier at 32 bit width.

For the decision upon whether an array multiplier or a shift-add multiplier should be deployed in an elliptic curve cryptography core for RFID tags (for the curve P-192), some further considerations have to be made. First, although it is comparably small, a standard 192-bit shift-add multiplier is rather slow: When the computational costs for the Montgomery ladder with (X,Y)-only co-Z doubling-addition (as shown in Table 4.3) are applied, the number of clock cycles used by the multiplier alone sums up to $295\,296\,cycles$. Second, use of a 192-bit array multiplier is off limits due to its demanding chip area consumption. Therefore both the array multiplier and the shift-add multiplier must be combined with another approach in order to become feasible. A possible approach would be to use either multiplier as the foundation of a multi-precision multiplication circuit, where a smaller multiplier is sequentially used in order to compute the full product. There are two well-known techniques for multi-precision multiplication, namely the

1. Operand-scanning integer multiplication
2. Product-scanning (or Comba) integer multiplication

algorithm. Comba multiplication is looked into more detail in Section 4.4.4.

4.4.4 Comba Multiplier and Squarer

Multiplication Product scanning (or Comba) integer multiplication is a multi-precision integer multiplication algorithm. Implemented in hardware, it can be used together with

an arbitrary multiplication circuit (like an array multiplier or a shift-add multiplier) at its core. The algorithm (taken from [15]) is shown in Algorithm 3 in form of a pseudo code. (UV) denotes the concatenation of the w-bit words U and V, R_0, R_1, and R_2 also have a width of w-bit, t is the word length. Comba's multiplication algorithm takes two large operands a and b and chops them into words of size t in order to fit the t-bit multiplier it uses. The advantage of this multiplication method is, that only a hardware multiplier for short word sizes is necessary. However, the multiplication takes many cycles to complete, and the cycle count grows with shorter word sizes. For the purpose of deployment in an elliptic curve cryptography core, various word lengths have been evaluated in this work (please refer to Section 4.5). When implemented with an array multiplier, each partial product $(UV) = A[i] * B[i]$ can be computed within one clock period and the algorithm takes $O(n^2)$ elementary operations [15].

Algorithm 3 Integer multiplication (Comba's method)

Require: $a, b \in [0, p-1]$
Ensure: $c = a * b$
1: $R_0 \leftarrow 0$;
2: $R_1 \leftarrow 0$;
3: $R_2 \leftarrow 0$;
4: **for** $k = 0$ to $2t - 2$ **do**
5: **for** each element of $\{(i, j) | i + j = k, 0 \leq i, j \leq t - 1\}$ **do**
6: $(UV) \leftarrow A[i] * B[i]$;
7: $(\epsilon, R_0) \leftarrow R_0 + V$;
8: $(\epsilon, R_1) \leftarrow R_1 + U + \epsilon$;
9: $R_2 \leftarrow R_2 + \epsilon$;
10: **end for**
11: $C[k] \leftarrow R_0$;
12: $R_0 \leftarrow R_1$;
13: $R_1 \leftarrow R_2$;
14: $R_2 \leftarrow 0$;
15: **end for**
16: $C[2t - 1] \leftarrow R_0$;
17: **return** c;

Squaring A slightly adapted version of Algorithm 3 is shown in Algorithm 4 (taken from [15]). It shows a realization of a multi-precision integer squaring algorithm with the benefit of the number of required single-precision multiplications roughly reduced by half. Of course, Algorithm 3 could be used for squaring too by applying operator a to both inputs.

4.4.5 Karatsuba-Ofman Multiplier

Theoretical Background

Karatsuba-Ofman multiplication utilizes a divide and conquer approach for integer multiplication, resulting in a complexity of $O(n^{log_2 3})$ as opposed to the complexity of Comba's method, which is $O(n^2)$ [15]. The principle of the Karatsuba-Ofman multiplication is to

Algorithm 4 Integer squaring

Require: $a \in [0, p-1]$
Ensure: $c = a^2$
1: $R_0 \leftarrow 0$;
2: $R_1 \leftarrow 0$;
3: $R_2 \leftarrow 0$;
4: **for** $k = 0$ to $2t-2$ **do**
5: **for** each element of $\{(i,j)|i+j=k, 0 \leq i \leq j \leq t-1\}$ **do**
6: $(UV) \leftarrow A[i] * A[j]$;
7: **if** $(i < j)$ **then**
8: $(\epsilon, UV) \leftarrow (UV) * 2$;
9: $R_2 \leftarrow R_2 + \epsilon$;
10: **end if**
11: $(\epsilon, R_0) \leftarrow R_0 + V$;
12: $(\epsilon, R_1) \leftarrow R_1 + U + \epsilon$;
13: $R_2 \leftarrow R_2 + \epsilon$;
14: **end for**
15: $C[k] \leftarrow R_0$;
16: $R_0 \leftarrow R_1$;
17: $R_1 \leftarrow R_2$;
18: $R_2 \leftarrow 0$;
19: **end for**
20: $C[2t-1] \leftarrow R_0$;
21: **return** c;

split the two n-bit operands a and b into their respective low- and high words a_{lo}, a_{hi}, b_{lo} and b_{hi}.

$$a * b = (a_{hi} * 2^l + a_{lo}) * (b_{hi} * 2^l + b_{lo}) \tag{4.3}$$

Equation 4.3 shows how the multiplication of a and b looks like after the split ($l = n/2$). Expanding the equation leads to:

$$a_{hi} * b_{hi} * 2^{2*l} + ((a_{hi} + a_{lo}) * (b_{hi} + b_{lo}) - a_{hi} * b_{hi} - a_{lo} * b_{lo}) * 2^l + a_{lo} * b_{lo} \tag{4.4}$$

From Equation 4.4 it can be seen that the one multiplication $a * b$ can be substituted by three multiplications of shorter width as well as two additions and two subtractions. In cases where multiplication costs vastly exceed addition (respectively subtraction) costs such an approach can become feasible. The Karatsuba-Ofman algorithm can be applied recursively, that is, each of the three multiplications ins Equation 4.4 can again be split into three multiplications, two additions and two subtractions.

Hardware Implementations

A possible hardware implementation of one recursion step of a Karatsuba-Ofman multiplier is depicted in Figure 4.4, on the example of 32-bit operands a and b. Registers *r1* and *r0* hold the operands a and b split into their respective high- and low words. Within one clock cycle, formulae 4.4 is executed, that is, instead a 32-bit hardware multiplier, two 16-bit and one 17-bit multipliers, two 16-bit adders and two 34-bit subtractors are used for computation.

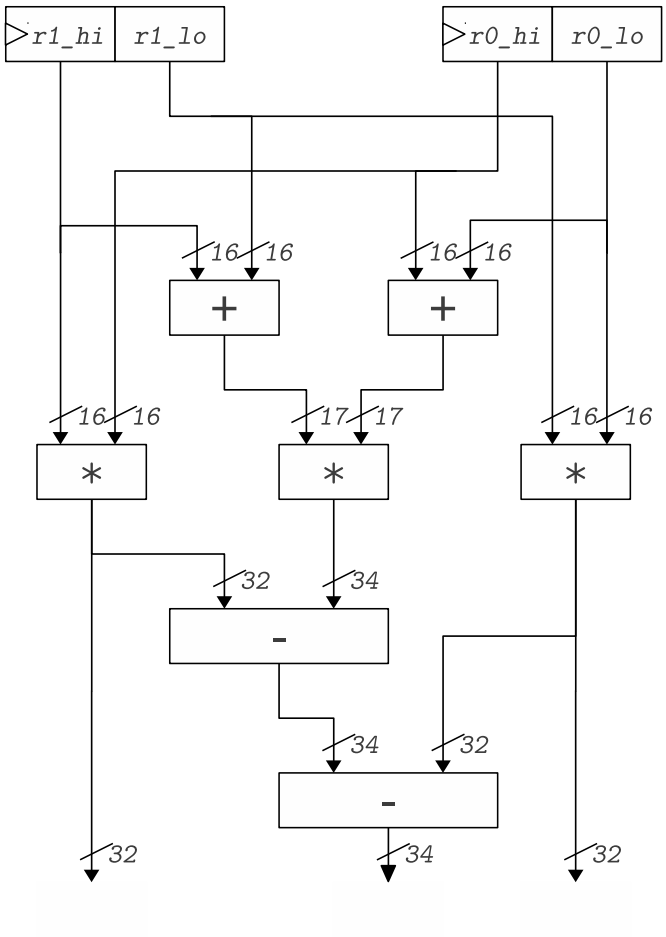

Figure 4.4: Simplified block diagram of a Karatsuba-Ofman multiplier (one recursion step), reducing one 32-bit addition to three 16-bit respectively 17-bit multiplications, two 16-bit additions and two 34-bit Subtractions.

There are two things to note with respect to the block diagram: First, the multiplier taking taking the addition results as its input needs to be able to handle $l+1$-bit operands, because of possible carry generation at the adder stage. Similarly, the subtractors have to be of width $l+2$ for the same reason. Second, only a simplified version of a hardware realization of the Karatsuba-Ofman multiplier is shown. What is missing in the block diagram is the output stage, which combines the three intermediate results to the $2 * l$-bit result c of the multiplication $c = a * b$. This output stage would consist of two further adders. Table 4.6 shows a comparison regarding chip area of a 32-bit array multiplier and various Karatsuba-Ofman multipliers as depicted in Figure 4.4, with different recursion depth. For simplicity reasons carry propagation is not included in the comparison, therefore all three multipliers of one recursion level have the same width and the subtractors have the same width as the input values of the respective stage. A hardware implementation of the first recursion step of a Karatsuba-Ofman multiplier used for the estimation in Table 4.6 consists of the following basic blocks:

Table 4.6: Area estimation results for 32-bit Karatsuba-Ofman multipliers with varying recursion depths. For comparisons sake the area required by a 32-bit array multiplier is included in the table.

	Array multiplier	**Karatsuba-Ofman multipliers (varying depth)**		
		One level	Two levels	Three levels
Area [GE]	6 688	5 640	5 364	6 138

- Three 16-bit multipliers
- Three 16-bit adders
- Three 32-bit subtractors
- Additionally: three 16-bit adders for recombination of the three partial output values to the 64-bit multiplication result

A second recursion step can be implemented by substituting each of the three 16-bit multipliers by:

- Three 8-bit multipliers
- Three 8-bit adders
- Three 16-bit subtractors
- Additionally: three 8-bit adders for recombination of the three partial output values to the 32-bit multiplication result

A third recursion step can be implemented by again substituting each of the three 8-bit multipliers by:

- Three 4-bit multipliers
- Three 4-bit adders
- Three 8-bit subtractors
- Additionally: three 4-bit adders for recombination of the three partial output values to the 16-bit multiplication result

From the area estimations given in the table it can be seen that for a 32-bit Karatsuba-Ofman multiplier the second recursion step reduces the estimated chip area, but at the third recursion step the area of the additional adders and subtractors begins to dominate since the area of the multipliers is getting rather small. What can also be seen is, that the two level Karatsuba-Ofman multiplier, which is the smallest of the three shown in the table does not offer a vast improvement regarding the area consumption compared to the array multiplier. For the P-192 curve, we would need a 192-bit Karatsuba-Ofman multiplier which would still be too big for use in an elliptic curve cryptography core for RFID tags. However, combined approaches, e.g., where the Karatsuba-Ofman multiplier is used together with Comba's method were evaluated (see 4.10).

4.5 Hardware Design of the Multi-precision Multiplier and squarer

4.5.1 Multiplier

Integer multiplication by operand scanning and integer multiplication by product scanning (Comba's method) are two elementary multi-precision integer multiplication algorithms. For implementation in hardware, the latter of those two was chosen. Figure 4.5 shows the hardware design of a Comba multiplier, the pseudo code of which is depicted in Algorithm 3. Although the figure shows an implementation for a 16-bit datapath, a 32-bit version was considered too. The only differences between the 16-bit and the 32-bit version are the bit widths of the basic blocks like adders, multipliers and registers. The hardware architecture of the Comba multiplier was designed with respect to a low cycle count. Therefore, the multiply-accumulate step that is shown in lines 6 to 9 in Algorithm 3 is implemented to be executed within one clock cycle. First, the 16-bit operands a and b are multiplied (line 6), the 32-bit result (UV) of this multiplication is fed to a 32-bit adder. The second input of this adder is wired to the concatenation ($R1R0$) of two 16-bit registers which contain intermediate results from the previous iteration (lines 7 and 8). The carry of this addition is then fed to another 16-bit adder, together with the content of a third 16-bit register $R2$, also containing an intermediate result from the previous iteration (line 9). The results of the 32-bit adder and the 16-bit adder are then stored in the registers $R0$, $R1$, and $R2$ for the next iteration (lines 7 to 9). Lines 11 to 14 show a shift operation, where the content of $R0$ is stored in the *result* register, the content of $R1$ is shifted into $R0$ and so forth. As the goal for the hardware design of the Comba multiplier was a low cycle count, some chip area was sacrificed for two multiplexer stages, one before and one after the register bank. These multiplexers enable the shift to be handled implicitly, that is, by applying a certain control logic for the select signals, the register's outputs are wired to different adders and the output of the adders are wired to different registers, thus emulating the shift operation shown in lines 12 to 13. Hence, the shift operation is performed within the same clock cycle as the multiplication.

Area Estimation: An estimation of the chip area for the Comba multiplier shown in Figure 4.5 is provided in Table 4.7, for both a 16-bit and a 32-bit datapath. Bear in mind that the adders within the Comba multiplier could also be used for other operations in the datapath like multi-precision addition and reduction. Four-input multiplexers were estimated with area needs of 4.9 GE and three-input multiplexers were estimated with area needs of 3.1 GE. The output register *result* is not included in the estimation since its width is dependent on the width of the input integers. For the curve P-192, the register would have to hold $192 * 2 = 384$ bit and thus would need an estimated area of 2 304 GE. The table shows that the area of the multiplier is the dominating component. Comparing the area consumed by the multiplexers to that of the adders or the registers is also quite interesting: It shows that the optimization regarding cycle count indeed comes with the drawback of an increased chip area. However, wasting a clock cycle for the shift instead of the two multiplexer stages would lead to an increased cycle count by $\approx 2 * t$[1], severely slowing down the elliptic curve point multiplication. When implemented as shown in the figure, the multiplier would require 144 clock cycles with a 16-bit datapath and 36 clock cycles with a 32-bit datapath, in order to finish a 192-bit multiplication.

[1] Remember, t is the word size.

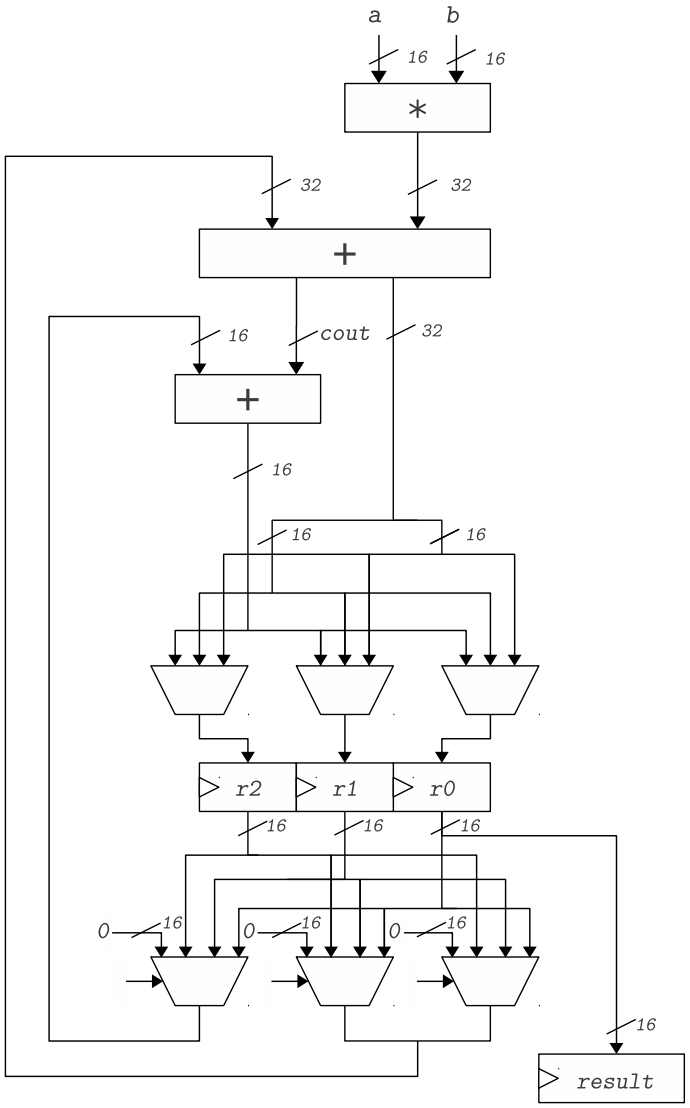

Figure 4.5: The datapath of a Comba multiplier operating on 16-bit word length.

4.5.2 Squarer

Integer squaring as depicted in Algorithm 4 is quite similar to the Comba integer multiplication. In fact, the only differences in their respective algorithms are the reduced iteration count of the squarer as well as a conditional multiplication by two of the intermediate value (UV) (lines 7 to 10). Thus, the hardware design of the squarer is also based on the hardware design of the Comba multiplier described above. Since only the datapath is shown in 4.5, the only difference here is the conditional multiplication. Figure 4.6 shows the datapath of the 16-bit integer squarer. Of course, the inputs a and b must be equal. The implicit shift unit (the register bank consisting of registers $r0$, $r1$, and $r2$, as well as their input and output multiplexers) is the same as the one of the Comba multiplier. However, the multiply-accumulate unit looks a bit different: After the multiplication of the operands a and b, the conditional multiplication by two is computed. In Figure 4.6 this

Table 4.7: Area estimation results of the Comba multiplier depicted in 4.5. Results are provided for both a 16-bit and a 32-bit version.

	Area [GE]	
	(16-bit)	**(32-bit)**
Multiplier	1 616	6 688
Adders	264	528
Multiplexers	463	925
Registers	288	576
$\sum$	2 631	8 717

step is depicted as a cloud. What actually happens between the output of the multiplier and the input of the adder is the following:

- If $i < j$ then
  - A left shift of the multiplier's output by one bit is performed (which is equivalent to the multiplication by two), and the result is wired to the 32-bit adder.
  - ϵ is set to the value of the most significant bit of the multiplier's output.
  - The content of register $R2$ is added to ϵ.
- Else
  - The output of the multiplier is wired to the 32-bit adder unchanged.
  - ϵ is set to zero.
  - The content of register $R2$ is added to $\epsilon = 0$, thus is left unchanged.

For a minimal cycle count, the conditional left shift of the multipliers output is again implemented implicitly by wiring the 31 least significant bits of the multipliers output concatenated with a single 0 bit as the LSB to the 32-bit adder. Lines 11 to 13 of the algorithm are implemented the same as with the Comba multiplier.

4.5.3 A Combined Approach

Since the hardware designs of the product-scanning integer squarer and integer multiplier described above are very similar, it makes sense to combine these two designs in order to save on chip area. In fact, the design depicted in 4.6 already is a combined datapath for both Comba multiplication and squaring with reduced cycle count (which is also the reason for the distinct a and b inputs). By ensuring that in case of multiplication ϵ is always set to zero, the first 16-bit adder has no effect. Further, in case of multiplication the output of the multiplier must be wired to the 32-bit adder unchanged. Both of these conditions must already be implemented for the squarer in case the condition $i < j$ is false, therefore only the control logic must be changed for multiplication.

Area Estimation: An estimation of the chip area for the combined Comba multiplier and squarer shown in Figure 4.6 is provided in Table 4.8, again for both a 16-bit and a 32-bit datapath. The multiplier is still the dominating component regarding the area. In this combined design the adders consume more area than the registers. By comparing the results to those of the Comba multiplier it can be seen that the additions needed for the

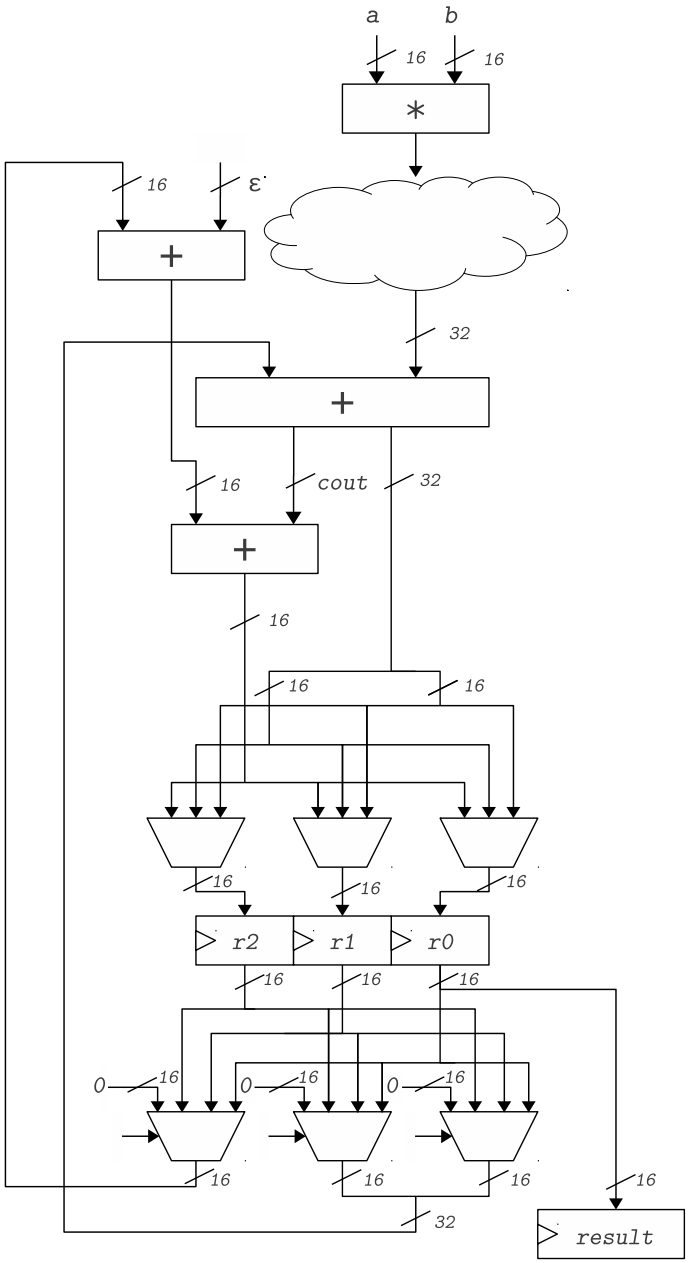

Figure 4.6: The datapath of a 16-bit integer squarer (product scanning form) with implicit conditional multiplication by two. This hardware design can also be used as a combined datapath for both integer squaring (with reduced cycle count) and integer multiplication.

combined circuit only consume ≈ 100GE in case of a 16-bit datapath and ≈ 150 GE in case of a 32-bit datapath, for the benefit of a largely decreased cycle count for squaring operations (see Table 4.9). What can also be seen from that table is, that implementing the Comba multiplier (or the combined multiplier and squarer) with a 32-bit datapath leads to a approximately four times faster (regarding the number of clock cycles) hardware than with a 16-bit datapath.

Table 4.8: Area estimation results of the combined Comba multiplier and squarer depicted in 4.6. Results are provided for both a 16-bit and a 32-bit version.

	Area [GE]	
	(16-bit)	**(32-bit)**
Multiplier	1 616	6 688
Adders	352	704
Multiplexers	463	925
Registers	288	576
$\sum$	2 719	8 893

Table 4.9: Comparison of the cycle counts for 192-bit integer multiplication and squaring with Comba's method, between a 16-bit and a 32-bit datapath implementation as depicted in 4.6.

	# clock cycles	
	(16-bit)	**(32-bit)**
Multiplication	144	36
Squaring	78	21

4.6 Method 1: Skipping Reduction after Addition/Subtraction

4.6.1 Idea

At first, the design concept for the datapath was to use a combined Comba multiplier and squarer as depicted in Figure 4.6 for multiplication and squaring. At this point, the datapath width was considered to be 16-bit in order to achieve a very small elliptic curve cryptography core. Of course, the disadvantage of such a narrow datapath is an increased cycle count. To accomplish a fast completion of the elliptic curve point multiplication nevertheless, several optimization techniques based on the principles described in the previous sections were considered.

Reduction after each field operation is a major factor contributing to the overall number of clock cycles. By using the NIST reduction and its optimized forms as described in 4.3 already reduces the required number of cycles. However, there is still a reduction step necessary after each field operation. Hence, the first idea investigated closer was to skip certain reduction steps. Since by adding two k-bit integers a and b, the resulting integer c can be of width $k+1$ at a maximum, the addition operation seems to be a fitting candidate for such an optimization. The principle would be the following:

- Carry out multi-precision integer additions (and subtractions) as usual.
- Skip the $\pmod p$ operation (the reduction) at the end of the addition (and subtraction).
- When the next field multiplication occurs in the algorithm execution, carry out the integer multiplication followed by the reduction step as usual.
- Thus, after a multiplication, the operand i is reduced to $i < p$ again.

Naturally, the approach could be combined with an incomplete reduction step as described in 4.3, where the integer i is merely reduced to $i < 2^{192}$ (in case of the curve P-192) instead of a complete reduction to $i < p$, thus saving many additional cycles.

4.6.2 Practical Issues

This method would rely on a highly regular point multiplication algorithm in order to become feasible, where after very few additions a multiplication would have to be executed. Otherwise the number of additional bits required in order to hold the increasing intermediate values would become too large. The increase of the intermediate values also leads to additional additions and subtractions in the datapath: While the addition of two 192-bit operands in a 32-bit datapath requires six individual 32-bit additions, by increasing the width of the operands by one bit already leads to seven required additions. The number of individual multiplications also increases for the same reasons. Thus, the benefit of skipping the reduction step after additions and subtractions is mitigated. When a maximum of three addition and subtraction operations before a multiplication are assumed, with the P-192 curve a multiplication of $194 * 194$ bit could occur. The result of such a multiplication could occupy up to $194 * 2 = 388$ bits. Since the NIST reduction algorithm as shown in Section 4.3 is only defined for integers $i < p^2$, the approach would have to be extended which would again increase the cycle count. This method therefore was abandoned quickly for a very similar approach described in the next section (4.7).

4.7 Method 2: Reduced NIST Reduction after Addition/-Subtraction

4.7.1 Idea

The approach described in this section is based on the idea illustrated in section 4.6, while trying to eliminate its disadvantages. Again, a full reduction is only executed after an integer multiplication. However, instead of completely skipping the reduction steps required by the field addition and field subtraction, this time an approach is chosen where a reduced reduction step is executed after those operations. Since k-bit addition of two integers leads to a $k+1$ bit result at a maximum, no full reduction step is needed. Instead, as described in Section 4.6, only one multi-precision addition has to be carried out in order to reduce the result of the addition or subtraction to $i < 2^{192}$ (when using the curve P-192), when both operands are smaller than p. Without this optimization, three multi-precision additions would have to be performed for a reduction to $i < 2^{193}$, with either one additional reduced reduction step or (conditional) subtractions of the modulus in order to reduce i to $i < 2^{192}$. Therefore by this approach, four multi-precision additions can be saved at each field addition or subtraction. Because of the reduced instruction step, no additional bits are wasted as by the approach described in section 4.6, thus the multiplication operands always have a width of 192 bit at a maximum. This in turn leads to a maximum multiplication result of 384 bit which can easily be reduced by the standard NIST fast reduction formulae or preferably the multi-iteration NIST reduction as described in 4.6, in order to save further multi-precision subtractions. In a nutshell, this approach can be described as follows:

- After integer addition and subtraction a reduced reduction step is executed, which reduces the integer i to $i < 2^{192}$.

Table 4.10: Results regarding the number of clock cycles necessary for the reduced NIST reduction after addition and multiplication approach.

		# clock cycles			
	width	**ADD**	**MUL**	**SQU**	**Montgomery Ladder**
Comba Multiplier	16	24	192	192	656640
	32	12	60	60	232560
Comba Multiplier/ Squarer	16	24	192	126	581400
	32	12	60	45	215460

- After multiplication, a multi-iteration reduction is executed, which reduces the multiplication result to $i < 2^{192}$.
- Only at the end of the elliptic curve point multiplication i is reduced to $i < p$ by one conditional subtraction.

4.7.2 Estimation Results

The chip area of the implementation of this method is depending on whether a Comba Multiplier or a combined Comba multiplier and squarer is used. For the reason of getting a faster crypto core, the combined version would have been used, leading to an estimated area of 2719 GE for the 16-bit datapath implementation or 8893 GE for the 32-bit datapath implementation. The various exploitations of the NIST reduction would not lead to any additional area requirements. Results regarding the number of clock cycles necessary for this method can be found in Table 4.10 for both a 16-bit and a 32-bit datapath. Computational costs for addition (ADD), multiplication (MUL) and squaring (SQU) include costs for reduction as described in this section. Computational costs of the point multiplication formulae used for the estimation are given in Table 4.3. The estimates do not include those operations that are listed under *additional costs* in the table.

4.7.3 Practical Issues

As hinted to above, only one reduced reduction step is necessary after an addition (or subtraction) in order to reduce the result to $i < 2^{192}$ when both operands are smaller than p. Unfortunately, after this first iteration, since no full reduction is performed by this method, the operands are not not smaller than p any more in general. This leads to another reduced reduction step or a conditional subtraction necessary in order to again get an integer $i < 2^{192}$. Because of our requirement of constant execution time to prevent side channel attacks, the conditional subtraction is no option, increasing the cycle count in a real-world scenario (two instead of one reduced reduction steps after each addition and subtraction are necessary). Thus, this idea was abandoned for the method described in Section 4.8, which is based on the same idea but tries to compensate for the second reduced NIST reduction iteration.

4.8 Method 3: Storing the Carry for Implicit Reduction

4.8.1 Idea

This method is again based on the same principle as the preceding method. The approach described here tries to get rid of the second reduction step which is necessary because of the possibility of additions and subtractions with operands greater than p. Same as above, one reduced NIST reduction is performed after additions and subtractions instead of the complete reduction. Again, due to possible consecutive additions or subtractions and the consequential possible operand size greater than p, a second reduced reduction iteration or a conditional subtraction (which is off-limits) would be necessary. At this point, the new idea was to store the carry bit instead, that is, to store the value of the 193th bit. At the next operation the carry would have to be taken care of. The straight forward approach would be to simply add the carry at bits 0 and 64 to the operand, which would be exactly the same as the second reduced NIST reduction operation: Due to a possible carry propagation through the whole word, a full 192- bit addition would have to be computed leading to the same cycle count as with the method above. Therefore, here the second idea was to *implicitly* perform the addition (and thus the second reduction iteration) by deploying a second adder in the datapath. For this approach to work, a relationship between an operand and its carry value would have to be established. By this relationship, at the next operation a decision could have to be made on whether the reduction word or zero would have to be added at the new additional adder in the datapath. The output of the adder would then always be smaller than 2^{192}.

4.8.2 Estimation Results

For the implicit reduction in the datapath, an additional adder is required in the datapath. Since there is the possibility that both operands of an addition have their stored carry bit set to 1, both operands require such an adder, leading to additional costs regarding the chip area of $2 * 176\text{GE} = 375\text{GE}$ in case of a 32-bit datapath and $2 * 88\text{GE} = 176\text{GE}$ in case of a 16-bit datapath. Additionally, the carry bit must be saved for each operand. The used point multiplication formulae uses eight so-called *field registers*, which means 8-bit of additional storage (estimated to require $\approx 48\text{GE}$ of chip area) would be necessary for the carry bits. The number of clock cycles can be assumed to be the same as in Table 4.10, since this method only compensates for the drawbacks of method 2.

4.8.3 Practical Issues

Since the implicit reduction occurs *before* additions and subtractions, when an addition or subtraction is succeeded by a multiplication, there is a possibility that both operands are greater than 2^{192}. In this case, instead of a $k * k$-bit multiplication, a $(k+1) * (k+1)$-bit multiplication would have to be performed, leading to a multiplication result greater than 2^{384}. Hence, there would have to be a bigger multiplier, bigger accumulator register, as well as a more complicated reduction algorithm requiring further additions. Another solution for this problem would be to also use implicit reduction before the multiplication. This would lead to another problem: Let's denote $a[n]$ the n-th t-bit chunk of the 192-bit operand a, where $t = 32$ and $0 \leq n < 6$ in case of a 32-bit datapath or $t = 16$ and $0 \leq n < 12$ in case of a 16-bit datapath. With the Comba multiplication algorithm, n is not monotonically increasing, what complicates carry propagation. There would have to

be an additional carry storage for each t-bit word of both operands (12 bit $\approx$ 72GE in case of a 32-bit datapath, 24-bit $\approx$ 144GE in case of a 16-bit datapath).

4.9 Method 4: 16-bit Multiplier and 32-bit Adder

4.9.1 Idea

Adders are relatively cheap regarding their chip area when compared to array multipliers (for reasonable bit lengths). A 16-bit array multiplier needs approximately 1 616 GE, a 16-bit adder 88 GE and a 32-bit adder 176 GE. By using a 32-bit adder instead of a 16-bit one, each multi-precision addition would only require half the clock cycles, speeding up the point multiplication: When going by the computational costs as listed in Table 4.3, $190*32+35 = 6\,115$ 192-bit add operations have to be performed. With a 16-bit datapath, these would be computed within 73 380 clock cycles (without reduction). With a 32-bit datapath, the add operations would be computed within 36690 clock cycles (again without reduction). By these estimates, the advantage of using a 32-bit datapath for additions can already be seen. Further, after each operation a reduction $\pmod p$ has to be performed which is also computed using the adder. Even if we use the reduced reduction techniques as described above, at least one reduced reduction step has to be performed after each addition. So, at a minimum, a 192-bit field addition (including the reduction) with a 16-bit datapath is completed within 146 760 clock cycles. The same operation would be completed within 73380 clock cycle with a 32-bit datapath at a minimum. The full reduction after each multiplication also uses the adder in the datapath, so even the cycle count of field multiplications would be decreased.

4.9.2 Estimation Results

When using implicit reduction, of course the two additional adders in the datapath would also have to be of 32-bit width. Thus, a minimum of three 32-bit adders would be needed instead of three 16-bit adders, which means that the chip area would be increased at least by 264 GE. This leads to a great trade off between chip area and cycle count, since the overall cycle count of a point multiplication can be largely decreased by using this technique.

4.9.3 Practical Issues

This idea actually led to satisfactory results but it is only one possible optimization and should be combined with other techniques. In fact, this idea was considered and would have been implemented together with all preceding methods and was only abandoned due to the final design method which is based on a 192-bit datapath.

4.10 Method 5: A Combination of Comba's Method and a Karatsuba-Ofman Multiplier

4.10.1 Idea

In Section 4.4.5, 32-bit Karatsuba-Ofman multipliers with various recursion depths were compared to a 32-bit array multiplier. The result of this comparison was that such a

Karatsuba-Ofman design could reduce the area required by a 32-bit multiplier (with respect to a 32-bit array multiplier) while still only needing one clock cycle for completion of the multiplication operation. At the end of Section 4.4.5 it was hinted that such a Karatsuba-Ofman multiplier could be used in combination with Comba's method for multi-precision multiplication. This idea is further investigated in this section. Karatsuba-Ofman designs become only feasible for greater bit-widths of their operands. This is due to the fact that only then the area costs for addition (and subtraction) are small enough in comparison to those of multiplication, because of the steep growth in area of array multipliers with the bit width. Therefore in this section instead of a 16-bit datapath a 32-bit datapath is considered.

From Table 4.6 it can be seen that the two recursion level approach offers the best area result. Thus, a 192-bit Comba multiplier with a 32-bit datapath is deployed for 192-bit integer multiplication, where instead of the array multiplier, a two-level 32-bit Karatsuba-Ofman multiplier is implemented at its core.

4.10.2 Estimation Results

Area The chip area occupied by a 192-bit Comba multiplier with 32-bit datapath combined with a two-level 32-bit Karatsuba-Ofman multiplier consist of:

1. 5 364 GE for the Karatsuba-Ofman multiplier
2. 528 GE for the adders in the Comba datapath
3. 925 GE for the multiplexers in the Comba datapath and
4. 576 GE for the registers in the Comba datapath

This leads to an estimated area of 7 393 GE for the whole multiplication circuit. For comparison, the same multiplication circuit with a 32-bit array multiplier instead of the Karatsuba-Ofman multiplier was estimated to require 8 717 GE (see Table 4.7). However, this estimation does not include area costs for the control logic. Further, the estimation assumes equal bit widths for all multipliers of a certain recursion depth of the Karatsuba-Ofman multiplier. As mentioned in Section 4.4.5, the actual hardware implementation would have to use one wider multiplier at each recursion depth because of carry generation, as well as wider subtractors. This reduces the benefit regarding required area with respect to an array multiplier. Of course, the adders and subtractors used for the Karatsuba-Ofman multiplier, as well as the adders used for the Comba multiplier could also be used for multi-precision addition as well as for reduction.

Clock Cycles The Karatsuba-Ofman multiplier used in this design completes a 32-bit multiplication within one clock cycle. The used Comba multiplier for multiplication of 192-bit words by a 32-bit datapath executes one multiplication operation per clock cycle and needs 36 clock cycles. So, as a whole this design finishes one 192-bit multiplication within $36 * 1 = 36$ clock cycles.

4.10.3 Practical Issues

The method described in this section improves the area of a 32-bit datapath implementation of the P-192 elliptic curve cryptography core, in comparison to a Comba + array

multiplier implementation. However, the estimation above does not include some factors like the requirement of somewhat wider multipliers and subtractors thus mitigating the already slight area improvement. Further, the number of clock cycles required for a 192-bit multiplication is not improved by this approach. Hence, further designs were explored and the method was abandoned for the final design described in Chapter 5.

4.11 Method 6: Utilizing a Karatsuba-Ofman Multiplier with Serial Multiplication

4.11.1 Idea

The combination of a Comba multiplier and a Karatsuba multiplier described in the previous section resulted in improved estimation results regarding the chip area when compared to a Comba multiplier with an array multiplier at its core. Hence, a similar approach was considered which is described in this section. The idea was to again use a Comba multiplier together with a Karatsuba-Ofman multiplier, but this time a design different from the one shown in Figure 4.4 should be used. The principle structure of the new design is the same but instead of the three multipliers only one should be used. The same multiplier calculates all three intermediate results of a recursion level, one per cycle, and store them in a 32-bit register each. The 32-bit registers are then wired to the two subtractors and the adders for recombination of the intermediate results the same way as the three multipliers in Figure 4.4. The approach would need three times the cycle count but would have a reduced area: Instead of the $3 * 1\,616\text{GE} = 4\,848\text{GE}$ for the three 16-bit array multipliers, only one 16 bit array multiplier and three 32-bit registers (occupying $\approx 3 * 32 * 6\text{GE} = 576\text{GE}$) would be used.

4.11.2 Estimation Results

The total area of the Karatsuba-Ofman multiplier used by this approach would be 2964 GE. For comparison, the area of the one level design with three multipliers was estimated to be $\approx 5\,640$GE. The 192-bit multiplication would be finished after 108 clock cycles. Table 4.11 gives a comparison between a Comba multiplier with 16 bit datapath and 32-bit datapath, both using an array multiplier and a Comba multiplier with 32-bit datapath which uses the Karatsuba-Ofman multiplier described in this section regarding cycle count, as well as a comparison between a 16-bit array multiplier, a 32-bit array multiplier and a 32-bit Karatsuba-Ofman multiplier as described in this section, regarding the area. It can be seen that by deploying this method, we trade approximately half of the area of a 32-bit array multiplier for three times the clock cycles, leaving this approach a bit faster than the 16-bit array multiplier version but also much bigger. What is also worth of note is, that the estimation for the Karatsuba-Ofman circuit was again calculated for the simplified circuit, that is, carry propagation was not considered. Thus, the real implementation would be somewhat larger still.

4.11.3 Practical Issues

Although this design is a bit faster than a Comba multiplier with 16-bit array multiplication it is close to twice as big, and although this design is much smaller than a Comba multiplier with 32-bit array multiplication, it needs thrice the number of cycles in order to

Table 4.11: Comparison between a 32-bit Karatsuba-Ofman multiplier with serial multiplication and 16-bit, as well as 32-bit array multipliers regarding area and cycle count when applied with Comba's method for 192-bit multiplication.

16-bit array multiplier		**32-bit Karatsuba**		**32-bit array multiplier**	
1616	<	2964	<	6688	**GE**
144	>	108	>	36	**# cycles**

complete one 192-bit multiplication. This leaves it at a point where it does not offer a reasonable area versus speed trade off for deployment in the P-192 elliptic curve cryptography core.

4.12 Method 7: Various Shift-Add Multiplier Designs

4.12.1 Idea

Since the approaches regarding an efficient datapath design described in the sections above, which were all based on a Comba multiplier, did not deliver in satisfactory results, a completely different approach was also investigated. In this section, various multiplier architectures based on a shift-add multiplier are explored. The basic principle of the shift-add algorithm as well as a simplified block diagram of a possible hardware implementation were already shown in Section 4.4.3. All shift-add designs described in this Section are based on the shift right version.

Wide shift-add Multiplication

The most straight-forward usage of a shift-add multiplier design in the P-192 elliptic curve cryptography core would be to implement it as a 192-bit shift-add circuit. The area requirements for such a design would be 1056 GE for the 192-bit adder, 2304 GE for the 384-bit accumulator register as well as 240 GE for the 192-bit multiplexer implemented as and-gate, thus in sum 3600 GE. The multiplication would be completed within 192 clock cycles. When compared to a 16-bit Comba multiplier (with an array multiplier at its core), which completes multiplication within 36 clock cycles, this shift-add multiplier would be much slower (≈ 5.2 times the clock cycles required by the Comba multiplier). Unfortunately this shift-add multiplier is also bigger than the 16-bit Comba multiplier, therefore a straight-forward implementation of this approach seems not suitable.

Another idea was to design a highly parallelized shift-add multiplier. The 192-bit operand words would have been split on 32-bit boundaries, leading to $192/32 = 6$ words for each operand. The shift-add multipliers would have also been 32-bit wide, thus 36 parallel shift-add circuits and 36 32-bit adders for recombination of the partial products would have been necessary. Naturally, the area of such a multiplier would have gotten huge and it would still have needed almost three times as much clock cycles for completion of the multiplication as the 16-bit Comba multiplier. So, this idea was abandoned quickly too.

A Comba Multiplier with 32-bit Shift-Add Multiplication

Since shift-add multipliers generally are small in comparison to array multipliers (see Table 4.5), a 32-bit datapath based on the shift and add multiplication algorithm was considered. In order to enable 192-bit integer multiplication, multi precision arithmetic should be used. The idea was to deploy a Comba multiplier as depicted in Figure 4.5, but instead of the array multiplier at its core, the 32-bit shift-add multiplier should be used. The estimated area for such a design looks very promising: The 32-bit shift-add circuit occupies approximately 600 GE of chip area. For the Comba multiplier, the additional area consists of

- 352 GE for the 64-bit Adder
- 176 GE for the 32-bit Adder
- 925 GE for the multiplexers
- 576 GE for the three 32-bit accumulator registers

The area of these components sums up to 2 629 GE which, in comparison to the 8717 GE used for a 32-bit Comba multiplier with an array multiplier at its core is a major improvement. The 16-bit Comba multiplier with an array multiplier at its core occupies an area of 2631 GE. Unfortunately the number of cycles required for completion of the multiplication is very large for this concept. The 32-bit shift-add circuit needs 32 cycles to complete one multiplication and the 32-bit Comba multiplier needs 36 cycles when multiplying two 192-bit operands, leading to 1152 cycles as a whole. This is 32-times the cycle count of the array-multiplier version and 8 times the cycle count of the 16-bit array multiplier version, which occupies approximately the same area as the 32-bit shift-add multiplier version.

A Parallel Approach Because the area results of the preceding approach looked promising but with the drawback of a large cycle count, a parallel approach was explored in order to speed things up. The design principle is the same as above, but instead of the 32-bit shift-add multiplier 16-bit shift-add multipliers should be deployed. The 32-bit shift-add multiplier can be substituted by four 16-bit shift-add multipliers, thus occupying two times the area ($300\text{GE} * 4 = 1\,200\text{GE}$ as opposed to 600GE). The four parallel circuits would compute four 32-bit intermediate results within 16 clock cycles. To recombine these four intermediate values, three 64-bit additions (again with the shift-add principle) would have to be computed, leading to an additional time consumption of tree clock cycles. No additional chip area would be necessary (except for additional wiring and multiplexers) for the recombination, since the 64-bit adder of the Comba datapath as well as the output register or two of the 32-bit registers of the shift-add circuits could be used for this purpose. Also, no additional input registers for the shift-add circuits would be necessary since their 16-bit inputs could just be wired to the lower or upper 16-bit word of the 32-bit input register. Thus, the area estimation of this approach would be:

- Area occupied by the four 16-bit shift-add circuits:
  - $4 * 300\text{GE} = 1200$ GE
- Additional area occupied by the Comba multiplier:

- 352 GE for the 64-bit Adder
- 176 GE for the 32-bit Adder
- 925 GE for the multiplexers
- 576 GE for the three 32-bit accumulator registers

The area of these components sums up to 3229 GE which is bigger than the area needs of a 16-bit Comba multiplier with an array multiplier at its core but still much smaller than the 32-bit version (which is more than twice as big). The cycle count could also be improved by this concept in comparison to the previous approach, since instead of 32 cycles for one 32-bit addition this parallelized approach only needs 19 cycles (16 cycles for the parallel execution of the four intermediate values plus three additional cycles for the recombination). Still, this is 19 times the cycle count of the 32-bit Comba multiplier with an array multiplier and almost five times the cycle count of the 16 bit version. In summary, the 16-bit Comba multiplier with an array multiplier is smaller **and** faster than this parallelized shift-add approach.

Another Parallel Approach Because of the better although unsatisfactory result of the previous approach a second parallel design was considered. This time massive parallelization should be applied. Again, the same Comba multiplier as in the previous sections is used, but this time with 16 parallel 8-bit shift-add circuits. These would produce 16 intermediate results within 8 clock cycles which would have to be recombined again. If a straight forward method like in the first parallel approach would be used (another shift-add like circuit) the calculation of the 64-bit result from the 16 16-bit intermediate results would take up 16 additional clock cycles. So, the recombination would have been parallelized too: The intermediate results would have been split into four groups of 4 x 16 bit intermediate results. The four groups would be summed up in parallel and another four cycles would be used to sum up the resulting four intermediate results. In sum one 192-bit multiplication would be completed within 16 cycles by this approach, which is a small improvement over the previous approach, but comes with the drawback of an increased area: 2400 GE would have to be used for the 16 8-bit shift-add circuits, which is already more than a 16-bit array multiplier would occupy. Thus, this approach had to be abandoned too.

Chapter 5

An ECC Core for Use in RFID-based IoT

In this chapter, we use the knowledge gained from the design exploration to design the final ECC core. In Section 5.1, the basic idea behind the final datapath design is presented. In Sections 5.2 and 5.3, we introduce several improvements to this idea. Section 5.4 details the concept of high-radix multiplication and describes remaining components of the final datapath, i.e., addition, subtraction, modular reduction, and field inversion. In Section 5.5, point multiplication based on the Montgomery ladder with (X,Y)-only co-Z doubling addition algorithm is described. Finally, in Section 5.6, we present the synthesis results of the elliptic-curve cryptography core, as well as a comparison with related work.

5.1 The Basic Idea Behind the Final Datapath Design

In Chapter 4 several shift-add multiplicatoin circuits were investigated. The 192-bit shift-add multiplier was estimated to require approximately 3 600 GE (without operand registers) and complete a 192-bit multiplication within 192 clock cycles. Assuming a 192-bit input register **A** and a small (e.g., 8-bit) input register **B**, where **A** contains the multiplicand and **B** contains part of the multiplier, the area would be $\approx 3\,600\,\text{GE} + 192 * 6\,\text{GE} + 8 * 6\,\text{GE} = 4\,800\,\text{GE}$. In each clock cycle, one bit of **B** would determine whether the 192-bit in **A** would be added to the (shifted) accumulator register. A straight implementation of this method would lead to a multiplier that is larger and slower than a Comba multiplier with 16-bit datapath but would need a much simpler control logic. By introducing another 192-bit adder, two shift-add iterations could be computed consecutively in each clock cycle: One bit of the multiplier register **B** would determine the input of the first adder (either zero or the the multiplicand in **A**, another bit of **B** would determine the input of the second adder (again either zero or the content of **A**), leading to a major reduction of the cycle count (96 clock cycles instead of 192). The additional area requirements would be $192 * 5.5\text{GE} = 1\,056\text{GE}$ for the second adder. This approach could be repeated an arbitrary number of times, trading chip area for clock cycles with each additional adder deployed. Since hereby a very fast multiplication circuit with very large chip area requirements could be constructed, a reduction of the area was aspired. By combining the approach with implicit reduction, a narrower accumulator register could be used, but at the cost of additional adders. Implicit reduction would also make the multiplier faster, since the NIST reduction required at the and of the field multiplication would require less

clock cycles. Experimenting with these parameters led to the first design concept of the final datapath design, described in Section 5.2.

5.2 Refining the Datapath Design

Shift-add multiplication can be either performed by repeatedly adding by the (unchanged) multiplicand to the shifted contents of an accumulator or by adding a shifted version of the multiplicand to the contents of an accumulator. For the first design concept of the final datapath the latter approach was chosen. The design is depicted in Figure 5.1. It illustrates the concept idea described above for the most conservative approach regarding chip area, where only one bit of the multiplier is taken at a time. The input stage would

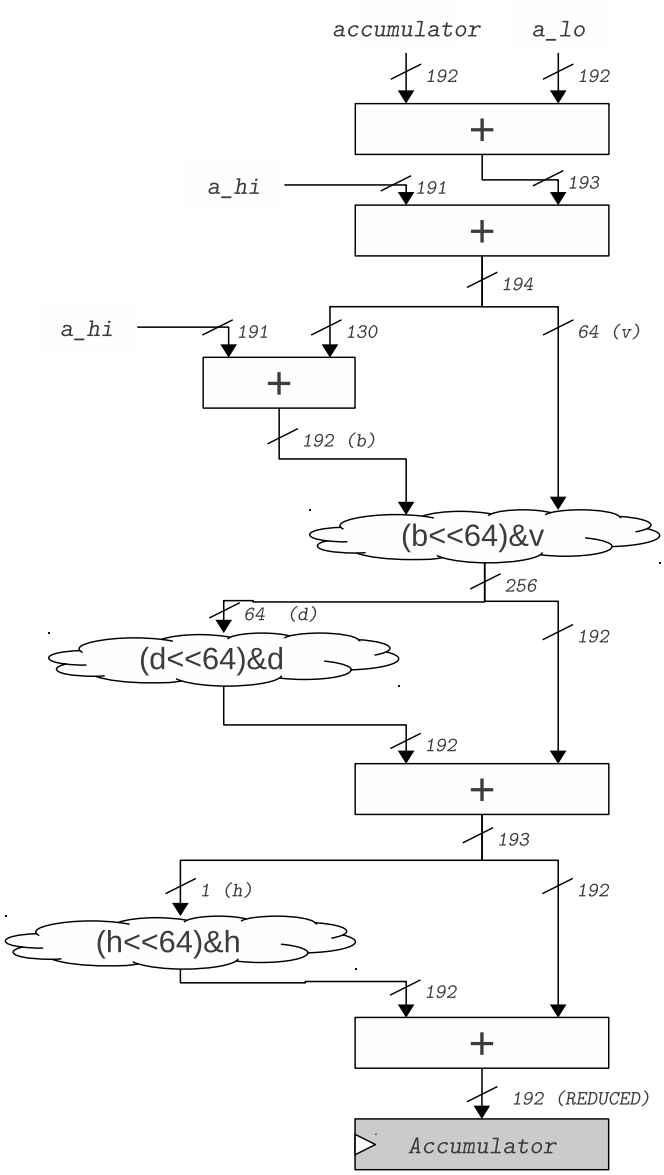

Figure 5.1: Simplified block diagram of a shift-add multiplier with implicit NIST reduction capabilities.

actually consist of one 384-bit adder as opposed to the two 192-bit respectively 193-bit adders as seen in the Figure. However, since an implicit reduction approach was taken, the input state works as follows: At first, the content of the accumulator register must be set to zero. The output of the accumulator is wired to one of the inputs of shift-add circuit, the second input a either contains the (shifted) multiplicand or zero, depending on the multiplier. For the purpose of the implicit reduction, a was divided into its lower 192-bits (denoted *a_lo* in the Figure) and its higher 192-bit (denoted *a_hi* in the Figure). The upper word *a_hi* is directly added to the lower word *a_lo*, which is already part of the implicit reduction: As Figure 5.2 shows, by reordering the chunks A,B and C it can be seen that the first reduction step actually consists of the addition of the higher 192-bit of the operand to the lower 192-bit. Hence, by exploiting this reordering, one 192-bit adder can be saved.

The output of the input stage is then reduced further. A shifted version of *a_hi* is added, which is realized in hardware by the utilization of another adder. In the figure this is denoted by the addition of the upper 130 bit of the result of the input stage and *a_hi*. The result of this addition is then concatenated with the lower 64 bit of the result of the input stage. Similarly, the 256-bit result of this concatenation is reduced by concatenating its upper 64 bit with their shifted version and adding the concatenation to the lower 192-bit. At this point a reduction as depicted in 5.2 has been performed. However, the result of this reduction can have a width of 193 bit, therefore another reduction step similar to the reduced reduction as described in 4.3 is computed. In a nutshell, this approach needs

- 3 168 GE for the three 192-bit adders
- 1 062 GE for the 193-bit adder
- 1 051 GE for the 191-bit adder
- 2 352 GE for the accumulator and operand registers

which leads to an area of approximately 7633 GE. This is a relatively large area consumption considering a cycle count of 192 for the multiplication, but the reduction is performed implicitly (leading to an overall improved cycle count) and the control logic for such a circuit is very simple. Expanding this hardware design to a version that computes two consecutive iterations of the shift-add algorithm concurrently (as suggested above) leads to an increased area but half the cycle count. Figure 5.3 shows the input stage of such an architecture (the overall design stays the same). Hereby, *a_lo1* and *a_hi1* denote the lower 192 bit respectively the upper 192-bit of the first shifted version of the multiplicand and *a_lo2* and *a_hi2* denote the lower 192 bit respectively the upper 192-bit of the second shifted version of the multiplicand. The estimated area of the adders required for this approach is increased to:

- three 192-bit adders
- one 190-bit adder
- one 191-bit adder
- one 193-bit adder
- one 194-bit adder

- one 195-bit adder

Together with the registers, this leads to an area of approximately 10618 GE. The area would get even bigger when the process would be sped up further by computing three or more iterations of the shift-add algorithm concurrently. What also has to be considered is, that a design as seen in Figures 5.1 and 5.3, with many serial ripple-carry adders of large bit width, leads to large carry propagation delays. Hence, the maximum possible clock frequency is very limited with such a design.

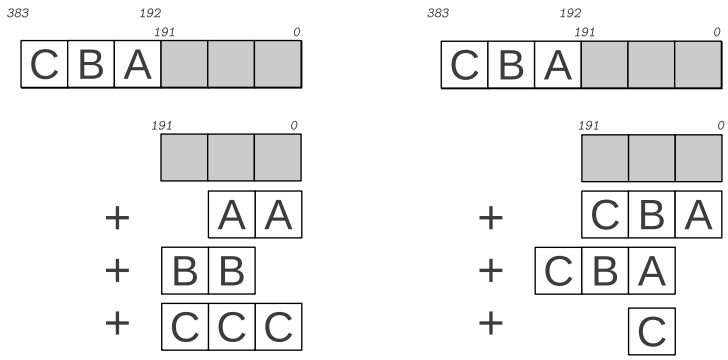

Figure 5.2: Principle of the fast reduction modulo p_{192}. The left version is the same as described in Section 4.3, the right version shows a reordered version leading to the same result, that is beneficial for the 192-bit shift-add multiplication with implicit reduction as described in Section 5.2.

5.3 Using Better Suited Adder Types

In order to reduce carry propagation delay, adder architectures with reduced carry propagation were considered. Because of the tree-like structure of the shift-add multiplier with implicit reduction, utilization of carry-save adders (CSA) seemed to be beneficial. A carry-save adder is composed of full adders, but instead of connecting them in series (wiring the carry out of a full-adder cell to the carry in of the next full-adder cell) the full adders are ordered in parallel. Hence, instead of adding two n-bit words to a (n+1)-bit word, a carry save adder *compresses* three n-bit words to two n-bit words (the carry output of the full adder cell is treated like a regular output). Carry propagation delay through such a carry-save adder is negligible, since independently of the width of the adder, a carry only travels though one full-adder cell.

5.4 The Final Datapath Design

5.4.1 High-Radix Multiplication

The original shift-add multiplier as depicted in 4.3 computes one partial product at each clock cycle. Another way to see this is, that the circuit computes one digit in each clock cycle. If, instead of a radix-2 representation we use a higher representation radix (e.g., radix-4, radix-8, or radix-16), an algorithm that computes one digit at a time needs fewer clock cycles [26]. Now the idea was to utilize this principle by using a radix-16

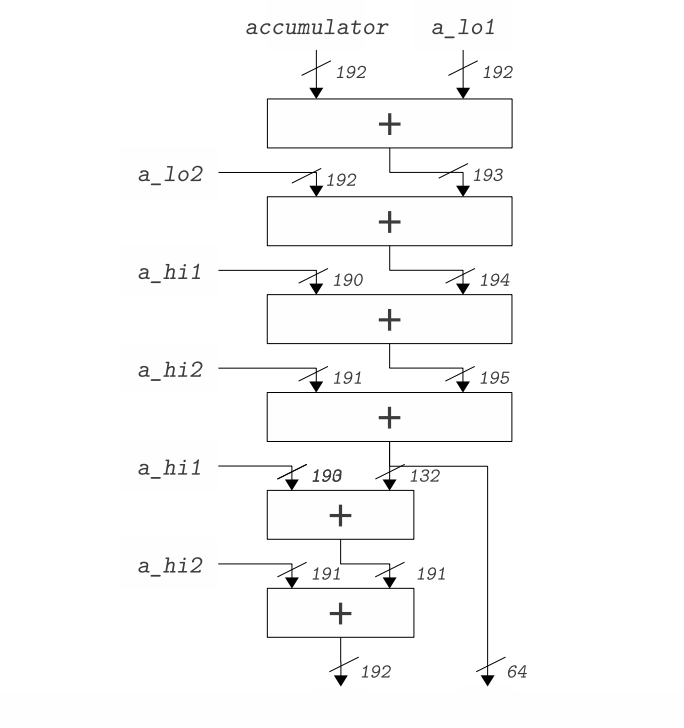

Figure 5.3: Input stage of a 192-bit shift-add circuit with implicit reduction and calculation of two shift-add iterations per clock cycle.

representation of the multiplier **B**, which means that at each clock cycle four bits of the multiplier would be processed simultaneously. For a reduced carry propagation delay, a small adder tree could be used, consisting of two carry-save adders and one ripple-carry adder in order to get the resulting sum from the compressed output of the second carry-save adder. This architecture is depicted in Figure 5.4.

The multiplier register is shown as a 192-bit register in the Figure but could be implemented as a register of arbitrary width ≥ 4. At each clock cycle, the content of the multiplier register is right shifted by four bits. These four bits are wired to the select inputs of four multiplexers. The multiplexer for the least significant bit either switches the multiplicand **A** or zero, the multiplexer for the second-least significant bit either switches $2 * \mathbf{A}$ or zero, and so forth. A first carry-save adder reduces the outputs of the three multiplexers for the lower significant bits, a second consecutive carry-save adder reduces the outputs of the multiplexer for the most significant bit and the first carry save adder, and a ripple carry adder then computes the sum, which is represented by up to 196 bits. The sum is then added to the content of an accumulator register, which on the other hand is shifted right by one bit in each clock cycle. Hence, an additional adder to those seen in Figure 5.4 is required.

For the actual hardware implementation of the input stage, further optimizations regarding the chip area can be made. First, since all four multiplexers either switch a multiple of **A** or zero, ordinary AND gates can be used instead. Second, the multiples of **A**, namely $2 * \mathbf{A}$, $4 * \mathbf{A}$, and $8 * \mathbf{A}$, actually are only shifted versions of **A**, with their one, two, or three least significant bits always set to zero. Hence, all four multiplexers only have to be of width 192-bit since the remaining bits are zero anyways. Third, instead of a

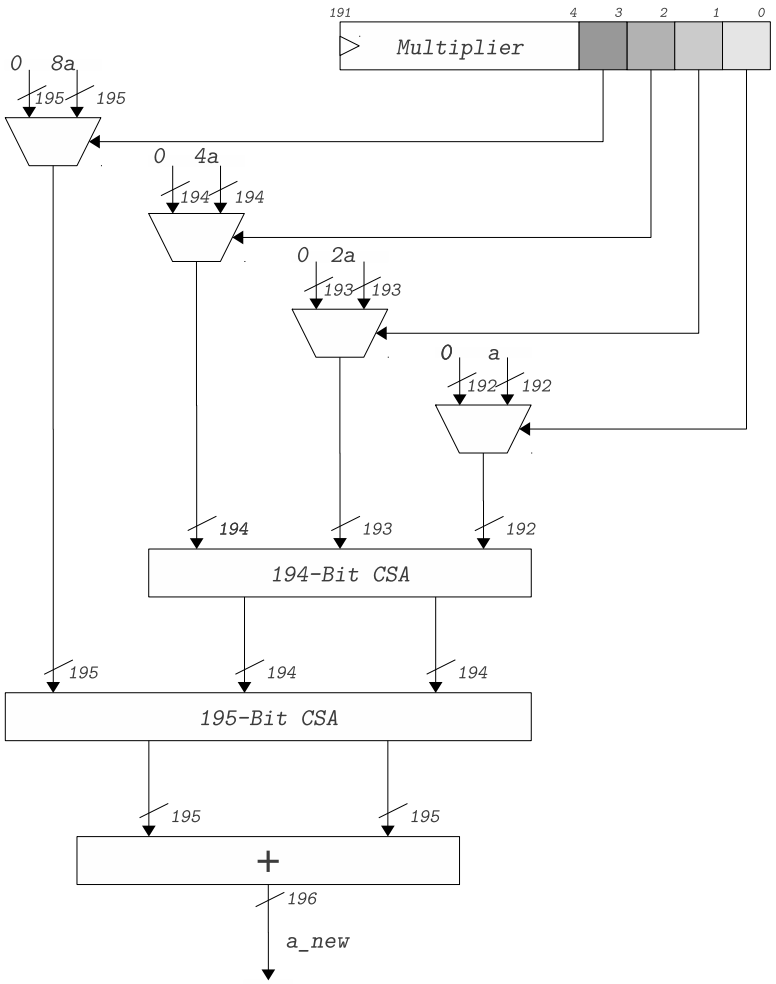

Figure 5.4: Block diagram of the input stage of a radix-16 shift-add multiplier with implicit reduction.

192-bit register for the multiplier, the shortest reasonable register should be used, which a four-bit register in case of a radix-16 shift-add multiplier. For the implicit reduction, at first a circuit like in the block diagram shown in Figure 5.1 was considered.

Besides the radix-16 version of shift-add multiplier's the input stage, other high-radix versions were considered. Their respective estimation results (including the implicit reduction) are given in Table 5.2. The upper half of the table shows the results of the area estimation including implicit reduction, the lower half gives estimates for an implementation without implicit reduction. Besides the components given in the table (adders, multiplexers and the accumulator register) the sum $\sum$ also includes the chip area needed for the 192-bit operand register **A** and the operand register **B** which is assumed to have a width of 8 bit. In case of the radix-16, radix-64, and radix 256 implementations, instead of one 2:1 multiplexer of width 192-bit for each of the concurrently evaluated bits of **B**, AND gates are used. In case of the radix-4 implementation, a 4:1 multiplexer is used, switching either zero, $\mathbf{1} * \mathbf{A}$, $\mathbf{2} * \mathbf{A}$, $\mathbf{3} * \mathbf{A}$, or $mathbf4 * A$, where $\mathbf{2} * \mathbf{A}$ and $\mathbf{4} * \mathbf{A}$ are again just

Table 5.1: Comparison between various high-radix shift-add multiplier designs regarding the number of required clock cycles for a elliptic curve point multiplication. The estimates do not include the *additional costs* in Table 4.3 and loading of the registers.

With implicit reduction				
radix:	4	16	64	256
width:	2	4	6	8
# cycles:	270180	142500	99940	78660
Without implicit reduction				
radix:	4	16	64	256
width:	2	4	6	8
# cycles:	272840	142500	99940	78660

bit-shifted versions of the contents of A and $\mathbf{3} * \mathbf{A}$ is composed by deployment of one additional adder which adds $\mathbf{1} * \mathbf{A}$ and $\mathbf{2} * \mathbf{A}$. Regarding the estimation results given in the table it can be seen that the area of the high-radix shift-add multiplier implementations in general are relatively big, especially those with implicit reduction capabilities. When the results of the shift-add multiplier are not implicitly reduced, an accumulator register of double the width is required in order to hold the whole $2 * k$-bit result of the $k * k$-bit multiplication. However, implicit reduction requires the deployment of additional adders in the datapath. Hence, the additional area consumption of the bigger accumulator register is mitigated. For example, if we look at the estimation results of the radix-16 shift-add multiplier without implicit reduction, the area needed by the adders is approximately 4279 GE, which is only nearly half the area than when using the circuit including implicit reduction and even lessen than the radix-4 version including reduction. On the other hand, the additional area consumed by the higher-width accumulator is only 1158 GE. From the table, it can also be seen that a radix-16 shift-add circuit without implicit reduction as a whole consumes less chip area than a radix-4 multiplier with implicit reduction capabilities. Thus, the advantage of implicit reduction regarding the cycle count is lessened. When we apply the same comparison for higher-radix circuits, for example comparing the radix-64 multiplier including implicit reduction capabilities with the radix-16 version without, it can be seen that the advantage regarding the area is moderately smaller (the high-radix version needs a bit more area than the low-radix version with implicit reduction). Therefore the radix-16 shift-add multiplier offers the best trade off as we also see by comparison of the cycle count.

In Table 5.1, the estimated number of clock cycles necessary for an elliptic-curve point multiplication is shown for diverse high-radix shift-add multipliers. The estimates do not include the *additional costs* in Table 4.3. Furthermore, the estimates do not include loading of the operand registers and writing back the results into the RAM, both of which can take a significant amount of time. Considering a 32-bit RAM interface, at least six clock cycles are necessary in order to load a 192-bit register. Nevertheless, the table gives insight into the relation between the representation radix and the cycle count. Again, the most significant improvement can be made by going from a radix-4 to a radix-16 representation.

When only the number of clock cycles needed for completion of the multiplication is considered (which means reduction is left out for the moment), due to the concurrent processing of multiple bits of the multiplier $\mathbf{B}$, the cycle count goes down drastically: The standard 192-bit shift-add multiplier completes within 192 clock cycles. The radix-

Table 5.2: Comparison between various high-radix shift-add multiplier designs regarding their chip area. Both implementations with and without implicit reduction were estimated. The sum $\sum$ includes the area used by the operand registers.

			Adders		**MUX**		**ACCU**		$\sum$
	radix	width	#	area [GE]	#	area [GE]	width	area [GE]	area [GE]
incl.	4	2	5	5 302	1	1 062	193	1 158	8 867
red.	16	4	7	7 458	4	960	193	1 158	10 776
	64	6	9	9 675	6	1 440	193	1 158	13 473
	256	8	11	11 886	8	1 920	193	1 158	16 164
excl.	4	2	2	2 123	1	1 062	384	2 304	6 834
red.	16	4	4	4 279	4	960	384	2 304	8 743
	64	6	6	6 495	6	1 440	384	2 304	11 440
	256	8	8	8 707	8	1 920	384	2 304	14 131

4 version where 2 bit are processed concurrently completes within $192/2 = 96$ cycles, the radix-16 version completes within 48 cycles, the radix-64 version completes within 32 cycles and the radix-256 version completes within 24 cycles. In comparison, a Comba multiplier with a 32-bit array multiplier at its core needs an area of 8718 GE (also see Table 4.7) and finishes a 192-bit multiplication within 36 clock cycles. This means, that the Comba multiplier needs almost the same area as the radix-16 shift-add multiplier, but only needs 36 clock cycles instead of 48. However, this estimation does not include reduction yet. If we use a high-radix shift-add circuit with implicit reduction capabilities, a reduction of the result to 193-bit is performed without needs of additional clock cycles. Nonetheless, this comes at the expense of an increased area. Further, there still would be one reduced reduction step necessary after completion of the multiplication, in order to reduce the 193-bit result to 192-bit, thus ensuring correct operation of the datapath. This reduced reduction step would consume one clock cycle, so the radix-16 version would need 49 cycles instead of 48 cycles.

5.4.2 Modular Reduction

The high-radix implementations including implicit reduction capabilities still need one additional clock cycle in order to reduce the 193-bit intermediate result to 192-bit. Further, implicit reduction comes at the expense of many additional adders in the datapath, on the one hand slowing down overall carry propagation, on the other hand consuming expensive chip area. Therefore, another reduction strategy was applied which is described in this section. As depicted in Figure 5.4, the high-radix shift-add circuits are implemented using an adder tree. Now, the idea was to reuse this adder tree for reduction as follows:

1. The shift-add multiplication operation is performed as usual until all bits of the multiplier **B** have been evaluated and the accumulator register contains the 384-bit result.

2. According to the NIST reduction scheme described in Section 4.3, the upper bits of the result are added to the lower bits. This is accomplished by a reuse of the adder tree at the expense of some additional multiplexers. The flow of operation is described for the radix-16 multiplier here but is very similar for other high-radix implementations:

- The first carry-save adder is supplied with the lower 192-bit of the result, the higher 192-bit of the result as well as a concatenation of the chunks B, A, and C, in accordance with the reordered NIST reduction modulo p_{192} as depicted on the right-hand side of Figure 5.2.
- The second carry-save adder, in addition to the outputs of the first one, takes the shifted chunk C as its input.
- Since the two carry-save adders only compress their inputs, the first ripple carry adder computes the sum of the carry-save adder's outputs and thus the reduced result.
- The computation of the reduced result can generate a carry, which means, that only a reduction to 193-bit is performed. Since there is still one adder of the adder tree unused, the final ripple-carry adder is utilized to perform the operation which was denoted "second reduced reduction round" above. The 193th bit therefore is shifted to the positions according to the NIST reduction and added to the remaining 192 bit, resulting in the reduction to 192-bit.

3. The reduced multiplication result is then again written into the accumulator.

Enabled by this approach, only one additional clock cycle is necessary after the shift-add multiplication to reduce the resulting 384 bit to 192 bit. Except for some multiplexers and a small enhancement of the control circuit no additional logic is required. The shift-add circuit with implicit reduction would have needed more adders and, regardless of that, would have had to perform one additional operation for the reduction to 192-bit and thus would have needed one additional clock cycle.

5.4.3 Overall Picture: Addition, Subtraction, and Multiplication

The previous sections dealt with the design of the multiplication circuit. In this section the overall design of the datapath, including addition, subtraction and multiplication will be introduced.

The multiplier is implemented as described above: A radix-16 implementation of the 192-bit shift-add multiplier **without** implicit reduction. It utilizes a small adder tree consisting of two carry-save adders and one ripple-carry adder for generation of the intermediate result as well as one further ripple-carry adder for addition of the intermediate result to the accumulator. A simple state machine is used to control the operation: For 48 rounds, the shift-add procedure is carried out, concluded with a final *reduction* round, where the multi-iteration reduced reduction as described above is performed within one clock cycle, using the same adder tree.

The second field operation required for elliptic curve point multiplication is squaring. In the final datapath squaring is executed by deployment of the shift-add multiplier, where instead of two distinct operands **A** and **B** the same operand is supplied to both operand registers. As a matter of course, such an implementation requires the same number of clock cycles for squaring as for multiplication which in our case is 49 cycles including reduction to 192-bit.

The third field operation required is addition as well as subtraction $\pmod p$. One way to handle these operations would be to use a narrow adder (e.g., a 32-bit adder) to perform multi-precision addition and subtraction, succeeded by the reduction scheme of choice, again using the narrow adder. Naturally this would need a lot of clock cycles severely

slowing down the elliptic curve point multiplication. A fast way to handle additions and subtractions would be to use a 192-bit adder straight away, which however would need far more area than the 32-bit adder. Since the shift-add multiplier deployed in the final design utilizes adders of width ≥ 192, and additions and multiplications are never executed concurrently, these adders can be used for field addition and subtraction too. In order to do this, first the design of the ripple carry adder has to be enhanced a little for also allowing subtractions. By enhancing the state machine controlling the shift-add multiplier, integer additions and subtractions are then possible: When an addition or subtraction should be performed, one of the ripple-carry adders is supplied with the operands and the result is written back into the accumulator. Thus, integer additions and subtractions are performed within one clock cycle while not needing any additional chip area (except for multiplexers and small enhancements to the control circuit, as well as enabling the adder to also perform subtractions). Further, reduction of the addition or subtraction result can be performed similar to the reduction after multiplication by again utilizing the adder tree. However, reduction after addition is simpler than after multiplication since only the reduced reduction has to be performed. Utilizing the adder tree of the shift-add multiplier also for addition and subtraction offered a great improvement regarding area and the number of clock cycles needed for a point multiplication. To still save more chip area, the accumulator register can also be used as one of the operands: Since for multiplication, the register for operand **B** can be as narrow as four bits (although making it the same width as the ram seems more practical), but for addition and subtraction both operands have to be of 192-bit width, there were three options:

1. Use a 192-bit register for operand **B** in general
2. Use an additional 192-bit register for addition and subtraction
3. Reuse the accumulator register for addition and subtraction

The third option saves most area and there are no great drawbacks regarding the cycle count when implemented carefully. The approach is possible since addition and subtraction are completed within one clock cycle, thus, at first operand **A** is loaded into the 192-bit operand register and operand **B** is loaded into the accumulator. After the addition or subtraction the result is stored in the accumulator again and after another addition (for the purpose of reduction) the reduced result is written to the accumulator.

5.4.4 Field Inversion

The Algorithm

Unlike addition, subtraction and multiplication, field inversion does not require its own dedicated hardware but can be implemented by merely using a state machine which utilizes the multiplier and adder/subtractor. Field inversion was implemented on the base of the *binary algorithm for inversion in* $\mathbb{F}_p$, taken from [15]. The algorithm is depicted in 5 and is based on the binary algorithm for finding the greatest common divisor.

Targeting Constant Execution Time

By investigation of Algorithm 5 it is apparent that a straight-forward implementation would lead to an execution time that is not constant at all. In more detail, lines 5, 6, and 14 cause problems. The inner loops are executed one after the other, again and

Algorithm 5 Binary algorithm for inversion in $\mathbb{F}_p$

Require: $p, a \in [1, p-1]$
Ensure: $a^{-1} \pmod{p}$

```
1:  u ← a;
2:  v ← p;
3:  x1 ← 1;
4:  x2 ← 0;
5:  while u ≠ 1 and v ≠ 1 do
6:     while u is even do
7:        u ← u/2;
8:        if x1 is even then
9:           x1 ← x1/2;
10:       else
11:          x1 ← (x1 + p)/2;
12:       end if
13:    end while
14:    while v is even do
15:       v ← v/2;
16:       if x2 is even then
17:          x2 ← x2/2;
18:       else
19:          x2 ← (x2 + p)/2;
20:       end if
21:    end while
22:    if u ≥ v then
23:       u ← u − v;
24:       x1 ← x1 − x2;
25:    else
26:       v ← v − u;
27:       x2 ← x2 − x1;
28:    end if
29: end while
30: if u = 1 then
31:    return x1 (mod p);
32: else
33:    return x2 (mod p);
34: end if
```

again, caused by the outer loop, until either $u = 1$ or $v = 1$. There could occur cases where only one inner loop is executed, and the number of iterations is dependent on the values u respectively v. A way to get constant execution time would be to "always execute everything" which drastically degrades the execution time by boosting the number of clock cycles necessary. Since fast point multiplications are crucial when designing an elliptic curve cryptography core for use on RFID tags, this approach seemed not fitting. However, a trade off between constant execution time and fast inversion was chosen. There are no changes made to the outer loop, but the inner loops are implemented with constant execution time in mind. The termination conditions of the inner loops already look similar. First the condition of the first loop is evaluated, then the loop body is executed. This is repeated until the condition could not be met any more and the same flow is executed for the second inner loop. Then the if condition is evaluated and everything is repeated until eventually $u = 1$ or $v = 1$. When implemented in hardware, the evaluation of the first loop condition takes some time T_1 and the loop body is executed within T_2. Both loops look the same, so T_1 and T_2 are the same for the loops in line 6 and 14. Thus, as long as the loop condition is met, the time consumption of the flow of operations looks like $(T_1, T_2, T_1, T_2, ...)$. However, when for example the loop condition of the first loop is not met anymore, the loop condition of the second loop is evaluated thereafter which means that the timing can now look like $(T_1, T_1, T_2, ...)$, that is, two consecutive evaluations of the loop condition have lead to a different timing footprint and the execution time is not constant at all. To compensate for that, after each evaluation of the loop condition, the loop body could executed. If the loop condition is not met, the body is just altered in such a way that it does not actually change values but otherwise looks exactly the same. Algorithm 6 shows one of the while loops with unchanged loop body, Algorithm 7 shows the same loop but with changes that lead to the same execution time while not altering any register values.

Another way to hide which of the two inner loops is executed would be as follows: Instead of *always* executing the loop body, the loop condition evaluation is performed twice. When for example the first loop condition is not met, the second one is evaluated leading to the timing footprint $(T_1, T_1, ...)$. Thus, the algorithm actually was implemented the following way: First, the first loop condition is evaluated. If it is not met, the second loop condition is evaluated, leading to $(T_1, T_1, ...)$. However, if the condition was met, it is re-evaluated nevertheless (the outcome of course is the same), leading to the same timing footprint. After the second evaluation, the loop body is executed, either of the first loop or of the second loop. In case the loop condition was not met, the if condition is evaluated. Thus, by always evaluating the loop conditions twice, there is no way to tell from the timing whether the body of the first loop or the body of the second loop was executed.

Both bodies of the inner loops contain a conditional addition (either $x \leftarrow x/2$ or $x \leftarrow (x + p)/2$ is executed). The difference regarding the execution time hereby is taken care of by performing an addition in both cases (adding 0 when x is not even, as depicted in Algorithm 7.

Additional Required Hardware

The inversion as depicted in Algorithm 5 contains several if and loop conditions which require comparison (i.e., $\neq$ and $\geq$). Naturally, a comparison circuit of some form is needed in hardware for this purpose. Since u, v, x_1, and x_2 have a width of 192-bit, a

Algorithm 6 Unchanged loop body

```
1: while u is even do
2:     u ← u/2;
3:     if x1 is even then
4:         x1 ← x1/2;
5:     else
6:         x1 ← (x1 + p)/2;
7:     end if
8: end while
```

192-bit wide comparator would be necessary for comparisons within one clock cycle. In order to keep the area within limits, a multi-precision approach was taken, utilizing a narrow comparator. Of course this has a negative impact on the cycle count of inversion and therefore the point multiplication, but since only one inversion is required per point multiplication due to the usage of projective coordinates, the performance loss is not too drastically. The comparator was implemented having the same width as the RAM and was wired to its output. The comparison therefore can be carried out concurrently while loading the respective operator into an operand register within $192/ram\_width$ clock cycles. Since the operands are loaded into the registers beginning with the least significant word, the comparison $\geq$ is carried out as follows: The comparator always outputs two flags, one indicating *greater or equal* (**goe**) and one indicating *equal* (**eq**). A flag is initialized to logic 1. For each evaluated word, the flag is set if **goe** is 1 and **equal** is zero. The flag is reset if **goe** is zero. In hardware, the flag is implemented as a 1-bit register. After the last word was loaded, a logic high value of the register indicates the comparison result "grater or equal". The comparison described actually tests for $>$, the test for $=$ is taken care of by initializing the register to logic one. A pseudo-code representation is given in Algorithm 8, for a 192-bit datapath with 32-bit RAM width.

The evaluations regarding whether an operand is even or odd does not need to utilize the multi-precision comparator. Instead, only the least significant bit has to be checked for deciding whether the operand is even (the bit is logic high) or odd (the bit is logic low). No clock cycles are wasted for this check.

5.4.5 The Memory Architecture

The elliptic curve crypto core includes two operand registers **A** and **B** and an accumulator register **accu**. The operand register **A** is a 192-bit register used for all multiplication,

Algorithm 7 Dummy loop body

```
1: while u is even do
2:     u ← u/1;
3:     if x1 is even then
4:         x1 ← x1/1;
5:     else
6:         x1 ← (x1 + 0)/1;
7:     end if
8: end while
```

Algorithm 8 Least-significant word first multi-precision $\geq$ comparison

Require: $a, b \in [0, 2^{192} - 1]$
1: $goe \leftarrow 1$;
2: **for** $i = 0$ to 5 **do**
3: **if** A[i] $\geq$ B[i] and A[i] $\neq$ B[i] **then**
4: $goe \leftarrow 1$;
5: **end if**
6: **if** A[i] $<$ B[i] **then**
7: $goe \leftarrow 0$;
8: **end if**
9: **end for**
10: **return** *goe*;

addition, and subtraction. In case of subtraction, **A** is always used as the minuend. The reason for this is, that experience has shown, that with 192-bit registers the area requirements for wiring and especially for additional multiplexers are relatively demanding. The operand register **B** is only used for multiplication. It is only 4-bit wide and used to control the select wires of the multiplexers shown in Figure 5.4. During a multiplication it is updated regularly. The **accu** has a width of 384 bit. It is used as result register for all field operations. It is the only register that can be written to the RAM. The **accu** is also used as the second operand for addition and subtraction in order to save memory and therefore chip area. In case of subtraction, the **accu** always contains the subtrahend (again for area reasons).

The RAM used is a 64*32 bit single-ported RAM. The 32-bit interface was chosen as a trade off between area and speed. By using RAM with greater width the algorithm could be sped up significantly. The RAM is used for storing 192-bit intermediate values also referred to as "field registers" in the algorithm descriptions below. Further, the RAM contains the scalar k for the point multiplication as well as the point P on the curve. The base point of the curve is implemented as a constant since it never changes.

5.5 Point Multiplication: Montgomery Ladder with (X,Y)-only Co-Z Doubling Addition

5.5.1 Introduction

Several methods for the elliptic curve point multiplication $k * P$, where k is a scalar and P is a point on the curve have been found to this day. Such a point multiplication is the core operation of cryptographic protocols based on elliptic curves, like ECDH or ECDSA. Same as the underlying datapath, elliptic curve point multiplication should be fast as well as robust with respect to side-channel attacks. Very basic algorithms for point multiplication for example would be the left-to-right and the right-to-left binary method ([15]) which utilize *point doublings* and *point additions* as described in Section 2.3.5. Unfortunately, both algorithms rely on conditional point additions, which makes them prone to side-channel attacks since the flow of operations and thereby the execution time is depending on the input scalar. Further, a comparison in [15] shows that these two algorithms are extremely slow when compared to more sophisticated approaches.

In [29] Rivian describes methods for point multiplication that take care of both draw-

backs of the basic algorithms mentioned previously, namely side channel attacks and efficiency. For withstanding side-channel attacks, the algorithms he proposes are *regular*, that is, their flow of operation is independent of the input scalar k. When implemented in hardware, this should lead to a cycle count that is independent of k. From Rivian's proposed algorithms the *Montgomery ladder with (X, Y)-only co-Z doubling addition* was chosen. It is shown in Algorithm 9.

5.5.2 Fundamentals

Using Jacobian Coordinates

It was shown in Section 2.3.5, that when using affine coordinates for representation of points on the elliptic curve field inversion is necessary for both *point addition* and *point doubling*. Since field inversions are expensive operations with respect to efficiency, usage of projective coordinates instead avoids the field inversions during point multiplication. Only one field inversion is required for transforming the projective coordinates back to affine coordinates. Jacobian projective coordinates [1] enable fast point doubling (see [29]), hence they are used here.

Co-Z Addition

An optimization regarding Jacobian addition is the so-called co-Z addition that can be performed when two points on the elliptic curve share the same Z-coordinate ([29]): The addition $\mathbf{P} + \mathbf{Q}$ of two points $\mathbf{P} = (X_1, Y_1, Z)$ and $\mathbf{Q} = (X_2, Y_2, Z)$ sharing a common Z-coordinate (where $\mathbf{P} \neq \mathbf{Q}$) is very efficient and enables a free update of the coordinates of $\mathbf{P}$ such that $\mathbf{P}$ and $\mathbf{P} + \mathbf{Q}$ again share a common Z-coordinate. Therefore, subsequent additions of $\mathbf{P}$ and $\mathbf{P} + \mathbf{Q}$ are possible. Further, for a small additional cost the conjugate $\mathbf{P} - \mathbf{Q}$ with the same Z-coordinate as $\mathbf{P} + \mathbf{Q}$ can be computed. The addition of $(\mathbf{P} + \mathbf{Q})$ and $(\mathbf{P} - \mathbf{Q})$, where $(\mathbf{P} + \mathbf{Q})$ and $(\mathbf{P} - \mathbf{Q})$ share a common Z-coordinate is called co-Z *conjugate* addition.

Montgomery Ladder

The Montgomery ladder is a regular algorithm for point multiplication without any conditional executed parts which benefits from (X,Y)-only co-Z arithmetic by rewriting the loop body in such way that a conjugate addition followed by a regular addition is performed instead of an addition and a doubling (see [29]).

5.5.3 The Algorithm

Algorithm 9 shows the Montgomery ladder with (X,Y)-only co-Z doubling addition which is an efficient and regular algorithm for elliptic curve point multiplication. The algorithm was taken from [29] and chosen for the implementation of the crypto core for its beneficial properties regarding robustness to side-channel attacks and fast execution of the point multiplication. It can be seen that the Algorithm does not contain any conditional parts. However, the execution time is highly dependent on the bit width of the input scalar k. This is taken care of by requiring that $n = 192$ for the curve P-192 and $k_n - 1$ is always 1. Hence, one bit of security is sacrificed for constant execution time, leading

[1] Remember: Jacobian projective coordinates are projective coordinates where $c = 2$ and $d = 3$

Algorithm 9 Montgomery ladder with (X,Y)-only co-Z doubling addition

Require: $\mathbf{P} \in E(\mathbb{F}_q), k = (k_{n-1}, ..., k_1, k_0)_2 \in \mathbb{N}$ with $k_{n-1} = 1$
Ensure: $\mathbf{Q} = k * \mathbf{P}$
1: $(R_1, R_0) \leftarrow XYCZ - IDBL(\mathbf{P})$;
2: $b \leftarrow k_{n-2}$;
3: $(R_{1-b}, R_b) \leftarrow XYCZ - ADDC(R_b, R_{1-b})$
4: **for** $i = n - 2$ to 1 **do**
5: $b \leftarrow k_i$;
6: $d \leftarrow k_{i-1}$;
7: $s \leftarrow d \oplus b$;
8: $(R_{1-d}, R_d) \leftarrow XYCZ - DA(R_{1-b}, R_b)$;
9: $R_d \leftarrow (-1)^s R_d$;
10: **end for**
11: $b \leftarrow k_0$;
12: $\lambda \leftarrow$ FinalInvZ$(R_0, R_1, \mathbf{P}, b)$;
13: $(R_b, R_{1-b}) \leftarrow XYCZ - ADD(R_{1-b}, R_b)$;
14: **return** $(X_0\lambda^2, Y_0\lambda^3)$

to a degradation of the security level from 96 bit to 95,5 bit. The Algorithm starts with the (X,Y)-only initial doubling with co-Z update (XYCZ-IDBL, see 5.5.3) followed by a first conjugate addition (XYCZ-ADDC, see 11). Depending on the value of the second-most significant bit of the input scalar k, either the (X,Y)-coordinates of $(R_1 + R_2)$ and $(R_1 - R_2)$ or of $(R_2 + R_1)$ and $(R_2 - R_1)$ is calculated thereby. R_i denotes the content of the respective register. Then, for each remaining bit of k an (X,Y)-only co-Z doubling addition with update (XYCZ-DA, see 12) is performed, again depending on the bit values of k. The conditional point inversion (negation of the Y-coordinate) seen in line 9 can be implemented based on the evaluation of the arithmetic expression, hence no if condition is necessary which would hurt the regularity of the algorithm and make it prone to side-channel attacks. By FinalInvZ (see 5.5.3) the inverse of the final Z-coordinate can be computed, which is needed to transform the projective coordinates back to affine coordinates. The final (X,Y)-only co-Z addition with update (XYCZ-ADD, see 10) yields the result of the point multiplication, namely (X_0, Y_0) in projective coordinates.

(X,Y)-only Initial Doubling with Co-Z Update (XYCZ-IDBL)

The initial (X,Y)-only point doubling with co-Z update from [29] takes a point $\mathbf{P}$ on the elliptic curve (provided in affine coordinates) and computes the (X,Y)-only co-Z points $\mathbf{P}$ and $[2]\mathbf{P}$ such that $\mathbf{P} \equiv (X_1' : Y_1' : Z_2)$ and $[2]\mathbf{P} \equiv (X_2 : Y_2 : Z_2)$ for some $Z_2 \in \mathbb{F}_p$ by a Jacobian doubling of $\mathbf{P} = (x_1, y_1, 1)$ followed by an update of $\mathbf{P}$ such that it shares a common Z-coordinate with $\mathbf{Q} = [2]\mathbf{P} = (X_2, Y_2, Z)$.

(X,Y)-only Co-Z Addition with Update (XYCZ-ADD)

The XYCZ-ADD algorithm taken from [29] and shown in 11 first computes the (X,Y)-only co-Z addition $\mathbf{P} + \mathbf{Q}$ from the points $\mathbf{P}$ and $\mathbf{Q}$ sharing a common Z-coordinate. Then, the (X,Y) coordinates of $\mathbf{P}$ are updated such that $\mathbf{P}$ and $\mathbf{P} + \mathbf{Q}$ share a common Z-coordinate. The algorithm can be performed by the crypto core using six field multiplications and seven field additions.

Algorithm 10 (X,Y)-only co-Z addition with update (XYCZ-ADD)

Require: (X_1, Y_1) and (X_2, Y_2) such that $\mathbf{P} \equiv (X_1 : Y_1 : Z)$ and $\mathbf{Q} \equiv (X_2 : Y2 : Z)$ for some $Z \in \mathbb{F}_q, \mathbf{P}, \mathbf{Q} \in E(\mathbb{F}_q)$

Ensure: (X_3, Y_3) and (X'_1, Y'_1) such that $\mathbf{P} \equiv (X'_1 : Y'_1 : Z_3)$ and $\mathbf{P} + \mathbf{Q} \equiv (X_3 : Y3 : Z_3)$ for some $Z_3 \in \mathbb{F}_q$

1: $A \leftarrow (X_2 - X_1)^2$;
2: $B \leftarrow X_1 A$;
3: $C \leftarrow X_2 A$;
4: $D \leftarrow (Y_2 - Y_1)^2$;
5: $E \leftarrow Y_1(C - B)$;
6: $X_3 \leftarrow D - (B + C)$;
7: $Y_3 \leftarrow (Y_2 - Y_1)(B - X_3) - E$;
8: $X'_1 \leftarrow B$;
9: $Y'_1 \leftarrow E$;
10: **return** $((X_3, Y_3), (X'_1, Y'_1))$;

(X,Y)-only Co-Z Conjugate Addition (XYCZ-ADDC)

XYCZ-ADDC is similar to XYCZ-ADD, but performs a conjugate addition, that is, it computes the (X,Y)-coordinates of $\mathbf{P} + \mathbf{Q}$ and $\mathbf{P} - \mathbf{Q}$ where $\mathbf{P} + \mathbf{Q}$ and $\mathbf{P} - \mathbf{Q}$ again share a common Z-coordinate. Algorithm 11 is again taken from [29], where two low-level implementations are given for different trade offs between memory requirements (which is analogue to area when considering a hardware implementation) and efficiency. For the purpose of keeping the chip area of the crypto core within limits, the less efficient algorithm was chosen, which trades one field register for five additions. It computes the conjugate addition using eight field multiplications and sixteen field additions.

(X,Y)-only Co-Z Doubling Addition with Update (XYCZ-DA)

The XYCZ-DA algorithm depicted in Algorithm 12 was again taken from [29]. Doubling addition with update can be performed by a consecutive execution of Algorithm 10 and Algorithm 11, that is, by fist performing an addition with update, followed by a conjugate addition, both in terms of (X,Y)-only co-Z arithmetic.

FinalInvZ

FinalInvZ in [29] indicates the calculation of $\lambda = Z^{-1}$ (where Z^{-1} denotes the *inverse* of Z). It is performed as follows:

$$\lambda = X_b * y_p * (x_p * Y_p * (X_1 - X_0))^{-1} \tag{5.1}$$

X_b and Y_b denote the coordinates in register R_0 respectively R_1, b is the value of the least significant bit of k, and $(x_p, y_p) = P$ are the coordinates of the elliptic curve point P. The computational costs of FinalInvZ are one field addition $(X_1 - X_0)$, four field multiplications and one field inversion.

5.5.4 Implementation

The Montgomery ladder with (X,Y)-only co-Z doubling addition algorithm was implemented as a finite state machine (FSM). The FSM controls the underlying circuits for

Algorithm 11 (X,Y)-only co-Z conjugate addition (XYCZ-ADDC)

Require: (X_1, Y_1) and (X_2, Y_2) such that $\mathbf{P} \equiv (X_1 : Y_1 : Z)$ and $\mathbf{Q} \equiv (X_2 : Y2 : Z)$ for some $Z \in \mathbb{F}_q, \mathbf{P}, \mathbf{Q} \in E(\mathbb{F}_q)$
Ensure: (X_3, Y_3) and (X_1', Y_1') such that $\mathbf{P} + \mathbf{Q} \equiv (X_1' : Y_1' : Z_3)$ and $\mathbf{P} - \mathbf{Q} \equiv (X_3 : Y3 : Z_3)$ for some $Z_3 \in \mathbb{F}_q$
1: $A \leftarrow (X_2 - X_1)^2$;
2: $B \leftarrow X_1 A$;
3: $C \leftarrow X_2 A$;
4: $D \leftarrow (Y_2 - Y_1)^2$;
5: $E \leftarrow Y_1(C - B)$;
6: $F \leftarrow (Y_1 + Y_2)^2$;
7: $X_3 \leftarrow D - (B + C)$;
8: $Y_3 \leftarrow (Y_2 - Y_1)(B - X_3) - E$;
9: $X_3' \leftarrow F - (B + C)$;
10: $Y_3' \leftarrow (Y_1 + Y_2)(X_3' - B) - E$;
11: **return** $((X_3, Y_3), (X_3', Y_3'))$;

loading registers, writing to the RAM and storing (intermediate) results in the RAM, multiplication, addition, subtraction and inversion. The FSM performs a sequence of multiplications, additions, and subtractions, as suggested in [29], which is a schedule using only eight field registers. Field multiplications are performed by first loading operand **A**, then loading operand **B**, followed by the first shift-add iteration. Operand **B** is updated in each cycle with the next four bit from the RAM. After a reduction step, the content of the RAM is either written back to the RAM, or, if possible, directly used as the second operand of a consecutive addition or subtraction. Additions and subtractions are also performed by first loading the operand **A** register followed by loading of the second operand to the **accu**. During the course of the point multiplication the 32-register holding part of the scalar k has to be (re)loaded six times. At the very end of the Montgomery ladder point multiplication, one final subtraction is performed (on both the X- and the Y-coordinate of the resultant point on the curve), in order to reduce the result to $i < p$.

5.6 Results

In this chapter the final results are presented. The final elliptic curve cryptography core was implemented using *Very High Speed Integrated Circuit Hardware Description Language* VHDL. Hardware synthesis was performed by the Cadence RTL compiler using the UMC-

Algorithm 12 (X,Y)-only co-Z doubling addition with update (XYCZ-DA)

Require: (X_1, Y_1) and (X_2, Y_2) such that $\mathbf{P} \equiv (X_1 : Y_1 : Z)$ and $\mathbf{Q} \equiv (X_2 : Y2 : Z)$ for some $Z \in \mathbb{F}_q, \mathbf{P}, \mathbf{Q} \in E(\mathbb{F}_q)$
Ensure: (X_4, Y_4) and (X_4', Y_4') such that $2\mathbf{P} + \mathbf{Q} \equiv (X_4' : Y_4' : Z_4)$ and $\mathbf{Q} \equiv (X_4' : Y_4' : Z_4)$ for some $Z_4 \in \mathbb{F}_q$
1: $((X_3, Y_3), (X_1', Y_1')) \leftarrow XYCZ - ADD((X_1, Y_1), (X_2, Y_2))$;
2: $((X_4, Y_4), (X_4', Y_4')) \leftarrow XYCZ - ADDC((X_3, Y_3), (X_1', Y_1'))$;
3: **return** $((X_4, Y_4), (X_4', Y_4'))$;

Table 5.3: Synthesis results

Component	**Area [GE]**
Montgomery-ladder FSM	3793
Shift-add circuit w/o registers	10 430
Control logic	115
$\sum$	**10545**
ECC crypto core w/o RAM	20753
2048-bit single-ported RAM macro	3451
$\sum$	**24204**

L130 CMOS technology.

5.6.1 Area

The synthesis results of the elliptic curve cryptography core regarding chip area can be seen in Table 5.3. The lower section of the Table shows the overall results, while the upper section shows partial results that are of particular interest. The whole core including a 2048-bit single-ported RAM macro with 32-bit interface needs an area of 24 204 GE. The area requirements of the finite state machine implementing the Montgomery ladder with (X,Y)-only co-Z doubling-addition algorithm as depicted in Algorithm 9 are relatively demanding. One particular reason for this is, that the algorithm is partitioned into three sub algorithms as described in Section 5.5, which all require a different flow of field operations. As predicted in Chapter 5, the area requirements for the control logic of the shift-add multiplication circuit are small (115 GE). The synthesis result of the shift-add circuit's datapath is 10 430 GE. The area of the shift-add multiplier was estimated to need 10 776 GE (cf. Table 5.2). However, this estimation result included the operand registers and the accumulator, both are not included in the final result in Table 5.3. The disparity between the estimation and the result can be explained by additional routing expenses and especially additional multiplexers needed for the final circuit: The shift-add multiplier also performs the reduction as well as field additions and field subtractions.

5.6.2 Comparison with Related Work

Table 5.4 gives a comparison of the final elliptic curve crypto core with related implementations. Regarding the area, the final core has an area comparable to the works of Wolkerstorfer [34] and Fürbass et al. [13]. Both implementations deal with elliptic curves over a 192-bit prime field. The implementation presented in this work needs 42,5% less clock cycles than the work of Fürbass et al. while having similar area requirements. For sake of comparison the work of Hein [16] is also included in the Table, which deals with an elliptic curve cryptography implementation for the binary field $\mathbb{F}_{2^{163}}$. It can be seen that the cycle count needed by the crypto core presented in this work is in the same range.

Table 5.4: Comparison of the ECC core implementation with related work

	Area [GE]	**# Cycles [kCycles]**	**ECC Curve**
Wolkerstorfer 2005 [34]	23818	678	$\mathbb{F}_{p192}$
Fürbass 2007 [13]	23656	500	$\mathbb{F}_{p192}$
This work 2013	**24204**	**287**	$\mathbb{F}_{p192}$
Hein 2008 [16]	13250	296	$\mathbb{F}_{2^{163}}$

Chapter 6

Conclusions and Outlook

The ambition of this thesis was the realization of the Internet of Things by integrating passive RFID devices into the Internet and providing a way to establish a secure end-to-end connection between the tag and a corresponding node. The work was separated into the implementation of an elliptic curve crypto core as well as the implementation of the basic IPsec functionality on the IAIK UHF DemoTag in software.

The focus of the crypto core implementation was on finding a suitable design for the proposed requirements, i.e., a fast, low-area implementation capable of point multiplication on an elliptic curve suitable for the IPsec Diffie-Hellman key exchange. In order to keep the required chip area within limits, the smallest prime curve defined for use in IKEv2 was chosen, i.e., the NIST curve P-192 over $\mathbb{F}_{p192}$. An extensive design-space exploration was carried out, in order to find a good trade off between area requirements and cycle count for the execution of a point multiplication. The decision fell in favor of a design which is very fast while keeping the area requirements within limits. A fast elliptic curve crypto core is convenient for use with IPsec. Point multiplication in general is a demanding task, and the Diffie-Hellman key exchange should be able to be performed quickly so as to enable a communication over the Internet, e.g., while a tag is merely passing the field of an RFID reader.

The final design of the elliptic curve crypto core was synthesized using the UMC-L130 CMOS technology. The crypto core consumes an area of 24 kGE while only needing 287 k clock cycles for a elliptic curve point multiplication. This is a very satisfactory result since the cycle count is in the same range as cycle counts of implementations over binary curves (cf. Table 5.4).

There is still room for improvement for future uses of the crypto core. The RAM interface could be changed to 64-bit in order to trade some chip area for even faster point multiplications. While the datapath is highly efficient, the control circuit for the Montgomery-ladder point multiplication could be further optimized regarding area consumption. Loading and storing the 192-bit operands is still a major factor contributing to the overall cycle count. By implementing a register bank instead of the RAM, an extremely fast ECC core could be produced.

For the realization of a secure end-to-end connection between an RFID tag and a corresponding node concepts like Mobile IPv6 and IPsec were considered. The implementation of the security layer for Internet of Things-enabled RFID tags is strongly based on the concept work of Dominikus et al. [7]. Nevertheless, plenty of work had to be put into refining the concept and deploying it for a real implementation. Since RFID tags do not have much memory in general, part of the complexity of IPsec is shifted to the reader.

The main challenge thereby is the key exchange via IKEv2. The protocol is very complex and not really suited for low resource implementations. Partial work can be offloaded to the reader. However, later payloads are encrypted and decrypted payloads must not leave the tag, so the duties of the reader are limited. Some constraints had to be made in order to realize the concept regardless. The modified protocol still mostly complies with the original IKEv2, i.e., the exchanges are the same, used algorithms are the same and payloads are the same. Changes that had to be applied were of subtle nature, i.e., only one transform for each type is supported at the moment (AES in CBC mode for encryption, AES-XCBC-MAC-96 for integrity protection, AES-XCBC-128 as pseudo-random function and the 192-bit random ECP group for the Diffie-Hellman exchange) and some payloads that originally are of variable size (e.g., nonces) are restricted to their respective minimal size that is still within the definitions of IKEv2.

All in all, the concept looks very promising. Challenges regarding the memory requirements will automatically be reduced in the future by memory becoming smaller and cheaper. While some additional assumptions to those proposed by Dominikus et al. [7] had to be made, no part of the implementation contradicts the original IKEv2/IPsec protocol. For now, integration of passive RFID tags into the Internet of Things is already possible as long as the corresponding node knows that it is talking with an RFID tag and therefore restricting payload sizes and used protocols to those supported by the tag.

Appendix A

Definitions

A.1 Abbreviations

AES	Advanced Encryption Standard
ASIC	Application Specific Integrated Circuit
AH	Authentication Header
CSA	Carry Save Adder
CSA	Carry Save Adder
CBC	Cipher Block Chaining
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman
ESP	Encapsulating Security Payload
FSM	Finite State Machine
FA	Full Adder
GE	Gate Equivalent
HDL	Hardware Description Language
IV	Initialization Vector
IoT	Internet of Things
IKEv2	Internet Key Exchange version 2
IPSec	Internet Protocol Security
IPv6	Internet Protocol Version 6
LSB	Least Significant Bit
MSB	Most Significant Bit
MUX	Multiplexer
NIST	National Institute of Standards and Technology
PRF	Pseoudo-Random Function
RFID	Radio Frequency Identification
RAM	Random Access Memory
RCA	Ripple Carry Adder
SA	Security Association
VLSI	Very Large Scale Integration

Appendix B

Datasheet

B.1 Features

- General features:
  - High Performance Elliptic Curve Point Multiplication core for the NIST curve P-192
  - Constant execution time
  - 95,5 bit security
  - UMC-L130 CMOS technology
  - 32-bit Interface
  - Supply voltage: 1.2 V
- Clock:
  - Target clock frequency: 2 MHz
  - Maximum clock frequency: 3.482 MHz
- Core area:
  - 24204 GE[1]
- Core size:
  - 332.1 mm x 332.1 mm^2
- Cryptographic performance:
  - 287 k clock cycles / point multiplication

[1] After synthesis, including area for the 64 x 32-bit RAM macro, excluding pads
[2] after place and route, excluding RAM, excluding pads

B.2 Port Description

Table B.1: Port description of the Elliptic Curve Cryptography Core

Signal Name	**Type**	**Description**
din32	in	32-bit parallel data input
dout32	out	32-bit parallel data output
startLadder	in	The start signal for execution of the point multiplication
addrRAM	in	6-bit parallel RAM address
loadRAM	in	Triggers write enable for the RAM
basepoint	in	Indicates whether multiplication with the base point of the curve or with an arbitrary point should be performed.
getX	in	Read out the X-coordinate of the resulting point $Q = k * P$ via dout32.
getY	in	Read out the Y-coordinate of the resulting point $Q = k * P$
fin	out	This flag indicates that the calculation of a point multiplication is completed
reset	in	Asynchronous reset signal
clk	in	Clock input

Appendix C

Sage Script for Radix-16 Multiplier Verification

```
##############################
#192 Bit HW-like solution#
##############################

accu=0;
p=2^192-2^64-1;
a= 0x12345CAFE6846816168DEADBEEF761300848409846369F86;
b= 0xFA936FCCEE684513313E8FAA23000E68C351DCF843000338;
r=a*b;

for i in range(0,48):
    bit_b1 = b & 0x1;
    bit_b2 = b & 0x2;
    bit_b3 = b & 0x4;
    bit_b4 = b & 0x8;
    b=b>>4;

    if (bit_b1 == 1):
        mux1_out = a;
    else:
        mux1_out = 0;

    if (bit_b2 == 2):
        mux2_out = 2*a;
    else:
        mux2_out = 0;

    if (bit_b3 == 4):
        mux3_out = 4*a;
    else:
        mux3_out = 0;
```

```
    if (bit_b4 == 8):
        mux4_out = 8*a;
    else:
        mux4_out = 0;

    csa1_out = (mux1_out.__xor__(mux2_out)).__xor__(mux3_out);
    csa1_cy = (mux1_out & mux2_out) | (mux1_out & mux3_out) |\
    (mux2_out & mux3_out);

    csa2_out = (csa1_out.__xor__(csa1_cy *2)).__xor__(mux4_out);
    csa2_cy = (csa1_out & (csa1_cy*2)) | (csa1_out & mux4_out) |\
    ((csa1_cy*2) & mux4_out);

    tmp_sum = csa2_out + csa2_cy*2;

    a_lo=(tmp_sum<<(i*4)).mod(2^192);
    a_hi=(tmp_sum<<(i*4))>>192;

    t1 = accu + a_lo;
    t2 = t1 + a_hi;
    t3 = t2 + a_hi*2^64;

    tmp_lo = t3.mod(2^192);
    tmp_hi = t3>>192;

    t4 = tmp_lo+ tmp_hi;
    t5 = t4 + tmp_hi*2^64;

    accu = t5

print accu==r.mod(p)
```

Bibliography

[1] Information technology Radio frequency identification for item management Part 6: Parameters for air interface communications at 860 MHz to 960 MHz.

[2] Internet Key Exchange Version 2 (IKEv2) Parameters. http://www.iana.org/assignments/ikev2-parameters/ikev2-parameters.xml, April 2013.

[3] L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[4] M. Darianian and M. P. Michael. Smart Home Mobile RFID-based Internet-Of-Things Systems and Services. In *International Conference on Advanced Computer Theory and Engineering*, 2008.

[5] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, December 1998.

[6] S. Dominikus, M. Aigner, and S. Kraxberger. Passive RFID Technology for the Internet of Things. In *Internet Technology and Secured Transactions (ICITST), 2010 International Conference for*, pages 1–8. IEEE, 2010.

[7] S. Dominikus and S. Kraxberger. Secure Communication with RFID Tags in the Internet of Things. *Security and Communication Networks*, 2011.

[8] M. Feldhofer and J. Wolkerstorfer. Strong Cypto for RFID Tags-A Comparison of Low-Power Hardware Implementations. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 1839–1842. IEEE, 2007.

[9] K. Finkenzeller and R. Waddington. *RFID Handbook: Radio-Frequency Identification Fundamentals and Applications*. John Wiley, 1999.

[10] P. FIPS. 197: Advanced Encryption Standard (AES). *National Institute of Standards and Technology*, 2001.

[11] S. Frankel, R. Glenn, and S. Kelly. The AES-CBC Cipher Algorithm and its Use with IPsec. *RFC3602*, 2003.

[12] S. Frankel and H. Herbert. The AES-XCBC-MAC-96 Algorithm and Its Use With IPsec. *RFC 3566*, 2003.

[13] F. Furbass and J. Wolkerstorfer. ECC Processor with low Die Size for RFID Applications. In *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pages 1835–1838. IEEE, 2007.

[14] C. Furlani. FIPS 186-3: Digital Signature Standard (DSS). *Online, National Institute of Standards and Technology (NIST) Std.(June 2009)*, 2009.

[15] D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[16] D. Hein, J. Wolkerstorfer, and N. Felber. ECC is Ready for RFID -A Proof in Silicon. In *Selected Areas in Cryptography*, pages 401–413. Springer, 2009.

[17] P. Hoffman. The AES-XCBC-PRF-128 Algorithm for the Internet Key Exchange Protocol (IKE). *RFC 4434*.

[18] M. Hutter, S. Mangard, and M. Feldhofer. Power and EM Attacks on Passive 13.56MHz RFID Devices. In *Cryptographic Hardware and Embedded Systems-CHES 2007*, pages 320–333. Springer, 2007.

[19] C. Kaufman et al. Internet Key Exchange (IKEv2) Protocol. RFC 4306, 2005.

[20] C. Kaufman, P. Hoffman, Y. Nir, and P. Eronen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 5996, September 2010.

[21] S. Kent. IP Authentication Header (AH) . RFC 4302, 2005.

[22] S. Kent. IP Encapsulating Security Payload (ESP). RFC 4303, 2005.

[23] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301, December 2005.

[24] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC press, 2010.

[25] C. Paar and J. Pelzl. *Understanding Cryptography*. Springer-Verlag Berlin Heidelberg, 2010.

[26] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 2000.

[27] C. Perkins, D. Johnson, and J. Arkko. Mobility Support in IPv6. Technical report, RFC 6275, July, 2011.

[28] T. Popp, E. Oswald, and S. Mangard. Power Analysis Attacks and Countermeasures. *Design & Test of Computers, IEEE*, 24(6):535–543, 2007.

[29] M. Rivian. Fast and Regular Algorithms for Scalar Multiplication over Elliptic Curves. Technical report, Cryptology ePrint Archive, 2011.

[30] J. I. Schiller. Cryptographic Algorithms for use in the Internet Key Exchange Version 2 (IKEv2). 2005.

[31] Y. Sheffer, H. Tschofenig, and P. Eronen. An Extension for EAP-Only Authentication in IKEv2. 2010.

[32] E. Wenger and M. Hutter. A Hardware Processor Supporting Elliptic Curve Cryptography for Less than 9 kGEs. In *Smart Card Research and Advanced Applications*, pages 182–198. Springer, 2011.

[33] E. Wenger and M. Hutter. Exploring the Design Space of Prime Field vs. Binary Field ECC-Hardware Implementations. In *Information Security Technology for Applications*, pages 256–271. Springer, 2012.

[34] J. Wolkerstorfer. Is Elliptic-Curve Cryptography Suitable for Small Devices? In *Workshop on RFID and Lightweight Crypto*, 2005.