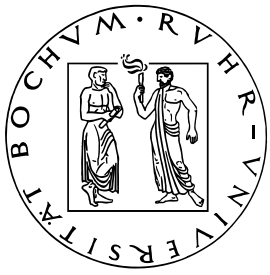
Diploma Thesis

Security in Ad-hoc Networks: Protocols and Elliptic Curve Cryptography on an Embedded Platform

Ingo Riedel (riedeli@web.de)

March 2003

Ruhr-Universität Bochum



Chair for Communication Security
Prof. Dr.-Ing. Christof Paar

NEC Europe Ltd. – Network Laboratories, Heidelberg
Dr. Dirk Westhoff, Dr. Bernd Lamparter

Erklärung

Hiermit versichere ich, dass ich meine Diplomarbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Ort, Datum

Unterschrift

Acknowledgements

Many thanks to NEC Europe Ltd. in Heidelberg, Germany, for providing the equipment and financial support for this work. In particular, I would like to thank Dr. Bernd Lamparter and Dr. Dirk Westhoff for their commitment and fruitful ideas.

Also, I would like to thank my advisors Prof. Christof Paar and André Weimerskirch as well as the entire Communication Security Group at the Ruhr-University Bochum for their support.

Abstract

Due to the insecure nature of the wireless link and their dynamically changing topology, wireless ad-hoc networks require a careful and security-oriented approach for designing communication protocols. One problem in multi-hop ad-hoc networks is to motivate network nodes to yield some of their constrained resources and to participate in forwarding the data packets of other nodes through the network. As a solution for ad-hoc networks with fixed backbone, e.g. Internet or intranet access networks, the secure charging protocol has been proposed. In this protocol, source and destination nodes are charged and the intermediate nodes receive a monetary reward for forwarding. The charging information is protected using digital signatures to establish authentication, integrity and non-repudiation for the charging information.
In this work, we study the use of digital signatures within this charging protocol. Our analysis includes the determination of appropriate key sizes and the evaluation of different signature schemes for this particular application. Focusing on the classical RSA scheme and the ECDSA scheme, which is based on elliptic curve cryptography, we propose a measure to assess the performance of these different schemes within the protocol.
Moreover, we developed a speed-optimized implementation of the elliptic curve digital signature algorithm (ECDSA) and integrated it into a prototype implementation of the secure charging protocol. In order to enhance the portability and to allow easy reuse for other applications, we designed our implementation such that it can easily replace the ECDSA routines of the open-source crypto library OpenSSL 0.9.8. On a Sharp Zaurus PDA platform, which is a typical device used in wireless ad-hoc networks, our implementation has an execution time of 5.4ms for signature generation and 11.7ms for signature verification. These times were obtained for 163-bit Koblitz curves. The times of our implementation are more than 3 times faster than the execution times of the corresponding routines of the OpenSSL project.

Contents

1. Introduction

In the past years, the use of wireless communication devices has heavily increased. A large number of handhelds, portables and mobile phones are sold every year and embedded processors with wireless communication abilities are about to be built into automobiles, refrigerators, TV sets, and microwave ovens. In the future, intelligent bar codes, wearable computers, and sensor networks will be a part of every day life. Hence, very small computer devices with wireless communication abilities will one day be embedded in almost every product. As many of those small devices will be highly mobile, communication will take place over decentralized and distributed networks with dynamically changing topology, so called *ad-hoc networks*. This everytime, everywhere connectivity will offer a broad range of new services.

However, in spite of the tempting advantages of such scenarios one must also consider many new security threads imposed by them. Due to the decentralized nature of ad-hoc networks, security requirements are different from those of traditional networks. Problems are caused for example by the weak physical protection of the network nodes, the inherent insecurity of the wireless communication channel, the mobility of the nodes and their limited processor and battery resources. Attackers may eavesdrop on communications, gain unauthorized access to devices or simply disable them by excessive battery exhaustion. Moreover, as John Doe will buy and use these devices, possibly without being even aware of the security threads, it is up to the manufactures to develop products that provide a sufficient level of security.

Several approaches to enhance the security in ad-hoc networks have been proposed. Most of them are based on cryptographic primitives such as hash functions, message authentication codes, encryption or digital signatures. However, many of them condemn the use of asymmetric cryptography [21, 50] and refer to its high computational cost, which can be a death blow on devices with low CPU power like mobile phones or personal digital assistants.

In this thesis, we will examine the use of asymmetric cryptography in state-of-the-art communication protocols for wireless ad-hoc networks. We will do this using a charging protocol proposed by Lamparter, Paul, and Westhoff [30] as an example for such protocols. A Sharp Zaurus personal digital assistent equipped with a 206MHz StrongARM CPU will serve as a reference platform for typical devices in ad-hoc networks. Starting with the determination of key sizes appropriate for this particular application, we will evaluate the performance of different digital signature schemes in this ad-hoc network protocol. In particular, we will not only determine which scheme allows the fastest execution times for signature generation or verification on the reference platform, but we will introduce a new measure to evaluate the performance within the protocol application.

We will develop a speed-optimized implementation of the elliptic curve digital signature algorithm (ECDSA) on the Sharp Zaurus reference platform. The implementation will be integrated into the prototype for the charging protocol, which is currently developed by Lamparter, Paul, and Westhoff. It will be based on general elliptic curves over binary finite fields. Encouraged by the results of a performance evaluation of Koblitz curves on a PalmOS device [64], we will also provide routines that are optimized for this special group of elliptic curves.

The thesis is organized as follows.

Chapter 2 provides an overview over different types and applications of ad-hoc networks. We discuss several security problems inherent to these networks and present existing approaches to counteract the security threads.

Chapter 3 introduces the most commonly used digital signature schemes RSA, DSA and ECDSA. Besides presenting the algorithms for key generation, signature generation and signature verification, we describe the underlying computationally hard mathematical problems together with known attacks against these schemes. We also provide a brief performance comparison.

A brief description of the secure charging protocol and our discussion of the required level of security can be found in Chapter 4. We also provide a careful analysis of the performance of RSA and ECDSA when applied within the secure charging protocol.

Motivated by the results of our analysis, a brief introduction to elliptic curves is given in Chapter 5. It is followed by a description of several methods for performing efficient arithmetic on general elliptic curves over binary fields as well as on Koblitz curves. A list of known attacks against elliptic curve cryptosystems completes the chapter.

Chapter 6 contains details about our implementation of ECDSA on the Sharp Zaurus platform. In particular, we explain the software architecture and give reasons for our design decisions. Besides describing how our routines are integrated into the OpenSSL framework, we present our approaches to optimize the implementation. A detailed overview of the execution times on the target platform can be found at the end of this chapter.

In Chapter 7, we will refer to previous implementations of elliptic curve cryptosystems. Chapter 8 contains a summary of this work and a discussion of the results. Finally, manuals for our implementation and the tools we provide are given in the appendix.

2. Security in Ad-hoc Networks

For establishing security, the special properties of wireless ad-hoc networks require new approaches that differ significantly from mechanisms to secure classical networks with fixed infrastructure. Reasons for this are the ability of nodes to move freely, enter or leave the network at any time and the vulnerability of the wireless communication channel.

In this chapter, we explain what an ad-hoc network is, in which applications these types of networks are used and which are the properties of typical devices participating in such networks. Due to the dynamic structure of ad-hoc networks, the routing mechanisms differ from classical mechanisms for fixed networks and offer a number of new vulnerabilities. Therefore, we introduce two common routing protocols for such networks in Section 2.2. We close this chapter with a discussion about several security issues related to ad-hoc networks and give examples for approaches to remedy them.

2.1. Ad-hoc Networks

2.1.1. Types of Ad-hoc Networks

An ad-hoc network is a concept that has received attention in scientific research since the 1970's. Over the years, the field has developed and the applications are constantly evolving. There seems to be no universally accepted definition of such networks [58]. The term *ad-hoc networks* covers a wide range of different types of wireless networks with different applications and different requirements concerning security. In the following, we describe the two most important models between which one can distinguish [14].

The first system model is the *mobile ad-hoc network* (MANET) which is a collection of wireless mobile nodes that do not require any fixed infrastructure or centralized administration for communication. However, since the transmission range of each node is limited to each other's proximity, data for out-of-range nodes is routed through intermediate nodes. The mobile nodes are not bound to any centralized control like base stations or mobile switching centers. This offers unrestricted mobility and connectivity to the users, but network management is now entirely up to the nodes that form the network. Each node operates not only as host but also as router, since it may be necessary to forward packets of other nodes in the network that may not be within direct wireless transmission range of each other. A MANET is formed instantaneously, and the mobile nodes dynamically establish routing among themselves. This can be extremely useful in situations where geographical or terrestrial constraints demand a totally distributed, flexible

network without any fixed infrastructure, such as battlefields, military applications, and other emergency and disaster situations. A future application of wireless MANETs is for example a sensor network, which consists of several thousand small low-powered nodes with sensing and communication capabilities [32, 49]. One could, for example, imagine a battlefield or disaster area where small sensors are dropped from a plane. The sensors establish a wireless network, thereby bridge over areas without communication infrastructure and also gather information about the area. Another application of MANETs is the scenario where a group of people meets in a conference room and wants to establish a network among their personal digital assistants (PDAs) or portable computers.

The second system model is a wireless network with a fixed backbone. These networks consist of a large number of mobile nodes and relatively fewer, but more powerful, fixed nodes. A fixed node and a mobile one communicate via the wireless medium, but a fixed permanent infrastructure is still required. In single-hop wireless networks mobile nodes can only communicate directly with a fixed node and communication between two mobile nodes is carried out via fixed nodes that are connected by a permanent link. Examples for such networks are cellular phone systems. However, those networks have some disadvantages. Mobile nodes that are not within range of a fixed node cannot communicate at all even if they are close enough to other mobile nodes. Moreover nodes may have to spend a lot of energy to reach the next fixed node just in order to communicate with a node that is right next to them. Multi-hop wireless networks in which a node can communicate directly with a fixed node, but also with other mobile nodes within the range of the wireless link, can overcome these disadvantages. In [35] the authors show that the throughput of multi-hop networks is superior to that of single-hop networks. A similar architecture is described in [25]. Examples for such networks could be Internet access points at airports or trade fairs, but also a household with a wireless access point and several home appliances equipped with wireless communication interfaces.

2.1.2. Typical Devices

Purely mobile ad-hoc networks as well as ad-hoc networks with fixed backbone have in common that the mobile nodes usually are heavily constrained devices. As the user wants such devices to be cheap, small, lightweight and easy to handle (*anytime, anywhere*), they are often battery-powered and have limited CPU power. The size of the available memory is also limited. The wireless link is bandwidth-constrained and consumes a lot of battery power, therefore the transmitting range is often not more than a few ten meters.

Due to these constraints communication protocols and software must be carefully designed and implemented. The wireless communication channel and the limited resources lead to a number of new security threads inherent to ad-hoc networks that must also be taken care of. The constrained resources require optimized implementations with low overhead that save CPU cycles and memory.

2.2. Routing in Wireless Ad-hoc Networks

As mentioned before, ad-hoc networks offer a number of new vulnerabilities of which some are caused by the different routing mechanisms employed in ad-hoc networks. The special structure of these networks requires the routing mechanisms to be dynamic and on-demand. Routes have to be adapted to the constantly changing network topology. Moreover, routing is not only performed by special nodes, but by every member of the network. Before we introduce the vulnerabilities of ad-hoc networks in the following section, let us briefly present two commonly used routing protocols for ad-hoc networks, namely DSR and AODV. This will allow the reader to better understand the cause of some of the vulnerabilities mentioned in the remainder of this chapter.

2.2.1. Dynamic Source Routing Protocol (DSR)

DSR [23] is an on-demand source routing protocol. It is referred to as "on-demand" because route paths are discovered at the time a source sends a packet to a destination for which the source has no path.
How is a route discovered? Suppose a node S wishes to communicate with a node D but does not know any path to D. It initiates a route discovery by sending a *route request* broadcast to its neighbors. This packet contains the destination address D. The neighbors append their own address to the *route request* packet and rebroadcast it. The process continues until the *route request* reaches D. D now sends back a *route reply* packet to S to inform it about the discovered route. D may choose the reverse path (all nodes on the path the *route request* packet traveled have been appended to the packet) or initiate a new route discovery back to S. A source may receive multiple route replies from a destination, because there may be many routes from S to D. These routes are cached for future use.
What happens if a link breaks? When two nodes are no longer within transmission range, the link is broken, and if an intermediate node detects such a link break when forwarding a packet to the next node on a path, it will send a message to the source that the link is broken. The source then tries one of the cached alternative paths or if it does not have any alternatives, it will initiate another route discovery [39].

2.2.2. Ad-hoc On-demand Distance Vector Routing Protocol (AODV)

The AODV [13] protocol is a table-driven routing protocol and it is based on the classical Bellman-Ford routing algorithm.
How is a route discovered? When a source node S wants to send a packet to a destination D and does not already have a route, it broadcasts a *route request* packet across the network. Nodes that receive this packet update their information for the source node and their routing tables. If the node is either the destination S or knows a recent path to S, it may send a *route reply* back to the source. Otherwise, the *route request* is rebroadcasted.

As the *route reply* propagates back to the source, the nodes on the way update their routing tables with the information about the route to D.
Routes are maintained as long as data packets are traveling periodically from the source to the destination. Once the source stops sending packets, the link will time out and eventually be deleted from the routing tables of the intermediate nodes. If a link break is detected by an intermediate node, a *route error* message is propagated back to S. If it still desires the route, it has to reinitiate a route discovery [1].
AODV typically minimizes the number of required broadcasts, i.e. nodes that are not on a selected path do not maintain routing information or participate in routing table exchanges [14].

2.3. Security

Due to the dynamically changing topology and infrastructureless, decentralized characteristics, security is hard to achieve in mobile ad-hoc networks. Hence, security mechanisms have to be a built-in feature for all sorts of ad-hoc network based applications. In this section, we talk about the security objectives for designing applications, possible attacks against ad-hoc networks and countermeasures that have already been proposed.

2.3.1. Security Goals

To secure a network, one usually considers the objectives *availability*, *confidentiality*, *integrity*, *authentication* and *non-repudiation* [45, 61, 68, 62]. In this subsection, we shortly explain the meaning of these terms.

Availability

Availability is a requirement that assures that systems work promptly and service is not denied to authorized users. Attacks that affect availability are called *denial of service* (DoS) attacks. In an ad-hoc network, denial of service attacks could be launched at any layer. An adversary could employ jamming to interfere with communication on the physical layer. On the network layer, an adversary could disrupt the routing protocol and disconnect the network. An adversary could also bring down high-level services. By massivly communicating with a node an attacker could prevent its victim from switching to power save mode and thereby drain the victim's battery.

Confidentiality

Confidentiality is the requirement that private or confidential information is not disclosed to unauthorized entities. The transmission of sensitive information, such as strategic or

tactical military information, but also information about financial transactions, requires confidentiality. Not only the high-level information must remain confidential, but also routing information is not to be disclosed, as it might be valuable to identify and locate targets in a battlefield or to create an information profile of a particular user.

Integrity

Integrity is the property that data has not been altered in an unauthorized manner while being transferred. To assure integrity, unauthorized manipulation must be detectable. Due to the wireless communication interface a message in an ad-hoc network could be corrupted by benign failures, such impairment of radio propagation, or because of malicious attacks on the network. It is often not easy to distinguish between those two reasons for altered data, therefore many countermeasures that work for fixed infrastructure networks (e.g. establishing a rating level for the trustworthiness of a node), cannot be easily applied to ad-hoc wireless networks.

Authentication

Authentication enables a node to ensure the identity of the peer node it is communicating with. When transmitting confidential data, it is important that the node that receives the information is the one it is meant for. A node could for example masquerade as another node and gain unauthorized access to resources and sensitive information. In some applications of ad-hoc networks, such as portable communication devices or sensor networks for battlefield situations, the device could be tampered with, so it might not only be necessary to authenticate the device but also to make sure it has not been compromised.

Non-repudiation

Non-repudiation ensures that the originator of a message cannot deny having sent the message. This goal is of particular importance in e-commerce applications or whenever charging and billing is involved. Otherwise a user could for example order some product or service, possibly make use of it, and deny that she has ordered or received it when the provider wants to bill her. Non-repudiation is also useful for detecting compromised nodes. When a node A receives an erroneous message from a node B, non-repudiation allows A to accuse B using this message and thereby convince other nodes that B is compromised.

2.3.2. Vulnerabilities of Ad-hoc Networks

Since there exists a broad range of applications for ad-hoc networks, the range of vulnerabilities is also quite broad. In this subsection, we present some of the most important vulnerabilities, which are mentioned in [14], although they are not all relevant for our particular application.

Weak physical protection

In classical network applications, the physical protection of a node is often quite well. Servers or workstations are installed stationary in rooms that unauthorized persons cannot enter.
In military applications, e.g. where soldiers are carrying mobile devices while fighting on a battlefield or where small sensors are dropped from a plane, one can easily imagine that mobile nodes are subject to capturing, compromising and hijacking. In such hostile environments it is almost impossible to provide perfect physical protection.
Today, portable devices like mobile phones or PDAs are getting smaller and smaller. Such a small device can be easily lost or stolen and misused by an adversary. Hence, we also have to pay attention to this problem in civilian applications.
As a consequence the case of compromised devices should always be considered during the design of an ad-hoc network system.

Constrained capabilities

As mentioned before, devices in ad-hoc networks have constrained capabilities concerning CPU power, battery power and transmission bandwidth. These limited resources are subject to denial of service attacks.
Denial of service attacks against CPU power and transmission bandwidth are well known from classical networks. One or more adversaries flood a node with so many requests at a time that the node cannot process them any more. As a result, benign users cannot be served either.
In [60] Stajano and Anderson present a denial of service attack that makes use of the limited battery power of ad-hoc networking devices. They call the attack *sleep deprivation torture*. Most portable devices try to spend as much time as possible in sleep mode in order to minimize energy consumption. In fact, the radio device and the CPU consume the most power in modern devices, so they are turned on only once a while during sleep mode. An attacker might communicate with a particular node in a legitimate way just to keep it from going into sleep mode. The adversary thereby exhausts the victim's batteries much faster than usual. Finally, when it has run out of battery power, the victim is disabled. So, this attack is more powerful than DoS attacks against CPU power or transmission bandwidth, because the device is not only disabled for the time of the attack, but forever (at least until the next battery change or recharge).

Required cooperative participation

The first applications that have been proposed for ad-hoc networks take place in military or disastrous situations. In those cases all nodes usually belong to one authority and have a common goal. However, in civilian applications one can no longer assume that the nodes have a common goal. Moreover, users may act selfish and are not concerned

about overall network performance. Therefore, recent research deals with the prevention of non-cooperative behavior.
First of all, why is cooperation so important in multi-hop ad-hoc networks? To transmit a message to a node B across the network, the originating node A often has to route it via intermediate nodes, because the receiving node B is not within the transmission range of A's wireless interface. Hence, to keep the network functioning, the intermediate nodes have to spend energy, CPU power and transmission bandwidth to forward other node's messages. But as mentioned before, energy, CPU power and transmission bandwidth are limited resources for most devices in an ad-hoc network, so there is a trade-off between cooperation and survival. At the same time, if a node does not forward foreign messages, other nodes might not forward either and thereby deny service [10].

Weaknesses of the wireless medium

It is part of the nature of the wireless medium, that it is not an exclusive use medium. As a consequence, possible attacks range from passive eavesdropping to active impersonation, message replay and message distortion. Actively interfering attacks allow the adversary to delete messages, to inject erroneous messages, to modify messages and to impersonate as a node [68].

Attacks on the network layer

Due to the dynamically changing topology of ad-hoc networks (i.e. nodes join and leave, disconnect temporarily and move around), routing has to be organized in a dynamic way. This opens among others the following security vulnerabilities, which are mentioned, for example, in [10, 14, 21].

1. **Incorrect forwarding:** As mentioned in the section about cooperation, a node could, instead of following the routing protocol, deny forwarding packets. This could also be done in a selective way, i.e. forward only a set of packets, maybe for a particular group of nodes. Furthermore a node could modify other nodes' answers to route requests and influence the routing throughout the network.

2. **Traffic deviation:** A malicious node could falsely advertise very attractive routes (for example, claim that the destination is only one hop away) and thereby convince other nodes to route their messages via that malicious node. An attacker could use this to collect information, influence network routes, and prevent some packets from being transmitted.

3. **Flooding with route updates:** By sending route updates at short intervals an adversary could overload the network. This is another kind of denial of service attack.

4. **Black hole attack:** This is a combination of traffic deviation and incorrect forwarding. A node advertises falsely itself as having the shortest path to the node whose packet it wants to intercept. Now many nodes route their packets to that destination via the malicious node, because it seems to be the most efficient route. The adversary now simply drops these packets.

5. **Gray hole attack:** In this special case of the black hole attack, the attacker selectively drops some packets but not others. The adversary may, for example, forward routing packets but not data packets. Alternatively, she may only forward packets for certain destinations or from particular origins.

6. **Wormhole attack:** In [21], the authors introduce a new attack that involves a pair of attackers that are linked via a private network connection. These malicious nodes tunnel packets they receive from the network through their direct link and rebroadcast them at the other end of the wormhole. This potentially disrupts routing by short-circuiting the normal flow of routing packets and offers the possibility to control a considerable amount of traffic.

2.3.3. Approaches to Establish Security in Ad-hoc Networks

Let us now present some approaches that have been proposed to counteract the security threads of ad-hoc networks. There exists a broad range of problems from authentication and key distribution, over secure routing, detection of misbehavior to motivating cooperative behavior. In the following, we give some examples for approaches to tackle these problems.

Authentication and key distribution

- **Resurrecting duckling:** In [60] Stajano and Anderson present a mechanism to authenticate users by *imprinting*. Analogous to a duckling that recognizes the first moving subject it sees as its mother, the node accepts a symmetric encryption key from the first device that sends such a key on a secure channel, e.g. by physical contact during the device initialization. The node may be imprinted several times. After performing this kind of key exchange, an encrypted connection can be established.

- **Self-organized public key certificates:** Public-key certificates issued, stored and distributed by the users are part of a model proposed by Hubaux, Buttyán and Čapcun [22]. In this model, each node keeps a small part of the certification knowledge. By sharing this information, several certificate paths can be found.

- **Localized certification services:** Kong, Zerfos, Luo, Lu and Zhang [29] suggest a public key infrastructure with certification authorities based on threshold secret sharing. The secret shares are updated periodically. For providing certification services K one-hop neighbors are needed within a given time window.

- **Asynchronous threshold security:** Zhou and Haas [68] propose a key management service based on *threshold cryptography* to distribute trust among a set of special nodes. Cryptographic operations can only be performed jointly by $t + 1$ nodes, but are infeasible for t or less nodes, even by collusion. Furthermore, the authors take advantage of inherent redundancies in ad-hoc networks due to multiple routes to enable diversity coding. The basic idea is to transmit redundant information through additional routes for error detection and correction. Thereby Byzantine failures given by several corrupted nodes or collusions can be tolerated.

Secure routing

- **Secure Routing Protocol:** Only assuming a security association between endpoints, Papadimitratos and Haas propose a routing protocol [50] that guarantees a correct route discovery without the need of trusted intermediate nodes. Route requests reach the destination along with a unique random query identifier. The route request reply contains a message authentication code computed over the path. The protocol is constructed such that compromised or replayed route requests will never reach the source.

- **ARIADNE:** This secure on-demand routing protocol by Hu, Perrig, and Johnson [21] prevents compromised nodes from disturbing uncompromised routes, i.e. routes that consist of uncompromised nodes. It uses a key management protocol based on one-way key chains called TESLA, which relies on loosely synchronized clocks. The protocol is based on DSR (Section 2.2) and protects the routes with hash chains and message authentication codes.

- **SAODV:** In [66], a security extension for AODV (Section 2.2) has been proposed by Zapata. Its basic idea is that the source appends a digital signature and a keyed hash chain on the control messages for route discovery. As the message traverses the network, the intermediate nodes verify the signature and update the hash chain. Thereby integrity, authentication and non-repudiation for source and destination are provided.

Detection of attackers

- **Intrusion detection:** To suite the needs of wireless ad-hoc networks Zhang and Lee [67] postulate that intrusion detection and response systems should be both distributed and cooperative. With statistical anomaly detection on several network layers every node watches for intrusions and a majority voting mechanism is used to classify behavior by consensus. Possible responses to compromised nodes are re-authentication or isolation.

- **Watchdog and pathrater:** For DSR, Marti, Giuli, Lai and Baker [39] introduce a *watchdog* that detects denied packet forwarding and a *pathrater* that manages trust

and routing policies. Misbehavior such as packet dropping is detected by utilizing the promiscuous mode of the wireless interface, i.e. the ability of nodes to overhear their neighbors' communication. Successfully detected malicious nodes are avoided in future routes, however, their outgoing data packets are still forwarded to the destinations.

- **CONFIDANT:** This acronym stands for 'Cooperation Of Nodes, Fairness In Dynamic Ad-hoc NeTworks' and has been proposed by Buchegger and Boudec [8, 9, 10]. This scheme detects malicious nodes by means of observation or reports about several types of attacks. Nodes have a monitor for observations, reputation records for first-hand and trusted second-hand observations, trust records to control trust given to received warnings, and a path manager for nodes to adapt their behavior according to reputation. Malicious nodes that have been detected are isolated from the network.

Motivating cooperation

- **Nuglets or Counters:** As incentives for cooperation Buttyán and Hubaux [11] propose so-called nuglets that serve as a per-hop payment in every packet or counters [12] to encourage forwarding. Both nuglets and counters reside in a secure module in each node, are incremented when nodes forward for others and decremented when they send packets for themselves.

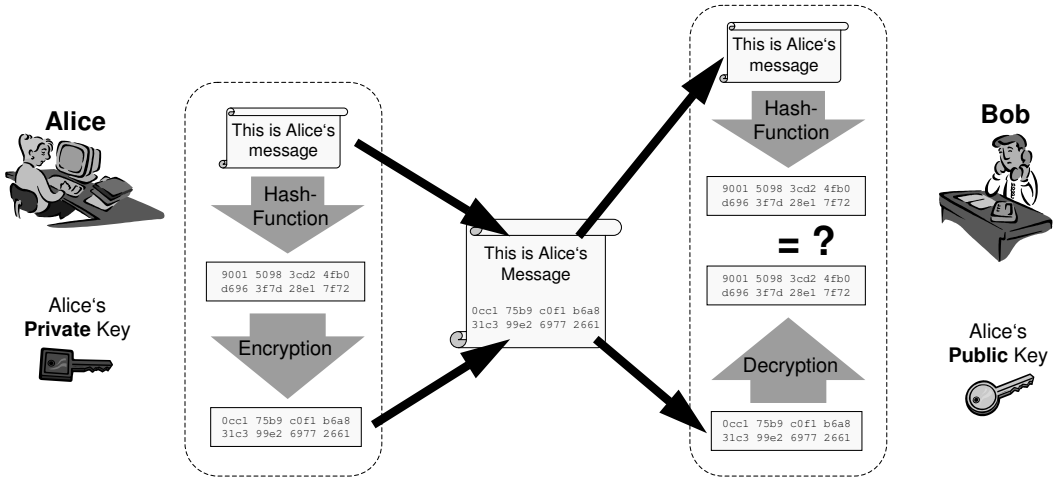
- **Secure Charging Protocol:** Lamparter, Paul, and Westhoff [30] propose a protocol that motivates intermediate nodes to forward packets by giving them a monetary reward. At the same time, nodes are charged for sending or receiving data. The envisioned ad-hoc network is not purely mobile, but it is an access network with some fixed backbone that provides for example Internet or intranet access. The protocol uses digital signatures and keyed hash chains. The certificate authority is maintained by a network service provider that also manages charging and rewarding.

3. Digital Signatures

One part of this work is the cryptographic analysis of the secure charging protocol. We briefly mentioned in the last chapter, that it employs digital signatures. For this reason, let us provide some information about the most common digital signature schemes before starting the actual analysis.

The concept of a digital signature was introduced in 1976 by Diffie and Hellman [15, 16]. Digital Signatures have been designed in order to provide the digital counterpart to a handwritten signature. Basically, a digital signature is a number that depends on some secret only known to the signer (the signer's secret key) and the content of the message to be signed. The design goal is to make the signature verifiable, i.e. an unbiased third party should be able to check, without knowing the secret of the signer, whether the message has been indeed signed by a particular person. Such a verification may be necessary when either the signer denies having signed the message (repudiation) or when an adversary has faked a signature and claims that it is valid.

Figure 3.0.1 Illustration of digital signatures.



Most digital signature schemes work as illustrated in Figure 3.0.1. Suppose Alice wants to send Bob a digitally signed message. For the sake of simplicity, let us assume that Bob already owns a copy of Alice's public key and Alice owns the corresponding private (secret) key. Moreover, we assume that Bob is sure that the key he owns is the public key of Alice and not of somebody else.
In the first step of the signature scheme, Alice calculates a hash value of her message using

some one-way hash function and converts it to an integer. A one-way hash function maps a message of arbitrary size to a bit string of predefined length. An important property of such a hash function is that it is *collision resistant*, i.e. it is computationally infeasible to find two distinct inputs that hash to the same output. In the following step, the hash value is used together with Alice's private key as input value for some mathematical function. This step is called 'Encryption' in Figure 3.0.1.
Now, Alice appends the output of this mathematical function, which is the digital signature of the original message, to her message and sends both to Bob.
Bob extracts the digital signature part from the received message and uses it as input argument of some other mathematical function. Together with Alice's public key as a second input argument, Bob calculates an integer that should be equal to the value that Alice used as input for her encryption function. This step is called 'Decryption' in the figure.
Bob uses the hash function on the message he received from Alice to calculate a hash value. By comparing this hash value to the output of his decryption function, he can detect whether the message has been altered.
If both values are equal, he knows that it has not been altered. Moreover, if Bob knows for sure that the public key he used to verify the message belongs to Alice, he can also be sure that Alice and not somebody else has sent the message, since only she owns the corresponding private key.

In the following, we introduce the RSA, DSA and ECDSA signature schemes and discuss their underlying mathematical problems and possible attacks.

3.1. The RSA Signature Scheme

The RSA signature scheme was discovered by Rivest, Shamir, and Adleman [51]. It was the first practical signature scheme based on public-key techniques. The underlying computationally hard mathematical problem for the RSA signature scheme is the *integer factorization problem (IFP)*[41]:

> Given a composite number n that is the product of two large prime numbers p and q, find p and q.

Since it is part of the public key, an adversary has access to the modulus n. Once p and q are computed, the system is broken and the attacker can use Algorithm 3.1.1 to determine the secret key.

Algorithm 3.1.1 summarizes how a key pair, i.e. a public and the corresponding private key, for the RSA signature scheme can be generated. The steps for signing a message are given in Algorithm 3.1.2 and the steps for verifying a message can be found in Algorithm 3.1.3.

Algorithm 3.1.1 Key generation for the RSA signature scheme [45]

OUTPUT: The public key (n, e) and the private key d.

1: Generate two large distinct random primes p and q, each roughly the same size.
2: Compute $n = pq$ and $\phi = (p-1)(q-1)$.
3: Use the extended Euclidean algorithm to compute the unique integer d, $1 < d < \phi$, such that $ed \equiv 1 \pmod{\phi}$.
4: The public key is (n, e); the private key is d.

Algorithm 3.1.2 RSA signature generation [45]

INPUT: The message m, the public key (n, e), and the private key d.
OUTPUT: The digital signature s.

1: Compute the hash value of the message $\tilde{m} = h(m)$, an integer in the range $[0, n-1]$.
2: Compute $s = \tilde{m}^d \mod n$.
3: The signature for m is s.

Algorithm 3.1.3 RSA signature verification [45]

INPUT: The message m, the public key (n, e) of the signer and the signature s on m.

1: Compute $\tilde{m} = s^e \mod n$.
2: Verify that $\tilde{m} = h(m)$; if not, reject the signature.

In [41] the following attacks for the IFP are mentioned:

- **Continued Fraction Algorithm:** This algorithm is based on the idea of using a factor base of primes and generating an associated set of linear equations whose solution leads to a factorization. It could factor numbers of up to 133-bits.
- **Quadratic Sieve Algorithm (QS):** This algorithm is based on the same idea as the continued fraction algorithm and can be easily parallelized to permit factoring on distributed networks.
- **General Number Field Sieve (NFS):** Also based on the idea of the continued fraction algorithm, the NFS is supposed to be the fastest known algorithm for factoring integers having at least 400 bits.
- **Elliptic Curve Factoring Method (ECM):** This algorithm attempts to exploit special features of an integer to be factorized. It tends to find small factors first.

3.2. The Digital Signature Algorithm (DSA)

The Digital Signature Algorithm has been proposed in August of 1991 by the U.S. National Institute of Standards and Technology (NIST). The underlying computationally hard mathematical problem is the *discrete logarithm problem (DLP)*[41]:

> Given a prime p, a generator α of $\mathbb{Z}_p$, and a non-zero element $\beta \in \mathbb{Z}_p$, find the unique integer $l, 0 \leq l \leq p-2$, such that $\beta \equiv \alpha^l \pmod{p}$.

The integer l is called the *discrete logarithm* of β to the base α. If p is a prime number, then $\mathbb{Z}_p$ is a finite field denoted by the set of integers $\{0, 1, 2, \ldots, p-1\}$, where addition and multiplication are performed modulo p. There exists a non-zero element $\alpha \in \mathbb{Z}_p$ such that each non-zero element in $\mathbb{Z}_p$ can be written as a power of α; such an element α is called a *generator* of $\mathbb{Z}_p$.

Algorithm 3.2.1 summarizes how a key pair, i.e. a public and the corresponding private key, for the DSA signature scheme can be generated. The steps for signing a message are given in Algorithm 3.2.2 and the steps for verifying a message can be found in Algorithm 3.2.3.

Algorithm 3.2.1 Key generation for the DSA [45, 24]

OUTPUT: The public key (p, q, α, y) and the private key a.

1: Select a 160-bit prime q and a 1024-bit prime p with the property that $q|p-1$.
2: Select a generator α of the unique cyclic group of order q in $\mathbb{Z}_p^*$, i.e. select an element $g \in \mathbb{Z}_p^*$, compute $\alpha = g^{(p-1)/q} \mod p$, and repeat this if $\alpha = 1$.
3: Select a random integer a such that $1 \leq a \leq q-1$.
4: Compute $y = \alpha^a \mod p$.
5: The public key is (p, q, α, y); the private key is a.

Algorithm 3.2.2 DSA signature generation [45]

INPUT: The message m, the public key (p, q, α, y), and the private key a.
OUTPUT: The digital signature (r, s).

1: Select a random integer $k, 0 < k < q$.
2: Compute $r = (a^k \mod p) \mod q$.
3: Compute $k^{-1} \mod q$.
4: Compute $s = k^{-1}\{h(m) + ar\} \mod q$, where $h(m)$ is the hash value of the message m.
5: The signature for m is (r, s).

Algorithm 3.2.3 DSA signature verification [45]

INPUT: The message m, the public key (p, q, α, y) of the signer and the signature (r, s) on m.

1: Verify that $0 < r < q$ and $0 < s < q$; if not, reject the signature.
2: Compute $w = s^{-1} \mod q$ and the hash value $h(m)$.
3: Compute $u_1 = w \cdot h(m) \mod q$ and $u_2 = rw \mod q$.
4: Compute $v = (\alpha^{u_1} y^{u_2} \mod p) \mod q$.
5: Accept the signature if and only if $v = r$.

According to [41], there exist the following known attacks against DSA and the discrete logarithm problem:

- **Index Calculus Method:** The fastest general-purpose algorithms known for solving the DLP are based on the index calculus method. In this method, a database of small primes and their corresponding logarithms is constructed. Subsequently, logarithms of arbitrary field elements can be easily obtained. This is reminiscent of the factor base methods for integer factorization. If an improvement in the algorithms for either the IFP or DLP is found, then shortly after this a similar improved algorithm can be expected to be found for the other problem. The index calculus method can be easily parallelized.
- **Number Field Sieve Algorithm:** This is the best current algorithm known for the DLP and has precisely the same asymptotic running time as the corresponding algorithm for factoring integers.

3.3. The Elliptic Curve Digital Signature Algorithm (ECDSA)

A variant of DSA based on elliptic curves is ECDSA. It was first proposed in 1992 by Scott Vanstone [63]. The underlying computationally hard mathematical problem is the *Elliptic Curve Discrete Logarithm Problem* (ECDLP)[41]:

> Given an elliptic curve E defined over $\mathbb{F}_q$, a point $P \in E(\mathbb{F}_q)$ of order n, and a point $Q \in E(\mathbb{F}_q)$, determine the integer $l, 0 \leq l \leq n-1$, such that $Q = lP$, provided that such an integer exists.

This discrete logarithm problem over elliptic curves is considered to be significantly harder than the DLP over $\mathbb{Z}_p$, which is the mathematical basis for DSA. Therefore, the strength-per-key-bit is substantially higher than in DSA and, hence, smaller parameters (keys) can be used for elliptic curve cryptosystems to achieve equivalent levels of security.

In order to facilitate interoperability, the domain parameters for ECDSA, which are the parameters of the curve E, the underlying finite field $\mathbb{F}_q$ and a base point $G \in E(\mathbb{F}_q)$, have to be negotiated and agreed upon by the communication partners. The curve is usually determined by its two parameters a and b and the curve equation. For the finite field $\mathbb{F}_{2^m}$, the curve equation is given by the equation

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

which is the same for all m. The base point G is defined by its affine coordinates x_G and y_G. Usually, the order n of the point G is also part of the domain parameters.

Algorithm 3.3.1 summarizes how a key pair, i.e. a public and the corresponding private key, for the ECDSA signature scheme can be generated. The steps for signing a message are given in Algorithm 3.3.2 and the steps for verifying a message can be found in Algorithm 3.3.3. Signature generation and verification requires the computation of the hash value of the message using the Secure Hash Algorithm (SHA-1) [45], which was proposed by the U.S. National Institute for Standards and Technology (NIST).

Algorithm 3.3.1 Key generation for the ECDSA [24]

INPUT: The elliptic curve domain parameters.
OUTPUT: The public key Q and the private key d.

1: Select a random integer d in the interval $[1, n-1]$.
2: Compute $Q = dG$.
3: The public key is Q and the private key is d.

Algorithm 3.3.2 ECDSA signature generation [24]

INPUT: The message m, the elliptic curve domain parameters, the public key Q, and the private key d.
OUTPUT: The digital signature (r, s).

1: Select a random integer $k, 0 < k < n$.
2: Compute $kG = (x_1, y_1)$ and convert x_1 to an integer.
3: Compute $r = x_1 \mod n$. If $r = 0$ then go to Step 1.
4: Compute $k^{-1} \mod n$.
5: Compute SHA-1(m) and convert this bit string to an integer e.
6: Compute $s = k^{-1}(e + dr) \mod n$. If $s = 0$ then go to Step 1.
7: The signature for the message m is (r, s).

Algorithm 3.3.3 ECDSA signature verification [24]

INPUT: The elliptic curve domain parameters, the message m, the public key Q of the signer and the signature (r, s).

1: Verify that r and s are integers in the interval $[1, n-1]$
2: Compute SHA-1(m) and convert this bit string to an integer e.
3: Compute $w = s^{-1} \mod n$.
4: Compute $u_1 = ew \mod n$ and $u_2 = rw \mod n$.
5: Compute $X = u_1 G + u_2 Q$.
6: If $X = \mathcal{O}$, then reject the signature. Otherwise, convert the x-coordinate of X to an integer x_1, and compute $v = x_1 \mod n$.
7: Accept the signature if and only if $v = r$.

A discussion about possible attacks on elliptic curve cryptosystems and their level of security can be found in Section 5.5.

3.4. Performance Comparison of RSA, DSA and ECDSA

Before comparing the performance of different signature schemes, i.e. the execution times of the signature generation and signature verification, we have to agree which key sizes provide a comparable level of security. This is necessary, because the computational hardness of the underlying mathematical problems is different and some schemes need smaller key sizes than others for achieving the same level of security.

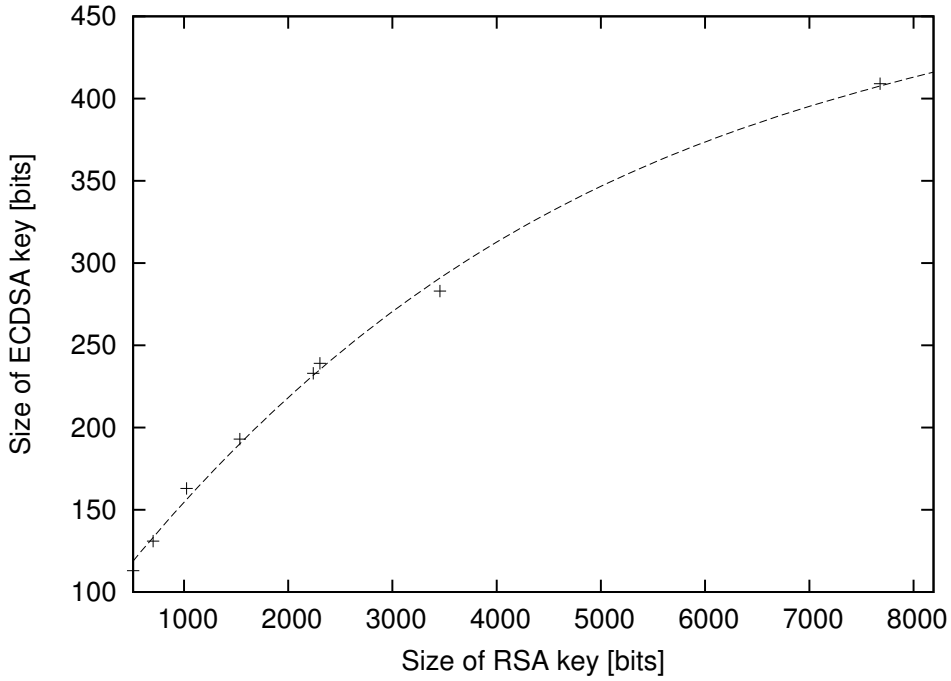
In a Standards for Efficient Cryptography document [44] a list of elliptic curves with different key sizes is given. The list also contains RSA / DSA key sizes that achieve comparable levels of security. Table 3.4.1 contains these values.

Clearly, the key sizes for elliptic curves are significantly smaller than those for RSA / DSA. Additionally, the key size does not increase as fast as the one of RSA / DSA. Figure 3.4.1 gives a good impression of this behavior. Hence, ECDSA has a major advantage for designs that might need an increased level of security in the future.

Lenstra and Verheul present in their paper [33] a recommendation of key sizes for symmetric cryptosystems, RSA, and discrete logarithm based cryptosystems both over finite fields and over groups of elliptic curves over prime fields. The authors formulate a model that based on Moore's law incorporates future changes of the available computational power and the arising hardware costs. Their model also takes into account future progress in cryptanalysis. Based on some hypotheses, e.g. that DES has been secure enough for commercial applications until 1982, the authors come up with an equation that predicts the computational load they consider to be infeasible until year y. The authors combine their model with data points that evaluate the computational power necessary to break

certain cryptosystems with certain key sizes. Finally, Lenstra and Verheul present a table stating for each year until 2050 which minimum key size for which cryptosystem can be considered to be secure until that year.

Figure 3.4.1 Size of the ECDSA key compared to the size of the RSA / DSA key providing a similar level of security



Lenstra and Verheul tend to recommend key sizes for elliptic curve cryptosystems that are smaller than those in the standard document cited above. As it is dated before the standard document and as the authors do not directly compare elliptic curve cryptosystems over binary fields, we prefer to use Table 3.4.1 as basis.

In a whitepaper [40] from 1997, Certicom Corp. examines RSA and elliptic curve based cryptosystems with respect to their security and efficiency. They point out that the best known general-purpose algorithm for breaking RSA has a sub-exponential time complexity whereas solving the elliptic curve discrete logarithm problem with the best general algorithm has fully exponential time complexity. For increasing levels of security, the gap between ECC key sizes and RSA key sizes dramatically increases. In terms of efficiency, Certicom claims that ECC outperforms RSA with respect to computational overhead, storage requirements, and bandwidth requirements. Their benchmarks are based on 160-bit ECC and 1024-bit RSA. To their opinion implementations of elliptic curve cryptosystems that are roughly 10 times faster than RSA can be realized. However, they also concede that a short public exponent in RSA may lead to signature verification times that are comparable with ECC.

Table 3.4.1 SECG recommended elliptic curves over $\mathbb{F}_{2^m}$ [44]

Curve Name	Curve Type	Key Size	equiv. RSA / DSA Key Size
sect113r1 sect113r2	random	113	512
sect131r1 sect131r2	random	131	704
sect163k1	Koblitz	163	1024
sect163r1 sect163r2	random	163	1024
sect193r1 sect193r2	random	193	1536
sect233k1	Koblitz	233	2240
sect233r1	random	233	2240
sect239k1	Koblitz	239	2304
sect283k1	Koblitz	283	3456
sect283r1	random	283	3456

In an RSA Laboratories Technical Note [52] from 1997, Robshaw and Yin analyze cryptosystems based on RSA and on ECDSA. For ECDSA with a key size of 160 bits and 1024-bit RSA, which they found to achieve a comparable level of security, the authors compare the performance of both schemes in terms of storage requirements and computational speed. While its short key size leads to clear advantages for ECDSA with respect to storage requirements, their findings for the computational speed do not allow such a clear statement. According to their benchmarks, the RSA sign operation is about 7 times slower than the one of ECDSA, but the verify operation is more than 6 times faster. Robshaw and Yin fear that elliptic curve cryptography might still offer some yet undiscovered loopholes due to the complex mathematical theory behind it.

Another article [65] by Wiener published in RSA Laboratories' CryptoByte newsletter contains a comparison between 1024-bit RSA, 1024-bit DSA and 168-bit ECDSA. The verification times of RSA are found to be more than 30 times faster than those of ECDSA. The signature generation is measured to be around 8 times slower (see Table 3.4.2 for the exact results). The author points out that the optimal choice of a signature scheme depends on the particular application. Discussing several different applications for public-key cryptosystems, he comes to the conclusion that RSA is well suited, for example, for certificate-based systems that require only few signature generations but thousands of signature verifications. However, in wireless communication scenarios Wiener favors ECDSA as public-key algorithm, because the short key size and low signature overhead save transmission bandwidth and lead to smaller silicon implementations.

Table 3.4.2 Digital signature timings (milliseconds on a 200 MHz Pentium Pro) [65].

	RSA-1024 (e = 3)	DSA-1024	ECDSA-168 (over $\mathbb{F}_{168}$)
Sign	43	7	5
Verify	0.6	27	19
Key generation	1100	7	7

An evaluation of the performance of ECC in protocol applications can be found in [18]. Besides adding ECDSA support to OpenSSL (see also Section 7), the authors analyze the performance influence of ECDSA and RSA on the SSL protocol. Their measures are the *Handshake Crypto Latency*, which is essentially the sum of the times the client and the server spend doing public key operations, and the *Server Crypto Throughput*, which is the rate at which the server can perform the cryptographic operations. For a security level of 1024-bit RSA, ECC performed more than five times better in terms of Server Crypto Throughput. However, in terms of Handshake Crypto Latency, the performance depends on the underlying scenario. For a PDA talking to a server, RSA beats ECC, while for PDA talking to PDA or server talking to server, ECC is nearly twice as fast as RSA. In experiments with a security level of 2048-bit RSA, ECC always outperformed RSA. For this reason, the authors see a performance advantage for ECC at higher levels of security.

Obviously, the opinions about which digital signature scheme is the best are not clear. However, most sources agree that RSA would be a good choice for systems in which signature verification dominates execution times and ECDSA for systems in which signature generation dominates execution times.

4. The Secure Charging Protocol (SCP)

Let us now take a more detailed look at one particular approach to enhance security in ad-hoc networks. In [30], Lamparter, Paul, and Westhoff proposed a protocol for charging support in multi-hop ad-hoc networks. The so-called Secure Charging Protocol employs digital signatures as cryptographic tool to establish authentication, integrity, and non-repudiation of charging information.

In this chapter, we point out the benefits of this protocol and describe the protocol architecture. Afterwards, we analyze the protocol from a cryptanalytic point of view and assess the required level of security as well as different digital signature schemes.

4.1. Underlying Scenario

First, let us summarize the scenario Lamparter et al. envisioned for their proposal:

The ad-hoc network is not purely mobile, but it is an access network with some fixed backbone that provides for example Internet or intranet access (see also Section 2.1.1). An access point (AP) forms the gateway between both networks. When a mobile node (MN) wants to communicate with a corresponding node (CN) in the fixed network, the AP may or may not be within transmission range. If it is not within range, the MN may use intermediate nodes (N_i) in a hop-by-hop fashion to reach the access point. When the corresponding node is within the ad-hoc network, AP can be bypassed and the CN can be reached in multi-hop mode.

The proposed protocol does not handle colluding attacks, i.e. any attack for which the mobile node MN or corresponding node CN and one or more intermediate nodes have to cooperate shall not be prevented.

4.2. Benefits

The concept of multi-hop ad-hoc networks offers several advantages over single-hop networks. Nevertheless, there are also some problems to be solved before such networks can be deployed. The protocol proposed by Lamparter, Paul, and Westhoff tackles the following two difficulties.

On first sight, a network service provider (NSP) is not interested in deploying multi-hop networks, because commonly users pay the NSP for providing the communication

infrastructure. However, in a multi-hop network, users may communicate directly with each other without using any infrastructure provided by the NSP. Obviously, the amount of traffic that is transmitted via the infrastructure of the NSP is less in a multi-hop network than in a single-hop network. Consequently, the main source of income of the NSP is endangered by such network structures.

On the other hand, why should a node forward other nodes' data packets? Mobile devices have only limited battery power and they usually try to spend as much time as possible in sleep mode in order to save energy. However, whenever the device has to communicate via its wireless interface, the power consumption rises dramatically. Forwarding foreign traffic would obviously reduce the time such a device can spend in sleep mode, simply because it has to transmit more data.

Lamparter, Paul, and Westhoff come up with an idea that solves both problems. They suggest charging the nodes for any transmitted or received data packet, regardless whether it is transmitted with or without using the infrastructure of the NSP. On the other hand, the nodes that forward foreign traffic are given a monetary reward. If the ratio between the charged and the rewarded amount of money is properly balanced, the NSP is able to earn money even if its infrastructure is not used. In this case, the users pay the NSP for providing certificate authority, network administration services and a reliably functioning ad-hoc network. Thereby, the NSP is more likely to deploy such a network, because the reduced number of necessary access points reduces spendings and billing of the users increases income. At the same time, users are encouraged to participate in forwarding, since this reduces their cost for transmitting information – of course with the disadvantage of increased power consumption.

Motivating users to cooperate and share resources as Lamparter, Paul, and Westhoff suggest is a striking approach to prevent dishonest passive behavior. This reduces the need to detect and sanction non-cooperating nodes and allows detection schemes to focus on the few actively malicious nodes.

4.3. Description

In the following, we present a shortened version of the description of the secure charging protocol architecture as it is provided by the authors of [30]. Their paper contains further details, in particular about authentication, charging and pricing.

When the *MN* joins the access network it first authenticates with the NSP. The *MN* provides its credentials to the NSP, the NSP verifies the home domain of *MN*, and sends back authorization information to *MN*.

At the source node *MN*:

1. *Path finding*: The *MN* uses some on-demand source routing protocol (see also Section 2.2) to find the path *MN*, N_1, N_2, ..., N_n, *CN*.

2. *Providing MN's legal registration*: *MN* sends the following security information along with the data:

   a) *To secure originator, destination and hops*: *MN*'s digital signature on the route *MN*, N_1, N_2, ..., N_n, *CN*,

   b) *To initialize a hash chain*: A keyed hash value on *MN* and *CN*,

   c) *To reveal key information*: An identity certificate of *MN* to prove registration to intermediate nodes.

At each intermediate node N_i:

1. *Service provision*: N_i checks the signature of *MN*. In case of a correct signature, N_i can be sure that an authorized *MN* is willing to communicate.

2. *Hash chain update*: N_i computes the next value of the hash chain, i.e. the keyed hash value of the received hash value using N_i's key.

3. *Packet forwarding*: N_i forwards the packet to the next node on the route.

At the last intermediate node N_n (in addition to the previously mentioned usual procedure at intermediate nodes):

1. *Service provision confirmation*: N_n acquires step-by-step a non-repudiative message from *CN*, confirming the received amount of data.

2. *Notification of AP*: N_n (later on) notifies the *AP* about the involved forwarding nodes and the service provision confirmation.

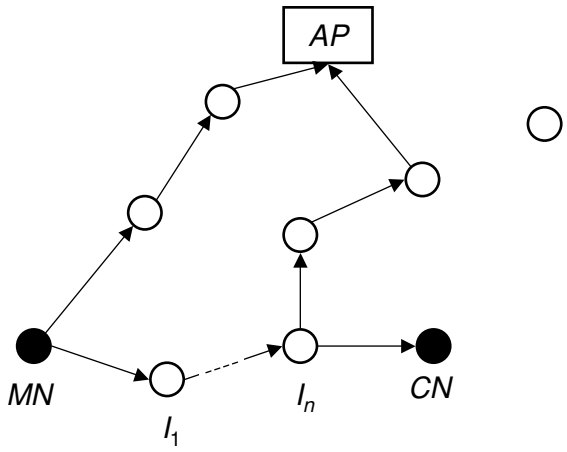
At the destination node *CN*:

The destination node *CN* signs the amount of data received from N_n.

At the access point *AP*:

Service provision verification: When it receives the hash chain and the service provision confirmation, the *AP* verifies the participation of each node and books the reward/cost to the accounts of the participating nodes.

Figure 4.3.1 Network scenario for the Secure Charging Protocol



4.4. Analysis of the Cryptographic Requirements

In their paper, Lamparter, Paul, and Westhoff proposed several cryptographic primitives for their protocol. In this section, we have a closer look at the cryptographic requirements of the protocol. The main design goals for the protocol are motivated by the constrained capabilities of the target platform. In general, cryptographic operations are said to be rather expensive with respect to execution time, which is in particular problematic due to the constrained CPU power of a portable device. Therefore, one goal is to choose cryptographic primitives that require only a small amount of CPU power. Another goal is certainly to choose primitives that keep the protocol overhead as small as possible, because the bandwidth of the wireless communication channel is also quite limited.

The cryptographic primitives proposed by Lamparter, Paul, and Westhoff in [30] for the secure charging protocol are

- unkeyed hash SHA-1,
- keyed hash MD5-MAC,
- digital signatures based on elliptic curves (ECDSA).

The hash functions turn out to be rather cheap with respect to CPU time compared to the digital signature operations. According to the OpenSSL `speed` program the computation of MD5-MAC requires approximately 250 nanoseconds and the RSA verify operation takes more than 10 milliseconds on an Intel Pentium II Workstation at 300MHz (factor $4 \cdot 10^4$). Therefore we will only consider different digital signature schemes in the following argumentation and neglect the influence of the hash functions.

4.4.1. Required Level of Security (Key Size)

A typical application scenario for the secure charging protocol is the provision of wireless Internet access at an airport terminal or railway station. These places have in common that the usual user does not stay longer than several hours at that place. Consequently, most user certificates and with them private keys and public keys need only to be valid for a short time, e.g. 24 hours. In case of longer stays, one could simply perform a key exchange every 24 hours and provide fresh certificates.

The objective of someone who attacks the secure charging protocol is certainly to get uncharged network access or to only collect the rewards without paying. However, the amount of money involved can be expected to be rather low. Today's (March 2003) prices for wireless network access range from 3.75 € per hour at the Munich airport [4] to 9 € per hour at the CeBIT fair [20]. Thus, the financial harm of successfully breaking one pair of keys is less than 200 €. Consequently, one can expect that an attacker will not spend a large amount of money to break a key. Note, that breaking a key of the secure charging protocol can only be used to compromise the node to which the key belongs. No secret information will be revealed; the attacker may forge only the secure charging protocol signatures of the compromised node.

Using the results of the DES Challenge III [53] as a basis, we can determine the key size for our purposes according to the recommendations of Lenstra and Verheul [33]. In the DES Challenge III launched in January 1999 by RSA Data Security, Inc., a message encrypted with the 56-bit DES algorithm has been broken within 22 hours and 15 minutes. As explained above, keys that can be broken in approximately this amount of time still provide enough security for our purposes. In their paper, Lenstra and Verheul offer equations to adapt their recommendations to this level of security. Table 4.4.1 contains these modified recommendations.

Table 4.4.1 Lower bound for RSA key sizes assuming that DES can be trusted until 1998 (as recommended by Lenstra and Verheul in [33]).

Year	RSA Key Size
2003	744
2005	810
2010	990
2015	1191
2020	1416
2025	1664
2030	1937

However, we feel that these recommendations are too careful. So far, the largest RSA Challenge Number that has been factored in RSA Security's Factoring Challenge [43] is

a 512-bit number. The factoring was finished in August 1999, took 3.7 months and the CPU-effort is estimated to approximately 8000 MIPS years. We feel that this margin of security would still be sufficient for our particular application. Lenstra and Verheul recommend a minimum key size of 513 bits for RSA in the year 1986, so we obtained the values in Table 4.4.2 by taking their recommendations for the year $1986 + x$ as guide value for our implementation in the year $1999 + x$. To our opinion this is still a correct interpretation of their recommendations, simply with a different margin of security. Note, that we do not claim that breaking cryptosystems that use the key sizes in Table 4.4.2 is infeasible. However, breaking such cryptosystems in less than 24 hours will require an amount of computational resources that is disproportionate to the financial gain a successful attacker might receive in our application.

Table 4.4.2 Lower bound for RSA key sizes based on the model of [33] and assuming that 512-bit RSA provided a sufficient level of security until 1999.

Year	RSA Key Size
1999	513
2003	622
2005	682
2010	844
2015	1028
2020	1235
2025	1464
2030	1717

Now that we have a guide for the RSA key sizes, we can use the table published in the SEC standard document [44] to derive the corresponding ECDSA key sizes. Table 4.4.3 contains different key sizes listed in the standard document and the year until which they prospectively provide sufficient security for the digital signature within the secure charging protocol. Note, that these key sizes are based on the assumptions made in [33]. In particular, we cannot fully anticipate the effects of future progress in cryptanalysis. The progress in this area should therefore be continously monitored and the key sizes should be adapted when necessary.

In November 2002, Certicom announced that the ECCp-109 challenge has been solved using a large network of 10,000 computers within 549 days [42]. This cryptosystem is considerably weaker than an elliptic curve cryptosystem over a 113-bit binary field. Hence, the level of security that the key sizes in Table 4.4.3 provide is probably still greater than necessary for our application.

Table 4.4.3 Overview of different key sizes for RSA and ECDSA together with the estimated year until which they prospectively provide sufficient security for the secure charging protocol (Note, that these recommendations are subject to future progress in cryptanalysis).

Year	RSA Key Size	ECDSA Key Size (for curves over $\mathbb{F}_{2^m}$)
1999	512	113
2006	704	131
2015	1024	163
2026	1536	193
2039	2240	233

4.4.2. Optimal Digital Signature Scheme

Having determined the required level of security, let us now evaluate which digital signature scheme is the best choice for the charging protocol. However, before we can start comparing different signature schemes, we have to come up with a proper measure that evaluates the performance of the schemes in a reasonable way. Previous performance comparisons suggest to compare, for example, the level of security per bit key size, the speed for generating or verifying a signature [65] or the sizes of the actual signatures. Although these measures may give a first hint for protocol designs, we feel that state-of-the-art communication protocols require more sophisticated measures that take also into account the individual features of the protocol. It is straightforward to see that the execution times for signature verification are a lot shorter for RSA than for ECDSA. On the other hand, the execution times for RSA signature generation are also a lot longer than those of ECDSA. In applications where not only one but both types of operations are used on the same platform, however, one has to come up with a different measure that takes into account the execution times of both operations.

Nevertheless, let us begin with an examination of the influence of the signature schemes RSA and ECDSA on the protocol data overhead. Afterwards, we will present our new application-oriented measure and discuss the optimal choice with respect to this approach.

Optimal choice with respect to data overhead

Since the transmission bandwidth of the network is limited and nodes usually do not want to waste too much energy for transmission, the amount of protocol data transmitted should be kept as small as possible. The following sizes have direct influence on the size of the packet transmitted from the mobile node *MN* to the corresponding node *CN*, because as stated in Section 4.3, every packet carries *MN*'s signature. Additionally, the first packet in a stream of packets also contains *MN*'s certificate:

- the length of the signatures in the protocol header,
- the size of the keys in a node's certificate, and
- the size of the signature in a node's certificate

In order to estimate how big the influence of different signature schemes on the above mentioned sizes is, we used the OpenSSL [3] programs `rsagen`, `rsautl`, `req` and `x509` to generate RSA signatures and RSA signed certificates with different key sizes (see Appendix A.3 for detailed command line options). We also used a self-developed tool for generating ECDSA signatures and ECDSA signed certificates. Figures 4.4.1, 4.4.2, and 4.4.3 for the different behavior with respect to increasing key sizes. The values for the RSA scheme have been obtained with key sizes that provide a comparable level of security as the ECDSA key sizes. Keys, signatures and certificates are stored in files according to the ASN.1 distinguished encoding rules (DER). Note, that the sizes of the certificates also depend on other certificate fields such as the issuer or the subject. However, Figure 4.4.3 shall only give a notion about the relative behavior for increasing key sizes and not about absolute certificate sizes.

Figure 4.4.1 Size of the private key file depending on the signature scheme

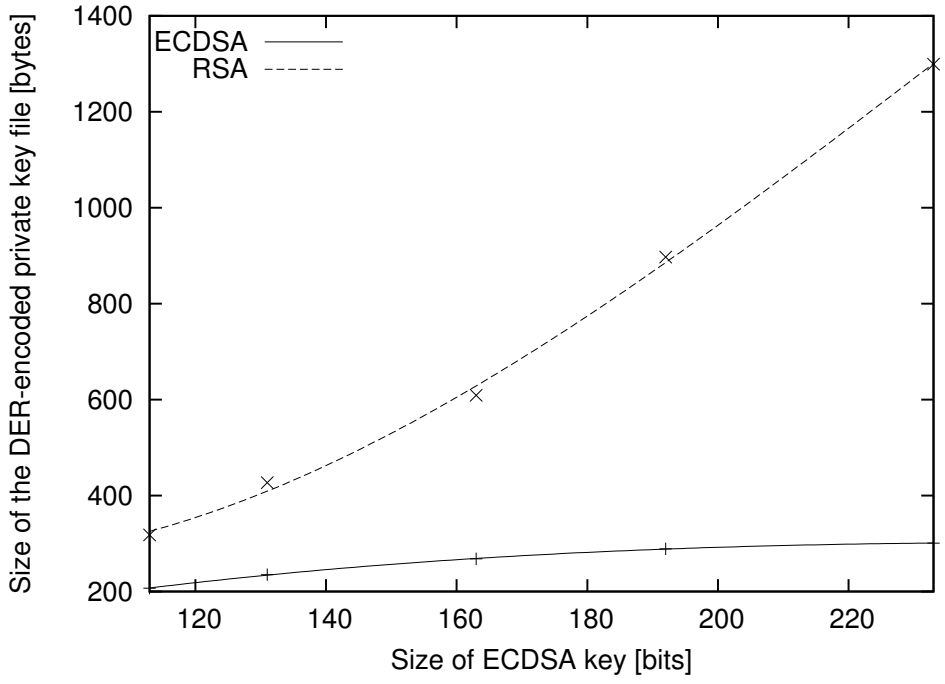


Figure 4.4.2 Size of the signature file depending on the signature scheme

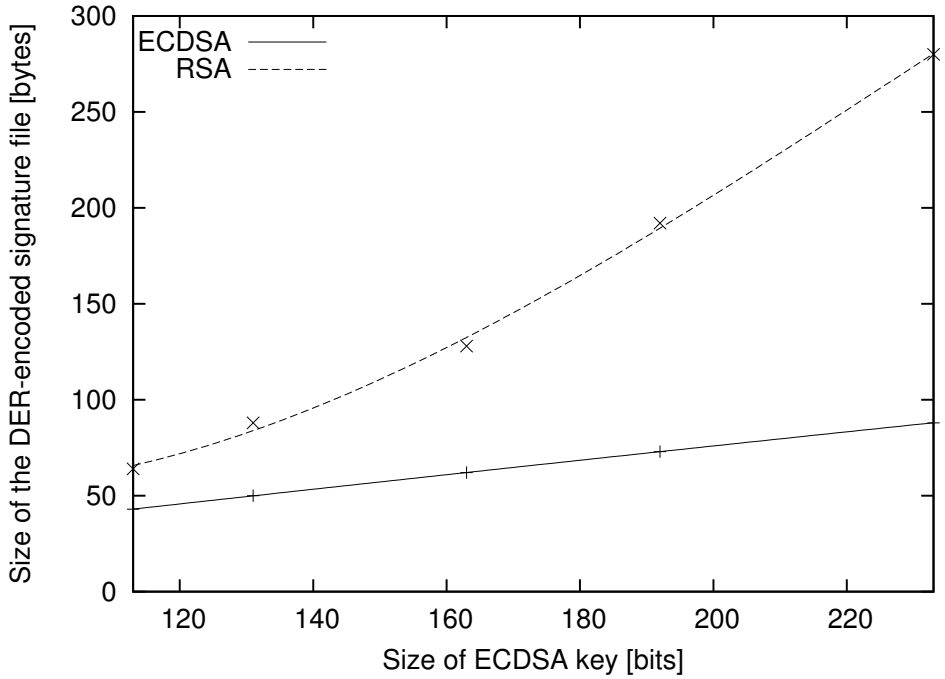
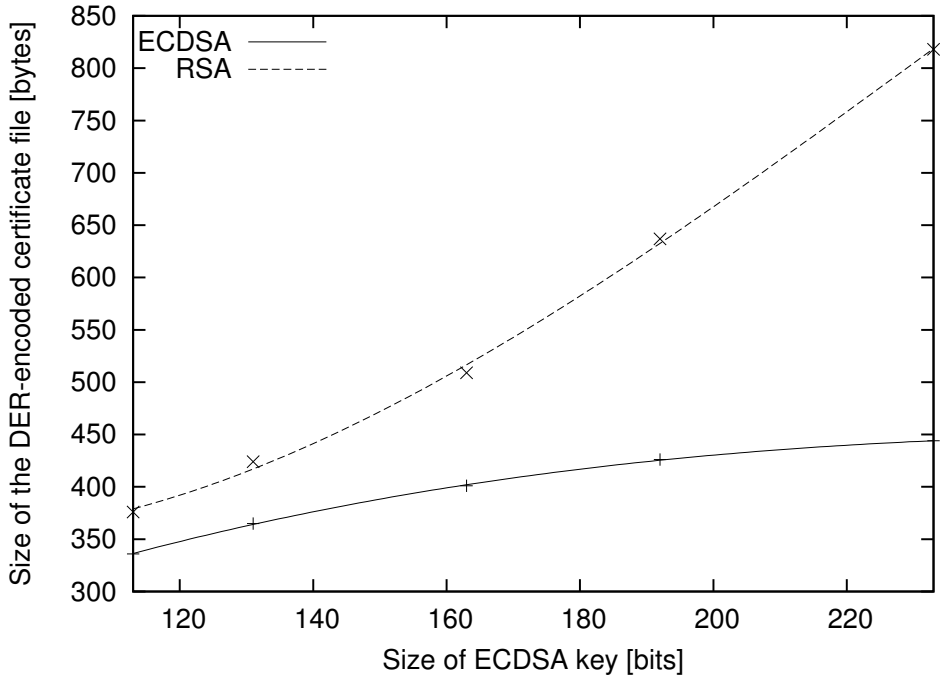


Figure 4.4.3 Size of the certificate file depending on the signature scheme



In all three figures, the size increases significantly with increasing key size for the RSA signature scheme whereas the ascent is relatively moderate for the ECDSA signature scheme. This fact is one of the major advantages of the ECDSA signature schemes. Since according to Moore's law computational power will grow exponentially in the next years, one can expect that the recommended key size for the same level of security will also increase in the future. Due to the moderate growth of the key, certificate and signature sizes the ECDSA signature scheme can be easily adopted to newly recommended key sizes. This means for example, that the space reserved for the signature in the protocol header may be chosen to be smaller. This obviously reduces the protocol overhead.
Hence, from the data overhead point of view and with respect to future developments ECDSA would clearly be the better choice for the secure charging protocol.

Optimal choice with respect to CPU time

Instead of using measures that reflect the overall network performance, for example the time a packet needs to travel from the source to the destination, we want to focus on an individual node and estimate the cost of the cryptographic operations. According to the specification of the secure charging protocol, a node within the network may play different roles in the communication process. It may, for example, be the mobile node in one session, the first intermediate node in another session or the corresponding node in again another session. For this reason, we suggest the following measure for evaluating the performance of the signature scheme in our application:

> *CPU time an individual node involved in the communication process spends on average for the signature operations necessary to process a single packet.*

Note, that this measure does not determine the time a packet needs to travel from the originator to the destination. Also, it does not reflect the total CPU time of all nodes involved in a particular communication chain. For an average node in the network the measure evaluates the CPU cost of the cryptographic operations. In mathematical terms, the measure for a node x can be stated as follows:

$$t_{\mathrm{CPU}}(x) = Pr(x = MN) \cdot t_{MN} + \sum_{i=1}^{n} Pr(x = N_i) \cdot t_{N_i} + Pr(x = CN) \cdot t_{CN} \qquad (4.1)$$

where $Pr(\cdot)$ denotes the probability of being the mobile node *MN*, an intermediate node N_i or the corresponding node *CN*. The quantities t_{MN}, t_{N_i}, and t_{CN} represent the CPU time spent in those roles for the signature operations.

Note, that this measure cannot be used to determine the end-to-end traffic delay caused by the protocol, because by some means such as multi-thread implementation the actual delay can be reduced significantly.

As mentioned before, the usual device in a wireless network is constrained with respect to battery power and CPU power. Obviously, reducing the number of CPU cycles spend

for a task also saves battery power. The advantage of this measure over a simple comparison of execution times is that it takes into account the execution times for signature generation *and* verification and *weights* them according to the probability of their occurrence for an average node. As we will see, estimations of the quantities in Equation (4.1) can be easily determined on the basis of the protocol definition.

Table 4.4.4 Times for signature operations with different signature schemes on a StrongARM CPU @ 206 MHz

Level of Security (key size)		Time for Signature Generation [ms]		Time for Signature Verification [ms]	
ECDSA	RSA	ECDSA	RSA	ECDSA	RSA
113 bit	512 bit	2.8	13.7	7.5	1.3
131 bit	704 bit	3.8	32.4	11.5	2.5
163 bit	1024 bit	5.7	78.0	17.9	4.3
193 bit	1536 bit	7.6	251.9	26.0	9.7
233 bit	2240 bit	10.1	731.8	37.3	20.4

For our evaluation, we used the RSA implementation of the OpenSSL project [3] (developer snapshot 20021202) as an example for the RSA signature scheme. We created the RSA keys and signatures with the OpenSSL crypto library functions. For the key generation, we used the same public exponent that the `genrsa` tool of the OpenSSL package uses by default, namely $2^{16} + 1 = 65537$. The source for the ECDSA execution times is our own implementation, which is also part of this work (see Chapter 6 for details). Table 4.4.4 contains the key sizes of comparable level of security, together with the execution times of signature generation and signature verification on our StrongARM 206MHz.

In the following, we will consider two design scenarios for the charging protocol and evaluate the performance of RSA and ECDSA by means of our proposed measure.

Scenario 1. In this scenario, the mobile node signs every packet it sends and each intermediate node on the route verifies this signature. In their original protocol proposal, the authors suggest not to sign and verify every packet. We will examine this in Scenario 2.

Let us begin our evaluation by summarizing for this scenario which signature operations are performed by which node.

For every packet:

- The mobile node *MN* calculates a signature on the route.
- The first intermediate node N_1 verifies *MN*'s signature on the route.

- The remaining intermediate nodes N_i (including the last intermediate node N_n) verify *MN*'s signature on the route.

After a bundle of b packets:

- The corresponding node *CN* signs the amount of data received.
- The last intermediate node N_n verifies *CN*'s signature on the received amount of data.

Considering that each communication within the network must involve all roles, it is straightforward to obtain the probabilities of the different roles. We feel that it makes sense to assume that it is equally likely for a network node to be the mobile node, some intermediate node or the corresponding node. Hence, if we suppose that on average there are n intermediate nodes between *MN* and *CN*, then the probability for a node x to play a particular role is

$$Pr(x = MN) = Pr(x = CN) = Pr(x = N_i) = 1/(n + 2).$$

Now that we have the probabilities, the remaining quantities we have to determine in order to evaluate our measure are the execution times of the signature operations performed in the different roles. Looking at the protocol definition, we find these times to be:

- *As mobile node MN:*
  $t_{MN} = t_s$ for signing the route of every packet.

- *As intermediate node N_i, but not N_n:*
  $t_{N_i} = t_v$ for verifying *MN*'s signature on the route of every packet.

- *As last intermediate node N_n:*
  $t_{N_n} = \frac{1}{b} \cdot t_v + t_v$ for verifying *CN*'s signature on the amount of data received and for verifying *MN*'s signature on the route of every packet.

- *As corresponding node CN:*
  $t_{CN} = \frac{1}{b} \cdot t_s$ for signing the amount of data received after a bundle of packets.

Thus, having determined all missing quantities of Equation (4.1), let us examine the measure for this concrete scenario:

$$t_{\mathrm{CPU}}(x) = \frac{t_s}{n+2} + \sum_{i=1}^{n-1} \frac{t_v}{n+2} + \frac{\frac{1}{b} \cdot t_v + t_v}{n+2} + \frac{\frac{1}{b} \cdot t_s}{n+2} \tag{4.2}$$

After reordering the equation, we get:

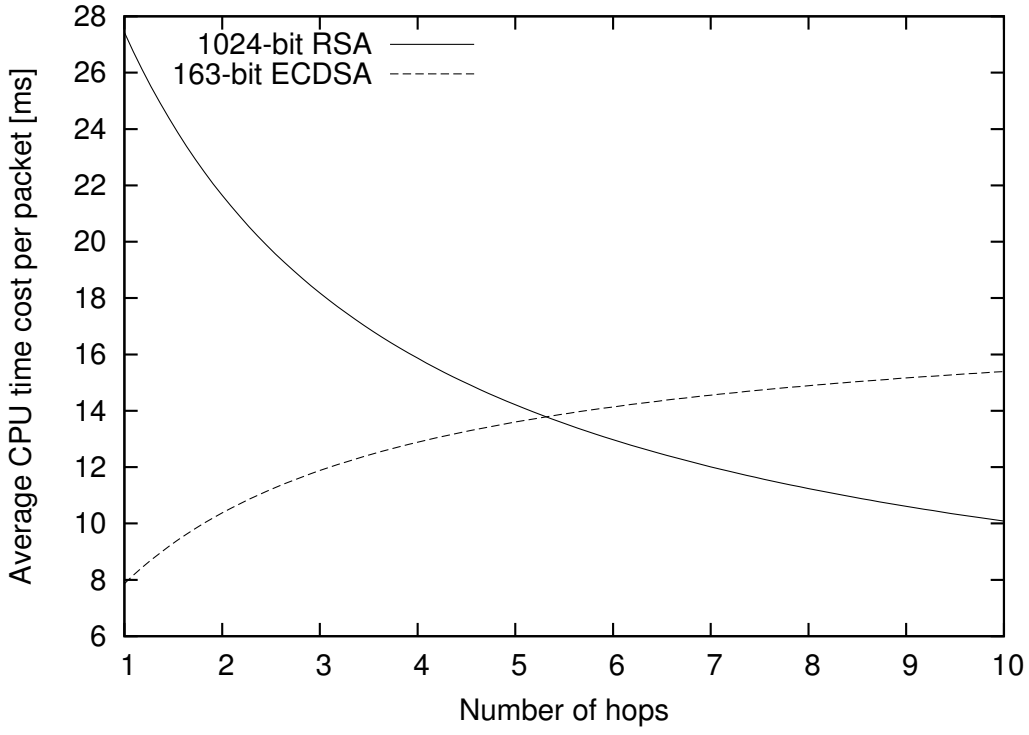
$$t_{\mathrm{CPU}}(x) = \frac{1}{n+2}\left(t_s\left[1+\frac{1}{b}\right] + t_v \cdot \left[n+\frac{1}{b}\right]\right) \tag{4.3}$$

The corresponding node confirms the amount of data received after each bundle of b packets. Lamparter, Paul, and Westhoff suggest b to be equal to 10. Hence, since $1/10$ is significantly smaller than 1 or n, let us simplify above equation by neglecting $1/b$:

$$t_{\mathrm{CPU}}(x) = \frac{1}{n+2}\left(t_s + t_v \cdot n\right) \tag{4.4}$$

As final step, we use the timings from Table 4.4.4 to evaluate the performance of our ECDSA and RSA implementations. Figure 4.4.4 demonstrates the influence of the average route length on the performance of 163-bit ECDSA and 1024-bit RSA. For networks in which the average number of hops between mobile node and corresponding node is relatively small, ECDSA would be the better choice as the average CPU time per packet is smaller. However, for networks with more than 5 hops, RSA yields better performance. Obviously, as the number of hops grows, the probability of being some intermediate node increases whereas the probability of being the mobile node *MN* decreases. Since the intermediate nodes predominantly perform signature verifications and the source nodes only perform signature generations, the signature verification time becomes more important for large n. Consequently, RSA performs better than ECDSA in this case.

Figure 4.4.4 Influence of the network topology on the performance of ECDSA and RSA.



Attentive readers will have noticed, that the performance gap between RSA and ECDSA changes with increasing level of security. In fact, for low levels of security, e.g. 512-bit RSA, the signature verification times of RSA are more than 5 times faster than the times of ECDSA. However, for high levels of security, e.g. 2240-bit RSA, the RSA scheme is not even twice as fast as the ECDSA scheme. For this reason, let us examine the performance of both schemes for different levels of security with the following inequality:

$$t_{\text{CPU}}^{\text{RSA}}(x) \geq t_{\text{CPU}}^{\text{ECDSA}}(x) \tag{4.5}$$

$$\frac{1}{n+2}\left(t_s^{\text{RSA}} \cdot + t_v^{\text{RSA}} \cdot n\right) \geq \frac{1}{n+2}\left(t_s^{\text{ECDSA}} + t_v^{\text{ECDSA}} \cdot n\right) \tag{4.6}$$

After some reordering we arrive at the following inequality:

$$n \leq \frac{t_s^{\text{RSA}} - t_s^{\text{ECDSA}}}{t_v^{\text{ECDSA}} - t_v^{\text{RSA}}} \tag{4.7}$$

Figure 4.4.5 Optimal choice depending on the average route length and the required level of security. The area below the curve covers situations in which ECDSA performs better than RSA.

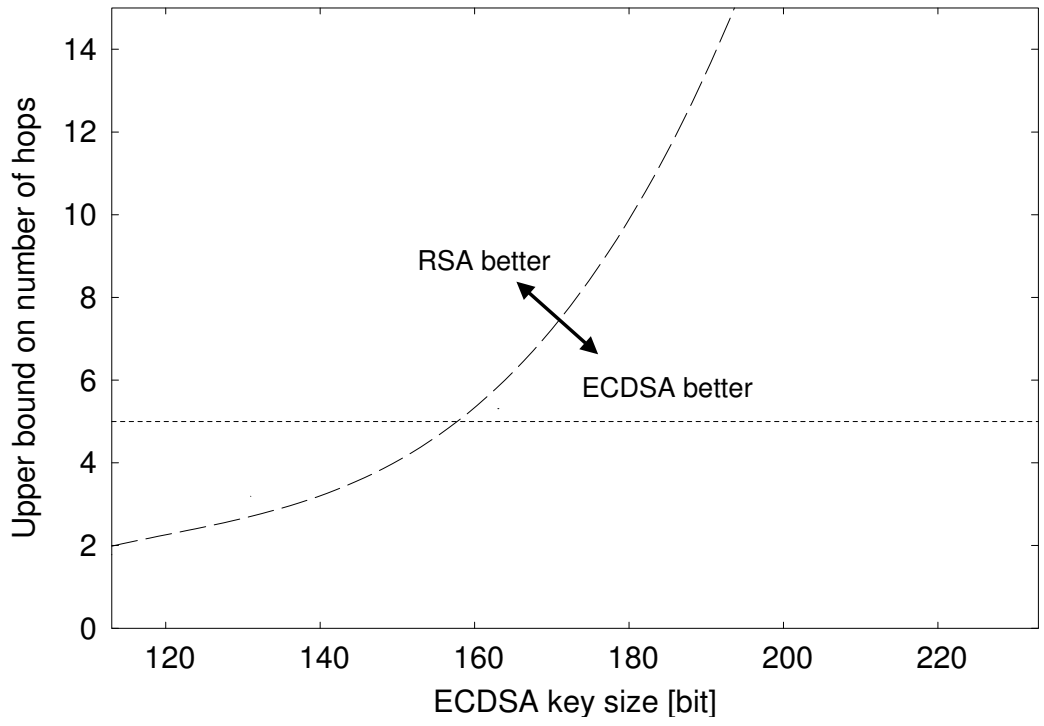


Figure 4.4.5 demonstrates the value of this bound on n for different levels of security. The area under the curve covers the situations when according to our measure ECDSA

performs better than RSA and the area above the curve covers the situations when RSA performs better. An analysis of co-operation approaches in multihop ad-hoc networks [31] shows that, under the assumption of UDP traffic, such approaches only improve the network performance for less than approximately $n = 5$ intermediate nodes. Hence, for this scenario, RSA may be the better choice for our recommended 131-bit ECDSA level of security and ECDSA will always be the better choice for levels of security greater than 163-bit ECDSA.

Scenario 2. In their protocol proposal, Lamparter, Paul and Westhoff come up with several optimizations that reduce the computational cost caused by the digital signatures. First of all, they demand only the first intermediate node to verify the signature on the data packets. The remaining nodes may choose not to verify the signatures. Moreover, the authors mention, that not every packet from mobile node to corresponding node has to be signed, but that signing and verifying only the first packet belonging to a stream of data packets would not deteriorate the security considerably. In this scenario, let us presume that still every packet is signed by the mobile node, but only the *first* intermediate node verifies the signature. It turns out that the other optimizations reduce the overall computational cost caused by signature operations, but do not influence the measure for choosing the best signature scheme. The reason for this is that signing only the first packet of a stream reduces the number of sign operations and as only the signed packets have to be verified, the number of verify operations decreases proportionally.

With these assumptions, the execution times for signature operations for the network nodes are the following:

- *As mobile node MN:*
  $t_{MN} = t_s$ for signing the route of every packet.

- *As first intermediate node N_1:*
  $t_{N_1} = t_v$ for verifying *MN*'s signature on the route of every packet.

- *As last intermediate node N_n:*
  $t_{N_n} = \frac{1}{b} \cdot t_v$ for verifying *CN*'s signature on the amount of data received.

- *As corresponding node CN:*
  $t_{CN} = \frac{1}{b} \cdot t_s$ for signing the amount of data received after a bundle of packets.

The influence of the new assumptions on our measure is the following:

$$t_{\mathrm{CPU}}(x) = \frac{t_s}{n+2} + \frac{t_v}{n+2} + \frac{\frac{1}{b} \cdot t_v}{n+2} + \frac{\frac{1}{b} \cdot t_s}{n+2} \tag{4.8}$$

With some reordering we get:

$$t_{\mathrm{CPU}}(x) = \frac{1}{n+2}\left(1+\frac{1}{b}\right)(t_s + t_v) \tag{4.9}$$

Obviously, the weighting of the execution times for signature generation and signature verification does no longer depend on the network structure, i.e. the average number of intermediate hops between source and destination. Consequently, ECDSA is clearly the best choice for this scenario, because the sum $t_s + t_v$ for ECDSA is smaller than for ECC independently of the desired level of security as presented in Table 4.4.4 (at least for security levels equal or greater than 512-bit RSA security).

Conclusions

From the computational point of view, we recommend to use ECDSA as signature scheme for the secure charging protocol. The reason for this recommendation is that ECDSA outperformed RSA in above analysis of Scenario 2. The final implementation of the secure charging protocol will certainly correspond to Scenario 2, since the developers of the protocol want to keep the amount of computationally relatively expensive signature operations as small as possible. Another argument for choosing ECDSA is the data overhead, which is significantly smaller for ECDSA than for RSA and also increasing at a much slower rater for higher levels of security.

5. Elliptic Curve Cryptography

Motivated by the results of our analysis in the last chapter, let us provide the information about elliptic curve cryptography that is necessary to develop a fast implementation of ECDSA for the prototype implementation of the secure charging protocol. Before presenting detailed algorithms and equations to perform arithmetic with points on elliptic curves, we give a short explanation of finite fields. The description of elliptic curve arithmetic focuses on elliptic curves over $\mathbb{F}_{2^m}$ and Koblitz curves. The last section of this chapter contains a summary of known attacks against elliptic curve cryptosystems. Moreover, we try to give the reader a notion about the security of such cryptosystems.

Note that more details about our implementation, in particular the algorithms used for finite field arithmetic, can be found in the next chapter.

5.1. Introduction to Finite Fields

A *finite field* consists of a finite set of elements F, two binary operations, addition and multiplication, and the additive and multiplicative inverses of each element. The binary operations satisfy certain arithmetic properties. The number of elements in the field is called the *order* of the finite field. There exists a finite field of order q if and only if q is a prime power. Essentially, there is only one finite field of order q denoted by $\mathbb{F}_q$. If $q = p^m$ where p is a prime and m is a positive integer, then p is called the *characteristic* of $\mathbb{F}_q$ and m is called the *extension degree* of $\mathbb{F}_q$.

In the following, we shortly describe the two most important types of finite fields applied in practice, the prime field $\mathbb{F}_p$ and the binary field $\mathbb{F}_{2^m}$.

5.1.1. The Finite Field $\mathbb{F}_p$

We call the finite field $\mathbb{F}_p$ where p is a prime number *prime field*. It is represented by the set of integers $\{0, 1, 2, \ldots, p-1\}$. The addition operation is *addition modulo* p, which means that for $a, b \in \mathbb{F}_p$, $a + b = r$, where r is the remainder of $a + b$ divided by p. The multiplication operation is *multiplication modulo* p, which means that for $a, b \in \mathbb{F}_p$, $a \cdot b = s$, where s is the remainder of $a \cdot b$ divided by p. If a is a non-zero element in $\mathbb{F}_p$, we say that the *inverse* of a modulo p, denoted by a^{-1}, is the unique integer $c \in \mathbb{F}_q$ for which $a \cdot c = 1$.

5.1.2. The Finite Field $\mathbb{F}_{2^m}$

The finite field $\mathbb{F}_{2^m}$ can be viewed as a vector space of dimension m over the field $\mathbb{F}_2$ which consists of the two elements 0 and 1. $\mathbb{F}_{2^m}$ is often referred to as *characteristic two finite field* or *binary finite field*. As it is a vector space, every element a of $\mathbb{F}_{2^m}$ can be represented as a bit string $(a_0 a_1 \dots a_{m-1})$:

$$a = a_0 \cdot \beta_0 + a_1 \cdot \beta_1 + \dots + a_{m-1} \cdot \beta_{m-1}, \text{where } a_i \in 0, 1.$$

The set $\{\beta_0, \beta_1, \dots, \beta_{m-1}\}$ is called a *basis* of $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$. There are many different bases and some of them lead to more efficient implementations than others. In this thesis, we only consider *polynomial basis representations*, because they are well suited to microprocessor architectures. Other bases are described, for example, in [24], which is also our main reference for this section. An irreducible polynomial of degree m over $\mathbb{F}_2$ can be written as $f(x) = x^m + f_{m-1}x^{m-1} + \dots + f_2x^2 + f_1x + f_0$, where $f_i \in \{0, 1\}$. Irreducible means that it cannot be factored as a product of two polynomials over $\mathbb{F}_2$, each of degree less than m. These so-called *reduction polynomials* $f(x)$ define a polynomial basis representation of $\mathbb{F}_{2^m}$, i.e.

$$\begin{aligned} \mathbb{F}_{2^m} &\simeq \{a_{m-1}x^{m-1} + \dots + a_1x + a_0 : a_i \in \{0, 1\}\} \\ &\simeq \{(a_{m-1} \dots a_1 a_0) : a_i \in \{0, 1\}\} \end{aligned}$$

Thus, the elements of $\mathbb{F}_{2^m}$ can be represented by the set of all binary strings of length m. The multiplicative identity element is represented by the bit string $(00 \dots 01)$ and the additive identity element is represented by the bit string of all 0's.
Addition is performed as bitwise XOR of the vector coefficients [6], i.e. if we have two elements of $\mathbb{F}_{2^m}$, $a = (a_{m-1} \dots a_1 a_0)$ and $b = (b_{m-1} \dots b_1 b_0)$, then $a + b = c = (c_{m-1} \dots c_1 c_0)$, where $c_i = a_i + b_i \mod 2$.
If $a = (a_{m-1} \dots a_1 a_0)$ and $b = (b_{m-1} \dots b_1 b_0)$ are elements of $\mathbb{F}_{2^m}$ multiplication is performed as follows: $a \cdot b = r = (r_{m-1} \dots r_1 r_0)$, where the polynomial $r(x) = r_{m-1}x^{m-1} + \dots + r_1x + r_0$ is the remainder when the polynomial

$$(a_{m-1}x^{m-1} + \dots + a_1x + a_0) \cdot (b_{m-1}x^{m-1} + \dots + b_1x + b_0)$$

is divided by the reduction polynomial $f(x)$.

5.2. Introduction to Elliptic Curves

In 1985, Miller [46] and Koblitz [27], independently proposed a public-key cryptosystem analogous to the ElGamal schemes [17] in which the multiplicative group of integers modulo p, denoted by $\mathbb{Z}_p^*$, is replaced by the group of points on an elliptic curve defined over a finite field. Since the best algorithm known for solving the underlying computationally hard mathematical problem, the *elliptic curve discrete logarithm problem* (ECDLP), takes

fully exponential time, whereas the best algorithms known for solving the underlying computationally hard mathematical problems in RSA (integer factorization problem) and DSA (the discrete logarithm problem) take sub-exponential time, significantly smaller parameters can be used in elliptic curve cryptography (ECC) than in other systems such as RSA and DSA. For example, a 163-bit ECC key has a comparable level of security (against known attacks) as RSA and DSA with a modulus of 1024 bits [36]. This means by using ECC one can reach the same level of security with less expense of processing power, storage space, bandwidth and electrical power, which makes it especially interesting for applications on constrained devices such as smartcards, mobile phones and handhelds.

The performance of ECC depends mainly on the efficiency of finite field computations and fast algorithms for elliptic scalar multiplications. Selecting particular underlying fields and/or elliptic curves also speeds up the implementation. In Section 5.1, we already gave examples of such finite fields. Examples of families of curves that offer computational advantages are Koblitz curves over $\mathbb{F}_{2^m}$.

Let us now introduce the mathematical definition of the elliptic curves we will work with in the following. Suppose we have a finite field $\mathbb{F}_{2^m}$. Then, the polynomial equation

$$E : y^2 + xy = x^3 + ax^2 + b, \tag{5.1}$$

with coefficients $a, b \in \mathbb{F}_{2^m}$ together with the point at infinity $\mathcal{O}$ define an *elliptic curve* over $\mathbb{F}_{2^m}$. Let us ask for solutions (x, y) with $x, y \in \mathbb{F}_{2^m}$. Such a solution is called *point on the elliptic curve* E.

5.3. Arithmetic on General Elliptic Curves over $\mathbb{F}_{2^m}$

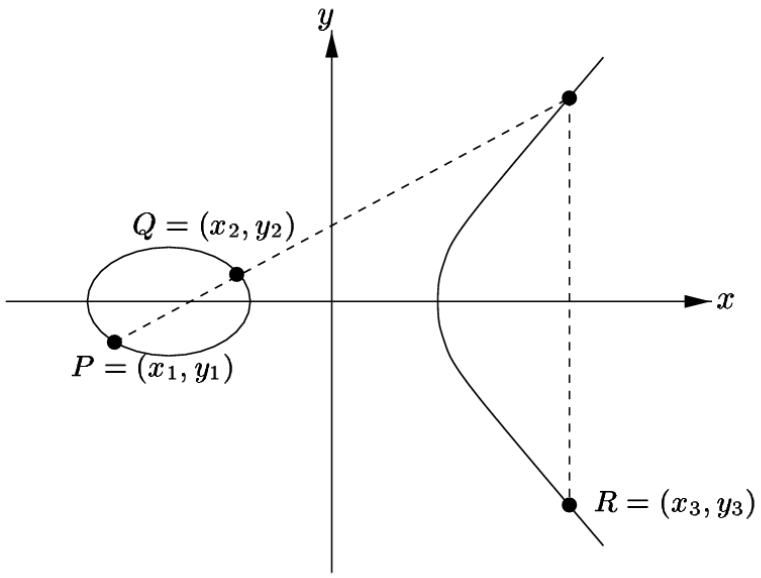
The elliptic curve digital signature algorithm (ECDSA, see Section 3.3 for details) is based on multiplying a point P on an elliptic curve with a scalar k. This is the nothing else than a k-fold addition of P. Hence, addition and scalar multiplication of points are important arithmetic operations, which for this reason will be described in the following.

5.3.1. Point Addition

There is an illustrative way to explain the law for adding two points on an elliptic curve. Let us assume that the coefficients of our elliptic curve E as well as x and y are rational numbers. Then, we can certainly draw the curve E which is the set of all solutions (x, y) of Equation (5.1).

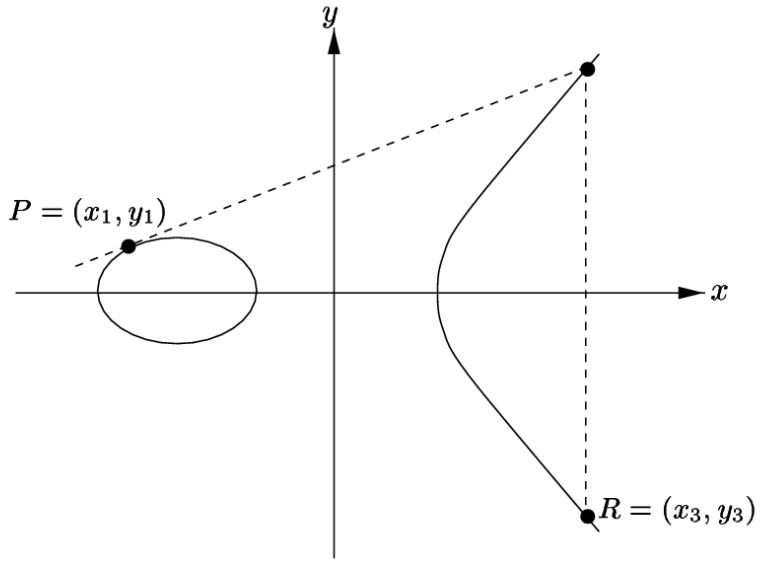
Starting with two distinct rational points P and Q on E, we draw a line through P and Q and obtain a third point of intersection of the line with the curve. Reflecting this point in the x-axis, we obtain a point R, which we define to be the result of adding $P + Q$ (see Figure 5.3.1). Since the line, the curve and the points of intersection are rational, R must also be rational.

Figure 5.3.1 Adding two points on an elliptic curve [24].



Even if we only have one rational point P, we can still generally get another one by drawing the tangent line to the curve at P. This line will intersect with another point on the curve. If we then reflect this point in the x-axis we obtain the point $R = 2P$, which will also be rational (see Figure 5.3.2).

Figure 5.3.2 Doubling a point on an elliptic curve [24].



One can show that together with a zero element $\mathcal{O}$, these operations form a group. Let us agree on the convention that the points on our elliptic curve consist of the ordinary

points in the ordinary affine x-y-plane together with a point $\mathcal{O}$ at infinity. We will call this point *point at infinity*.

Addition using affine coordinates

Let us assume that $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ with $\mathcal{O} \neq P_1 \neq -P_2 \neq \mathcal{O}$ are two distinct points on the curve E given by Equation (5.1). Based on the previously mentioned geometric motivation one can derive the following formulas [19] for computing the coordinates of the point $P_3 = P_1 + P_2 = (x_3, y_3)$. Note that we are in $\mathbb{F}_{2^m}$, i.e. $a, b, x_i, y_i \in \mathbb{F}_{2^m}$:

$$\begin{aligned} \lambda &= \begin{cases} \frac{y_1+y_2}{x_1+x_2} & \text{if } P_1 \neq P_2 \\ \frac{y_1}{x_1} + x_1 & \text{if } P_1 = P_2 \end{cases} \\ x_3 &= \lambda^2 + \lambda + x_1 + x_2 + a \\ y_3 &= (x_1 + x_3)\lambda + x_3 + y_1 \end{aligned} \tag{5.2}$$

The solutions for the two trivial cases $P_j = \mathcal{O}$ and $P_1 = -P_2$, which we have excluded above, are straightforward, namely $P_i + \mathcal{O} = P_i$ and $P_1 + (-P_2) = \mathcal{O}$, respectively.

In order to estimate and compare the computational cost of the following algorithms, which will use above equations to add two points in affine coordinates, let us examine the equations with respect to the required number of field operations. Obviously, the formulas require 1 field division and 1 field multiplication. The computational cost of field additions and squarings can be neglected, since they can be done much faster than inversion or multiplication (see Table 6.3.1 in Section 6.3.3 for concrete execution times on our target platform).

Addition using projective coordinates / mixed coordinates

As inversion in $\mathbb{F}_{2^m}$ is computationally expensive relative to multiplication, it turns out that representing a point using projective coordinates might be advantageous. There exist several types of projective coordinates. In standard projective coordinates one finds that the projective point $(X, Y, Z), Z \neq 0$ corresponds to the affine point $(x, y) = (X/Z, Y/Z)$. Then, the projective equation of the elliptic curve is $Y^2Z + XYZ = X^3 + aX^2Z + bZ^3$.

In [37] a different set of projective coordinates is introduced that leads to computationally fast point arithmetic. The underlying map is the following

$$\begin{aligned} \{(X, Y, Z) : Z \neq 0\} &\longmapsto \left\{(x, y) = \left(\frac{X}{Z}, \frac{Y}{Z^2}\right)\right\} \\ \{(X, Y, 0)\} &\longmapsto \mathcal{O} \end{aligned}$$

and

$$\begin{aligned} \{(x, y)\} &\longmapsto \{(X, Y, Z) = (x, y, 1)\} \\ \mathcal{O} &\longmapsto \{(\alpha_1, \alpha_2, 0), \text{ for any } \alpha_i\} \end{aligned}$$

The projective equation of the elliptic curve is

$$Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4. \tag{5.3}$$

The addition of two points using mixed coordinates, i.e. one point given in affine coordinates and the other in projective coordinates, is especially of use in point multiplication methods presented later in this thesis. Here are the computation steps for a projective point $P_1 = (X_1, Y_1, Z_1)$ and an affine point $P_2 = (x_2, y_2)$ in the non-trivial case with $\mathcal{O} \neq P_1 \neq -P_2 \neq \mathcal{O}$. The result is the projective point $P_3 = P_1 + P_2 = (X_3, Y_3, Z_3)$.

$$\begin{array}{lllllllll} A & = & y_2 \cdot Z_1^2 + Y_1 & B & = & x_2 \cdot Z_1 + X_1 & C & = & Z_1 \cdot B \\ D & = & B^2 \cdot (C + aZ_1^2) & Z_3 & = & C^2 & E & = & A \cdot C \\ X_3 & = & A^2 + D + E & F & = & X_3 + x_2 \cdot Z_3 & G & = & X_3 + y_2 \cdot Z_3 \\ Y_3 & = & E \cdot F + Z_3 \cdot G & & & & & & \end{array} \tag{5.4}$$

This operation requires 10 field multiplications, which on most platforms is faster than addition in affine coordinates (1 field multiplication plus 1 field division). On our Sharp Zaurus platform, the execution times are 121 µs vs. 158 µs (Table 6.4.1).

Doubling a projective point, i.e. calculating $P_3 = P_1 + P_1$, can be done using the following formulas:

$$\begin{array}{lll} Z_3 & = & X_1^2 \cdot Z_1^2 \\ X_3 & = & X_1^4 + b \cdot Z_1^4 \\ Y_3 & = & bZ_1^4 \cdot Z_3 + X_3 \cdot (aZ_3 + Y_1^2 + bZ_1^4) \end{array} \tag{5.5}$$

This operation requires 5 field multiplications.

5.3.2. Scalar Point Multiplication

The computation of kP, where k is an integer and P is an elliptic curve point, is the basic operation of cryptographic schemes based on elliptic curves. It also dominates the execution time of those schemes. Thus, using efficient algorithms for point multiplication has a strong influence on the performance of elliptic curve cryptosystems. [19] presents a good overview over fast multiplication algorithms. In the following, we will describe the algorithms we implemented.

Binary method

The simplest method for multiplying kP is based on the repeated-square-and-multiply method for exponentiation. Algorithm 5.3.1 shows how it works.

Algorithm 5.3.1 Left-to-right binary method for point multiplication [19].
INPUT: $k = (k_{m-1}, \ldots, k_1, k_0)_2$, $P \in \mathbb{F}_{2^m}$.
OUTPUT: kP.
1: $Q \longleftarrow \mathcal{O}$.
2: **for** $i = m - 1$ downto 0 **do**
3: $\quad Q \longleftarrow 2Q$.
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \longleftarrow Q + P$.
6: **return** Q.

P is usually given in affine coordinates. In order to minimize the number of computationally expensive field inversions, we store Q in projective coordinates. The doubling in Step 3 is done as in Equations (5.5). For the addition in Step 5, we use Equations (5.4) to add P in affine coordinates to Q in projective coordinates.

Assuming that the average number of ones in the binary representation of k is $m/2$ and neglecting the fact that the very first point doubling is simply a doubling of $\mathcal{O}$, the algorithm requires approximately m point doublings and $m/2$ point additions. Using the computational complexity in terms of finite field operations derived for the point addition methods, we can also express the complexity of this multiplication method in terms of field multiplications and field divisions. The expected number of multiplications is $(5m + 10 \cdot (m/2)) = 10m$. Another multiplication and division is necessary to convert the result back to affine coordinates.

NAF and width-w NAF methods

The binary method can be sped up by using a different representation for k [59], the so-called nonadjacent form (NAF). This is a signed binary expansion with the property that no two consecutive coefficients are nonzero. For example,

$$\text{NAF}(29) = (1, 0, 0, -1, 0, 1)$$

since $29 = 32 - 4 + 1$. Every positive integer has a unique NAF. The NAF has the fewest nonzero coefficients of any signed binary expansion and it is at most one coefficient longer than the binary representation [48].

The NAF can be generalized to the so-called *width-w NAF* that additionally increases execution speed. For any $w > 1$, each positive integer has a unique width-w NAF, denoted by $\text{NAF}_w(n)$:

$$n = \sum_{j=0}^{l-1} u_j 2^j$$

where each u_j is odd, $|u_j| < 2^{w-1}$ and among any w consecutive coefficients at most one is nonzero. Algorithm 5.3.2 shows how $\text{NAF}_w(n)$ can be computed.
In Step 4 of Algorithm 5.3.2, the expression "k mods 2^w" denotes the integer u satisfying $u \equiv k \pmod{2^w}$ and $-2^{w-1} \leq u < 2^{w-1}$. The actual implementation of Step 4 involves some binary-bit arithmetic. The idea behind it is that k mods 2^w is a signed integer representation of the rightmost w bits. Therefore, we just isolate the rightmost w bits, extend the sign to a 32-bit value and get u as the result. Note that the subtraction in Step 5 is not a simple XOR-operation, but a long integer subtraction with borrow.

Algorithm 5.3.2 Computing the width-w NAF of a positive integer [19].

INPUT: A positive integer k.
OUTPUT: $\text{NAF}_w(k)$.

1: $i \longleftarrow 0$.
2: **while** $k \geq 1$ **do**
3: **if** k is odd **then**
4: $k_i \longleftarrow k$ mods 2^w.
5: $k \longleftarrow k - k_i$.
6: **else**
7: $k_i \longleftarrow 0$.
8: $k \longleftarrow k \gg 1$.
9: $i \longleftarrow i + 1$.
10: **return** $(k_{i-1}, k_{i-2}, \ldots, k_1, k_0)$.

Algorithm 5.3.3 finally illustrates how point multiplication is done using the width-w NAF. For the point subtraction in Step 9, we use the fact that if $P = (x, y)$ then $-P = (x, x + y)$. After performing this operation the remaining addition is again done according to Equations (5.4).

Algorithm 5.3.3 Window NAF method for point multiplication [19].

INPUT: Window width w, $\text{NAF}_w(k) = \sum_{i=0}^{l-1} k_i \cdot 2^i$, $P \in E(\mathbb{F}_{2^m})$.
OUTPUT: kP.

1: Precompute $P_i = iP$, for $i \in \{1, 3, 5, \ldots, 2^{w-1} - 1\}$.
2: $Q \longleftarrow \mathcal{O}$.
3: **for** $i = l - 1$ downto 0 **do**
4: $Q \longleftarrow 2Q$.
5: **if** $k_i \neq 0$ **then**
6: **if** $k_i > 0$ **then**
7: $Q \longleftarrow Q + P_{k_i}$
8: **else**
9: $Q \longleftarrow Q - P_{k_i}$
10: **return** Q.

Using affine coordinates the precomputation requires $2^{w-2}-1$ point additions and 1 point doubling. This equals 2^{w-2} field divisions and field multiplications. The average number of non-zero coefficients of the width-w NAF is approximately $1/(w+1)$. Hence, the main computation takes m point doublings and $m/(w+1)$ point additions in mixed coordinates. This corresponds to $5m + 10m/(w+1)$ field multiplications.

In case the same elliptic curve point P is multiplied with different integers k, the product kP can be computed faster by performing the precomputation in Step 1 only once for the point P and reusing the precomputed values for subsequent multiplications. The resulting method is called *fixed-base window NAF method*.

Montgomery method

The Montgomery Method presented in [38] is based on the following idea by Montgomery [47]: Suppose we have two points $Q_1 = (x_1, y_1)$ and $Q_2 = (x_2, y_2)$ in affine coordinates with $Q_1 \neq \pm Q_2$. Let $Q_1 + Q_2 = (x_3, y_3)$ and $Q_1 - Q_2 = (x_4, y_4)$. Using the addition formulas 5.2 we get

$$x_3 = x_4 + \frac{x_1}{x_1 + x_2} + \left(\frac{x_1}{x_1 + x_2}\right)^2. \tag{5.6}$$

The x-coordinate of $Q_1 + Q_2$ can be computed from the x-coordinates of Q_1, Q_2 and $Q_1 - Q_2$. Iteration j of Algorithm 5.3.4 computes $T_j = (lP, (l+1)P)$, where l is the integer given by the j leftmost bits of k. Then $T_{j+1} = (2lP, (2l+1)P)$ or $((2l+1)P, (2l+2)P)$ if the $(j+1)$st leftmost bit of k is 0 or 1, respectively. After the last iteration, having computed the x-coordinates of $kP = (x_1, y_1)$ and $(k+1)P = (x_2, y_2)$, the y-coordinate can be recovered as:

$$y_1 = x^{-1}(x_1 + x)[(x_1 + x)(x_2 + x) + x^2 + y] + y. \tag{5.7}$$

We derived this equation using the addition formula 5.2 for computing the x-coordinate x_2 of $(k+1)P$ from $kP = (x_1, y_1)$ and $P = (x, y)$. See Algorithm 5.3.4 for details.

Algorithm 5.3.4 Montgomery method for point multiplication [19].

INPUT: $k = (k_{s-1}, \ldots, k_1, k_0)_2$ with $k_{s-1} = 1$, $P \in E(\mathbb{F}_{2^m})$.
OUTPUT: kP.

1: $X_1 \longleftarrow x$, $Z_1 \longleftarrow 1$, $X_2 \longleftarrow x^4 + b$, $Z_2 \longleftarrow x^2$.{Compute $(P, 2P)$.}
2: **for** $i = s-2$ downto 0 **do**
3: **if** $k_i = 1$ **then**
4: $T \longleftarrow Z_1$, $Z_1 \longleftarrow (X_1 Z_2 + X_2 Z_1)^2$, $X_1 \longleftarrow xZ_1 + X_1 X_2 T Z_2$.
5: $T \longleftarrow X_2$, $X_2 \longleftarrow X_2^4 + bZ_2^4$, $Z_2 \longleftarrow T^2 Z_2^2$.
6: **else**
7: $T \longleftarrow Z_2$, $Z_2 \longleftarrow (X_1 Z_2 + X_2 Z_1)^2$, $X_2 \longleftarrow xZ_2 + X_1 X_2 T Z_1$.
8: $T \longleftarrow X_1$, $X_1 \longleftarrow X_1^4 + bZ_1^4$, $Z_1 \longleftarrow T^2 Z_1^2$.
9: $x_3 \longleftarrow X_1/Z_1$.

10: $y_3 \longleftarrow (x + X_1/Z_1)[(X_1 + xZ_1)(X_2 + xZ_2) + (x^2 + y)Z_1Z_2](xZ_1Z_2)^{-1} + y$.
11: **return** (x_3, y_3).

The loop in Algorithm 5.3.4 takes $6m$ field multiplications and Steps 9 and 10 sum up to 7 field multiplications and 2 field divisions.

Comb method

In the comb method, proposed in [34], the basic idea is to write the binary representation of the scalar k in w rows and to process the columns of the resulting rectangle one column at a time.

Algorithm 5.3.5 Comb method for point multiplication [19]

INPUT: Window width w, $d = \lceil s/w \rceil$, $k = (k_{s-1}, \ldots, k_1, k_0)_2$, $P \in E(\mathbb{F}_{2^m})$.
OUTPUT: kP.

1: Precompute $P_n = \sum_{i=0}^{w-1} n_i 2^{i \cdot d} P$ for $0 \leq n < 2^w$, where n_j is bit j in the binary representation of $n = (n_{w-1}, \ldots, n_0)_2$ (see Algorithm 6.4.1).
2: By padding k on the left with 0's if necessary, write $k = K^{w-1} \| \ldots \| K^1 \| K^0$, where each K^j is bit string of length d. Let K_i^j denote the ith bit of K^j.
3: $Q \longleftarrow \mathcal{O}$.
4: **for** $i = d - 1$ downto 0 **do**
5: $\quad Q \longleftarrow 2Q$.
6: $\quad Q \longleftarrow Q + P_{(K_i^{w-1}, \ldots, K_i^1, K_i^0)_2}$.
7: **return** Q.

Similar to the window NAF method, there is also a variant of this method for fixed points P, which performs the precomputation in Step 1 only once. This variant is called *fixed-base comb method*. Neglecting the computational steps for the precomputation, the calculation of kP requires d point doublings in projective coordinates and d point additions in mixed coordinates. The sum of field multiplications is $(5+10)d = 15d \approx \frac{15m}{32w}$. However, this requires memory for the storage of $2^w - 1$ precomputed elliptic curve points, which might be a serious disadvantage on some constrained devices.

Table 5.3.1 summarizes the complexity of the above mentioned multiplication methods for points on general elliptic curves over $\mathbb{F}_{2^m}$. Obviously, the Montgomery method yields the best performance among all methods for arbitrary points, whereas the fixed-base comb method is the best choice among all fixed-base methods.

Table 5.3.1 Estimated computational complexity of different multiplication methods in terms of finite field operations.

Multiplication Method	Number of Field Multiplications	Number of Field Divisions
Arbitrary Point Methods		
Binary Method	$10m$	-
Montgomery Method	$6m + 7$	2
Window NAF Method	$2^{w-2} + 5m + 10m/(w + 1)$	2^{w-2}
Fixed Point Methods		
Fixed-base Window NAF Method	$5m + 10m/(w + 1)$	-
Fixed-base Comb Method	$(15/32)m/w$	-

5.4. Arithmetic on Koblitz Elliptic Curves

In 1991, Koblitz presented a paper [28] in which he examines a subclass of binary elliptic curves with properties that allow for the use of efficient algorithms for scalar point multiplication. This type of curves is called *Koblitz curve* and they are defined by the equation

$$E_a : y^2 + xy = x^3 + ax^2 + 1, \tag{5.8}$$

where $a \in \mathbb{F}_2$, i.e. $a \in \{0, 1\}$. Obviously, the difference between general binary elliptic curves and Koblitz curves is that the choice for the coefficients in the defining equation is limited to the set $\{0, 1\}$. Koblitz discovered interesting properties of these curves that can be used to compute the point multiplication more efficiently. In this section, we introduce some of the theory that was summarized by Solinas in [59].

5.4.1. Basic Properties

The coefficients of Koblitz curves are either 0 or 1. Koblitz curves have the property that *if $P = (x, y)$ is a point on E_a then so is the point (x^2, y^2)* [59]. Hence, one finds

$$(x^4, y^4) + 2(x, y) = \mu \cdot (x^2, y^2) \tag{5.9}$$

for every point (x, y) on the curve E_a, where $\mu := (-1)^{1-a}$.

Using the Frobenius map $\tau(x, y) := (x^2, y^2), \tau(\mathcal{O}) := \mathcal{O}$ this can be written symbolically

$$(\tau^2 + 2)P = \mu\tau P, \tag{5.10}$$

where P is some point on the Koblitz curve.

Note that the Frobenius map is a very cheap operation from the computational point of view. It only requires squaring the point coordinates and this can be done in only a fraction of the time a field multiplication or division takes (see Section 6.3.1 for details).

The Frobenius map can be regarded as the complex number τ satisfying $\tau^2 + 2 = \mu\tau$, namely $\tau = \frac{\mu+\sqrt{-7}}{2}$. We can now represent the scalar k in our point multiplication operation as an element of the ring $\mathbb{Z}[\tau]$: $k = \sum_{i=0}^{l-1} u_i\tau^i$. Consequently, we can multiply points on the Koblitz curve by such an element

$$kP = (u_{l-1}\tau^{l-1} + \cdots + u_1\tau + u_0)P = u_{l-1}\tau^{l-1}(P) + \cdots + u_1\tau(P) + u_0P \tag{5.11}$$

Efficient point multiplication methods can now be developed by finding a nice representation $k = \sum_{i=0}^{l-1} u_i\tau^i$ where l is relatively small and the coefficients u_i are small and sparse.

Before describing the multiplication algorithms, let us introduce the so-called *Lucas sequences* as we will need them later in this section. They are defined as follows:

$$\begin{array}{llll} U_0 = 0, & U_1 = 1 & \text{and} & U_{k+1} = \mu U_k - 2U_{k-1} \quad \text{for } k \geq 1; \\ V_0 = 2, & V_1 = \mu & \text{and} & V_{k+1} = \mu V_k - 2V_{k-1} \quad \text{for } k \geq 1. \end{array} \tag{5.12}$$

Tables 5.4.1 summarizes some important definitions and relations for Koblitz curves, which we will need in the following.

Table 5.4.1 Important definitions and relations for Koblitz curves.

Definition		
μ	:=	$(-1)^{1-a}$
$\tau(x, y)$	:=	(x^2, y^2)

Relation
$f = \begin{cases} 2 & \text{for } a = 1 \\ 4 & \text{for } a = 0 \end{cases}$
$\tau^2 + 2 = \mu\tau$
$U_0 = 0, \quad U_1 = 1 \quad \text{and} \quad U_{k+1} = \mu U_k - 2U_{k-1} \quad \text{for } k \geq 1$
$V_0 = 2, \quad V_1 = \mu \quad \text{and} \quad V_{k+1} = \mu V_k - 2V_{k-1} \quad \text{for } k \geq 1$

5.4.2. Point Multiplication

τ-addic NAF (TNAF) multiplication method

Using the fact that $\tau^2+2 = \mu\tau$, one can show that every element in $\mathbb{Z}[\tau]$ can be expressed in canonical form $r_0+r_1\tau$. Algorithm 5.4.1 takes an element κ in this representation as input

and calculates its so-called *τ-addic NAF* (TNAF) $\kappa = \sum_{i=0}^{l-1} u_i \tau^i$. In this representation the coefficients u_i are either 0 or ± 1, and no two consecutive coefficients u_i are nonzero. Note, that this algorithm involves multi-precision integer arithmetic.

Algorithm 5.4.1 Computing the τ-addic NAF of an element in $\mathbb{Z}[\tau]$[19]

INPUT: $\kappa = r_0 + r_1\tau \in \mathbb{Z}[\tau]$.
OUTPUT: TNAF(κ).

1: $i \longleftarrow 0$.
2: **while** $r_0 \neq 0$ OR $r_1 \neq 0$ **do**
3: **if** r_0 is odd **then**
4: $u_i \longleftarrow 2 - (r_0 - 2r_1 \mod 4)$.
5: $r_0 \longleftarrow r_0 - u_i$.
6: **else**
7: $u_i \longleftarrow 0$.
8: $t \longleftarrow r_0/2$.
9: $r_0 \longleftarrow r_1 + \mu t$.
10: $r_1 \longleftarrow -t$.
11: $i \longleftarrow i + 1$.
12: **return** $(u_{i-1}, u_{i-2}, \ldots, u_1, u_0)$.

Each $\kappa \in \mathbb{Z}[\tau]$ has a unique τ-addic NAF. For example for $a = 1$, we find

$$\text{TNAF}(9) = (1, 0, -1, 0, 0, 1),$$

since we can write $9 = \tau^5 - \tau^3 + 1$. However, it turns out that the length of such a τ-addic NAF is approximately $l(k) \approx 2\log_2 k$ and therefore twice as long as the length of the normal NAF (introduced in Section 5.3.2).

The solution to this problem is an additional modular reduction step $\rho = \kappa \mod ((\tau^m - 1)/(\tau - 1))$, which is similar to the modular reduction of integers. In this formula, the variable m is the m in $\mathbb{F}_{2^m}$. Algorithm 5.4.2 shows how this modular reduction step can be done in an efficient way. Note, that we do not exactly compute the modular reduction, but an approximation, the so-called *partial modular reduction* $\rho' = \kappa$ partmod $((\tau^m - 1)/(\tau - 1))$.

Algorithm 5.4.2 Computation of the partial modular reduction of an element in $\mathbb{Z}[\tau]$ [19]

INPUT: An integer $k, 0 < k < n$, $C \geq 2$, the Lucas sequence V_m, and s_0, s_1, where $s_i = \frac{(-1)^i}{f}(1 - \mu U_{m+3-a-i})$.
OUTPUT: $\rho' = k$ partmod $((\tau^m - 1)/(\tau - 1))$.

1: $k' \longleftarrow \lfloor k/2^{a-C+(m-9)/2} \rfloor$.
2: **for** $i = 0$ to 1 **do**

3: $g' \longleftarrow s_i \cdot k'$.
4: $j' \longleftarrow V_m \cdot \lfloor g'/2^m \rfloor$.
5: $\lambda_i \longleftarrow \text{round}\left((g'+j')/2^{(m+5)/2}\right)/2^C$.
6: $f_i \longleftarrow \text{round}\,(\lambda_i)$.
7: $\eta_i \longleftarrow \lambda_i - f_i$.
8: $h_i \longleftarrow 0$.
9: $\eta \longleftarrow 2\eta_0 + \mu\eta_1$.
10: **if** $\eta \geq 1$ **then**
11: **if** $\eta_0 - 3\mu\eta_1 < -1$ **then**
12: $h_1 \longleftarrow \mu$.
13: **else**
14: $h_0 \longleftarrow 1$.
15: **else**
16: **if** $\eta_0 + 4\mu\eta_1 \geq 2$ **then**
17: $h_1 \longleftarrow \mu$.
18: **if** $\eta < -1$ **then**
19: **if** $\eta_0 - 3\mu\eta_1 \geq 1$ **then**
20: $h_1 \longleftarrow -\mu$.
21: **else**
22: $h_0 \longleftarrow -1$.
23: **else**
24: **if** $\eta_0 + 4\mu\eta_1 < -2$ **then**
25: $h_1 \longleftarrow -\mu$.
26: $q_0 \longleftarrow f_0 + h_0$.
27: $q_1 \longleftarrow f_1 + h_1$.
28: $r_0 \longleftarrow k - (s_0 + \mu s_1)q_0 - 2s_1q_1$.
29: $r_1 \longleftarrow s_1q_0 - s_0q_1$.
30: **return** $(r_0 + r_1\tau)$.

The function round() in this algorithm is a mapping of its input argument to the integer that is the nearest neighbor of that argument. Our implementation is based on the mathematical definitions

$$\text{round}(x) = \begin{cases} \text{trunc}(x+1/2) & \text{if } x > 0 \\ \text{trunc}(x-1/2) & \text{if } x < 0 \end{cases} \tag{5.13}$$

and

$$\lfloor x \rfloor = \begin{cases} \text{trunc}(x) & \text{if } x > 0 \\ \text{trunc}(x) - 1 & \text{if } x < 0 \end{cases} . \tag{5.14}$$

where trunc(x) returns the integer part of x.
In order to perform the arithmetic in Steps 5 through 7 without implementing long floating point numbers, we do the calculations with $2^{C+1}\lambda_i$ and $2^{C+1}f_i$. Since the absolute value of η is less than one, we switch to double precision floating-point variables (`double`) in Step 7. To our opinion, using floating-point arithmetic does not deteriorate the overall

performance in a noticeable way.
The parameter C ensures that TNAF(ρ') of the partially reduced element is not much longer than TNAF(ρ) of the reduced element. One can show that the length $l(\rho) \leq m+a$ and for $C \geq 2$ that $l(\rho') \leq m+a+3$. The probability that $\rho \neq \rho'$ is less than $\left(\frac{1}{2}\right)^{C-5}$. Furthermore, we know that the average density of nonzero coefficients of such a TNAF is approximately 1/3. Following the suggestion in [59], we chose $C = 12$ to obtain a good performance for our TNAF multiplication methods.

The above results lead to the following algorithm for scalar point multiplication on Koblitz curves, which has an expected running time of approximately $m/3$ point additions in mixed coordinates. This corresponds to $10 \cdot m/3 = (10/3)m$ field multiplications.
Note that Step 3 corresponds to simply squaring all three coordinates of Q.

Algorithm 5.4.3 TNAF method for point multiplication [19]
INPUT: TNAF(ρ') $= \sum_{i=0}^{l-1} u_i \tau^i$, where $\rho' = k$ partmod $((\tau^m - 1)/(\tau - 1))$, $P \in E_a(\mathbb{F}_{2^m})$.
OUTPUT: kP.
1: $Q \longleftarrow \mathcal{O}$.
2: **for** $i = l-1$ downto 0 **do**
3: $\quad Q \longleftarrow \tau Q$.
4: $\quad$ **if** $u_i = 1$ **then**
5: $\quad\quad Q = Q + P$.
6: $\quad$ **if** $u_i = -1$ **then**
7: $\quad\quad Q = Q - P$.
8: **return** Q.

Window TNAF multiplication method

Analogous to the width-w NAF multiplication method, we can extend the TNAF multiplication method to a window method. Using the Lucas sequences U_w, let us define

$$t_w := 2U_{w-1}U_w^{-1} \mod 2^w. \tag{5.15}$$

One can show that the odd numbers $\pm 1, \pm 3, \ldots, \pm(2^{w-1}-1)$ are incongruent modulo τ^w. For $\alpha_i = i \mod \tau^w$ for $i \in \{1, 3, \ldots, 2^{w-1}-1\}$, the *width-w TNAF* of $\kappa \in \mathbb{Z}[\tau]$ is the expression $\kappa = \sum_{i=0}^{l-1} u_i \tau^i$, where $u_i \in \{0, \pm\alpha_1, \pm\alpha_2, \ldots, \pm\alpha_{2^{w-1}-1}\}$.

One key property of the width-w TNAF is that at most one of any w consecutive coefficients is nonzero. Algorithm 5.4.4 is a simple method for computing TNAF$_w(\kappa)$. Note, that Step 4 is the same as in the computation of the width-w NAF. Hence, the implementation of this step does not differ a lot from the corresponding step in the window NAF algorithm.

Algorithm 5.4.4 Computing the width-w TNAF of an element in $\mathbb{Z}[\tau]$[19]

INPUT: w, t_w, $\alpha_i = \beta_i + \gamma_i\tau$ for $i \in \{1, 3, \ldots, 2^{w-1} - 1\}$, $\rho = r_0 + r_1\tau \in \mathbb{Z}[\tau]$.
OUTPUT: $\text{TNAF}_w(\rho)$.

1: $i \longleftarrow 0$.
2: **while** $r_0 \neq 0$ OR $r_1 \neq 0$ **do**
3: **if** r_0 is odd **then**
4: $u \longleftarrow r_0 + r_1 t_w \bmod 2^w$.
5: **if** $u > 0$ **then**
6: $s \longleftarrow 1$.
7: **else**
8: $s \longleftarrow -1$.
9: $u \longleftarrow -u$.
10: $r_0 \longleftarrow r_0 - s\beta_u$.
11: $r_1 \longleftarrow r_1 - s\gamma_u$.
12: $u_i \longleftarrow s\alpha_u$.
13: **else**
14: $u_i \longleftarrow 0$.
15: $t \longleftarrow r_0/2$.
16: $r_0 \longleftarrow r_1 + \mu t$.
17: $r_1 \longleftarrow -t$.
18: $i \longleftarrow i + 1$.
19: **return** $(u_{i-1}, u_{i-2}, \ldots, u_1, u_0)$.

Among all width-w TNAFs of length l the average density of non-zero coefficients is approximately $1/(w + 1)$. Using the partial modular reduction algorithm, we obtain a width-w TNAF of length approximately $l(\rho')$. The following algorithm uses $\text{TNAF}_w(\rho')$ to compute kP:

Algorithm 5.4.5 Window TNAF method for point multiplication [19]

INPUT: $\text{TNAF}_w(\rho') = \sum_{i=0}^{l-1} u_i\tau^i$, where $\rho' = k \bmod ((\tau^m - 1)/(\tau - 1))$, $P \in E_a(\mathbb{F}_{2^m})$.
OUTPUT: kP.

1: Precompute $P_u = \alpha_u P$, for $u \in \{1, 3, \ldots, 2^{w-1} - 1\}$.
2: $Q \longleftarrow \mathcal{O}$.
3: **for** $i = l - 1$ downto 0 **do**
4: $Q \longleftarrow \tau Q$.
5: **if** $u_i \neq 0$ **then**
6: Let u be such that $\alpha_u = u_i$ or $\alpha_{-u} = -u_i$.
7: **if** $u > 0$ **then**
8: $Q = Q + P_u$.
9: **else**
10: $Q = Q - P_{-u}$.

11: **return** Q.

The precomputation requires $2^{w-2} - 1$ point additions in affine coordinates, which corresponds to $2^{w-2} - 1$ field multiplications and divisions. The main multiplication routine has an expected running time of $m/(w+1)$ point additions in mixed coordinates, which corresponds to $10 \cdot m/(w+1)$ field multiplications. Of course, for a fixed point P, the precomputation needs only to be done once. In this case, we call the method *fixed-base window TNAF method*.

Table 5.4.2 summarizes the complexity of the above mentioned multiplication methods for points on Koblitz curves. Comparing these results to the complexities of methods for general elliptic curves over $\mathbb{F}_{2^m}$ (Table 5.3.1), one notices that the TNAF method is faster than the Montgomery Method. However, the window TNAF method is not faster than the fixed-base comb method for fixed points.

Table 5.4.2 Estimated computational complexity of different multiplication methods in terms of finite field operations.

Multiplication Method	Number of Field Multiplications	Number of Field Divisions
Arbitary Point Methods		
TNAF method	$(10/3)m$	-
Window TNAF Method	$10m/(w+1) + 2^{w-2} - 1$	$2^{w-2} - 1$
Fixed Point Methods		
Fixed-base Window TNAF Method	$10m/(w+1)$	-

5.5. Known Attacks Against Elliptic Curve Cryptosystems

In this section, we present some known attacks against elliptic curve cryptosystems. The scope of this section is limited to algorithms solving the elliptic curve discrete logarithm problem (ECDLP), i. e. to determine l given a point P and a point $Q = lP$, and it does not consider attacks against particular elements of digital signature algorithms based on elliptic curves. The following enumeration and further references can be originally found in [24].

1. **Naive Exhaustive Search:** The most simple approach to obtain l is to compute successive multiples of P: $P, 2P, 3P, 4P, \ldots$ until the result is equal to Q. In the worst case, this takes n steps, where n is the order of the point P.

2. **Baby-Step Giant-Step Algorithm:** This algorithm is a time-memory trade-off of the method of exhaustive search. It requires storage for about $\sqrt{n}$ points, and its running time is roughly $\sqrt{n}$ steps in the worst case.

3. **Pollard's Rho Algorithm:** This algorithm is a randomized version of the baby-step giant-step algorithm. With some modifications it can be sped up to have an expected running time of $\sqrt{\pi n}/2$ steps and it requires only a negligible amount of storage.

4. **Parallelized Pollard's Rho Algorithm:** The original Pollard's rho algorithm can be parallelized so that when it is run in parallel on r processors, the expected running time is roughly $\sqrt{\pi n}/(2r)$.

5. **Pollard's Lambda Method:** Pollard also presented a lambda method for computing discrete logarithms which is applicable when l, the logarithm sought, is known to lie in a certain interval. In particular, when l is known to lie in a subinterval $[0, b]$ of $[0, n-1]$, where $b < 0.39n$, the parallelized version of Pollard's lambda method is faster than the parallelized Pollard's rho algorithm.

6. **Multiple Logarithms:** It turns out that if a single instance of the ECDLP is solved using (parallelized) Pollard's rho method, the following instances (for the same curve E and the same base point P) can be solved faster, since some of the necessary work has already been done in the previous steps. In fact, solving k instances of the ECDLP takes only $\sqrt{k}$ as much work as it does to solve one instance.

Hence, the best known attack against a single instance of the ECDLP is Pollard's rho algorithm and has an expected running time of $\sqrt{\pi n}/2 = O(n^{1/2})$, which is fully exponential.

However, there are certain elliptic curves with special vulnerabilities that can be exploited by the following algorithms. These algorithms may have shorter running times as those mentioned before, therefore elliptic curves with these vulnerabilities should be avoided. Further references can be found in [24] where the following list originates.

1. **Pohlig-Hellman Algorithm:** This algorithm exploits the factorization of n, the order of the point P, and reduces the problem of recovering l to the problem of recovering l modulo each of the prime factors of n. We can then recover l by using the Chinese Remainder Theorem. As a countermeasure, one should select an elliptic curve whose order is a prime or almost a prime (i. e. a large prime times a small integer).

2. **Supersingular Elliptic Curves:** In some cases, the ECDLP in an elliptic curve E defined over a finite field $\mathbb{F}_q$ can be reduced to the ordinary discrete logarithm problem (DLP) in the multiplicative group of some extension field $\mathbb{F}_{q^k}$ for $k \geq 1$. The DLP is the underlying computationally hard mathematical problem for the DSA and one can solve it using the number field sieve algorithm, which has a sub-exponential running time. To ensure that the reduction algorithm does not apply to a particular curve, one only needs to check that n does not divide $q^k - 1$ for all small k for which the DLP in $\mathbb{F}_{q^k}$ is tractable. In practice, it suffices to check this for $1 \leq k \leq 20$ when $n > 2^{160}$.

3. **Prime-Field Anomalous Curves:** If the number of points on a curve E over $\mathbb{F}_p$ is equal to p, the ECDLP can be solved efficiently. This attack can be avoided by verifying that the number of points on an elliptic curve is not equal to the cardinality of the underlying field.

4. **Curves Defined Over a Small Field:** For elliptic curves E with coefficients in $\mathbb{F}_{2^e}$, Pollard's rho algorithm for computing elliptic curve logarithms in $E(\mathbb{F}_{2^{ed}})$ can be further sped up by a factor of $\sqrt{d}$. For example, if E is a Koblitz curve, then Pollard's rho algorithm for computing elliptic curve logarithms in $E(\mathbb{F}_{2^m})$ can be sped up by a factor of $\sqrt{m}$.

5. **Curves Defined Over $\mathbb{F}_{2^m}$, m Composite:** The Weil descent might be used to solve the ECDLP for elliptic curves defined over $\mathbb{F}_{2^m}$ where m is composite. There exists some evidence that when m has a small divisor l, e.g. $l = 4$, the ECDLP can be solved faster than with Pollard's rho algorithm. Thus, elliptic curves over composite fields should not be used.

Finally, let us examine how secure elliptic curve cryptography is in practice. One source that gives a notion about the security of ECC is the Certicom challenge [42]. The challenge is to compute the ECC private keys from a given list of ECC public keys and associated system parameters. The challenge has been issued in November 1997 and consists of two levels:

- Level I: 109-bit and 131-bit challenge; considered to be feasible
- Level II: 163-bit, 191-bit, 239-bit and 359-bit challenge; expected to be computationally infeasible

Of the Level I challenges, the 109-bit ECC2K-108 challenge has been solved in April 2000 and the 109-bit ECCp-109 challenge has been solved in November 2002. All other challenges are (until March 2003) unsolved. According to Certicom, the computational cost to solve these challenges met the expected values.

Concrete recommendations based on the expected cost of the Certicom challenge have been presented by Lenstra and Verheul in [33]. Using a model that incorporates technological and cryptanalytical advances they predict which elliptic curve key sizes can be considered as secure until which year. Elliptic curves with a key size of 140-bit, for example, shall provide a sufficient level of security for commercial applications until the year 2003. The authors assume that a computational cost of about $3.5 \cdot 10^{10}$ Mips years can be considered to be infeasible until the year 2003. Table 5.5.1 presents their recommendations for future years.

Table 5.5.1 Minimum key size for elliptic curve cryptosystems providing a sufficient level of security [33].

Year	Elliptic Curve Key Size	Infeasible Number of Mips Years	Corresponding Number of Years on 450MHz Pentium II PC
2002	139	$2.06 \cdot 10^{10}$	$4.59 \cdot 10^{7}$
2003	140	$3.51 \cdot 10^{10}$	$7.80 \cdot 10^{7}$
2004	143	$5.98 \cdot 10^{10}$	$1.33 \cdot 10^{8}$
2005	147	$1.02 \cdot 10^{11}$	$2.26 \cdot 10^{8}$
2006	148	$1.73 \cdot 10^{11}$	$3.84 \cdot 10^{8}$
2007	152	$2.94 \cdot 10^{11}$	$6.54 \cdot 10^{8}$
2008	155	$5.01 \cdot 10^{11}$	$1.11 \cdot 10^{9}$
2009	157	$8.52 \cdot 10^{11}$	$1.89 \cdot 10^{9}$
2010	160	$1.45 \cdot 10^{12}$	$3.22 \cdot 10^{9}$
2011	163	$2.47 \cdot 10^{12}$	$5.48 \cdot 10^{9}$
2012	165	$4.19 \cdot 10^{12}$	$9.32 \cdot 10^{9}$
2013	168	$7.14 \cdot 10^{12}$	$1.59 \cdot 10^{10}$
2014	172	$1.21 \cdot 10^{13}$	$2.70 \cdot 10^{10}$
2015	173	$2.07 \cdot 10^{13}$	$4.59 \cdot 10^{10}$
2016	177	$3.51 \cdot 10^{13}$	$7.81 \cdot 10^{10}$
2017	180	$5.98 \cdot 10^{13}$	$1.33 \cdot 10^{11}$
2018	181	$1.02 \cdot 10^{14}$	$2.26 \cdot 10^{11}$
2019	185	$1.73 \cdot 10^{14}$	$3.85 \cdot 10^{11}$
2020	188	$2.94 \cdot 10^{14}$	$6.54 \cdot 10^{11}$
2021	190	$5.01 \cdot 10^{14}$	$1.11 \cdot 10^{12}$
2022	193	$8.52 \cdot 10^{14}$	$1.89 \cdot 10^{12}$

6. Implementation

Following the results of our analysis of the secure charging protocol in Chapter 4 and equipped with the background information about elliptic curve cryptography, we developed an efficient implementation of the elliptic curve digital signature algorithm and the appertaining arithmetic operations on the Sharp Zaurus PDA platform. We provided an interface for the OpenSSL crypto library [3] allowing for easy integration into the prototype implementation of the secure charging protocol as well as other applications. As programming language for the implementation we chose ANSI C resulting in an easy portability of the code to other platforms.

As mentioned in Section 5.1.2, the finite field $\mathbb{F}_{2^m}$ with a polynomial basis representation of its elements is well-suited to modern microprocessor architectures. Consequently, we chose elliptic curves over $\mathbb{F}_{2^m}$ as curves for our ECDSA implementation and not elliptic curves over prime fields. Moreover, the authors of [19, 64] showed that Koblitz curves allow faster elliptic curve arithmetic and are well-suited for constrained target platforms. This is our motivation for implementing ECDSA routines that are optimized for Koblitz curves. Note, that a typical device in ad-hoc networks has only limited CPU power and that in protocol applications possibly a considerable amount of signature operations needs to be done a short time. Consequently, we focused on a fast and ressource saving implementation.

In this chapter, we describe which software development process we chose for our particular problem and give reasons for our design decisions. The chapter is organized as follows. We start with an introduction to our target platform in Section 6.1. A description of the software architecture, the development process and details of the implementation is contained in Section 6.2 and the following sections. We present our approaches to speed up the execution times of the signature operations in Section 6.7. The known limitations of our implementations together with the execution times on our target platform end this chapter.

6.1. Target Platform

Target platform for the implementation is a Sharp Zaurus SL-5500G Personal Digital Assistant (PDA) which is a typical device for wireless ad-hoc networks. According to [57], it is equipped with the Embedix™ Linux® 2.4 operating system and a StrongARM® (SA-1110) CPU clocked at 206 MHz. The Intel® SA-1110 CPU is based on the ARM V4 architecture described in [5]. The processor incorporates typical Reduced Instruction Set Computer (RISC) features such as:

- 31 general-purpose 32-bit registers. 16 of them are visible at a time, the others are used to speed up exception processing. Of the visible registers 14 can freely be used, the remaining two are reserved as program counter and return address storage
- A load/store architecture, where data-processing only operates on register contents, not directly on memory contents
- Uniform and fixed-length instruction fields with conditional execution feature
- Arithmetic Logic Unit (ALU) and shifter can be controlled in every processing step
- Load and Store Multiple instructions maximize data throughput

The 32-bit register architecture motivates the usage of 32-bit word variables throughout the implementation.

The size of the memory is specified as 64 MB RAM and 16 MB flash ROM. The RAM is further subdivided into 32 MB for processes, buffers, cache, and 32 MB for additional file and directory storage [56]. Compared to usual desktop PCs this is very few available memory, so one of the major goals of this thesis is to develop a fast but small implementation of the elliptic curve routines.

Our implementation is done in C. The compiler used for the development is a GNU cross-compiler supplied by Embedix^{TM} [2] that builds executables for the ARM V4 architecture and Linux OS.

6.2. Software Architecture

After analyzing the structure of ECDSA, we feel that it is the best choice to use the bottom-up approach for developing the implementation. Hence, we organize the software in the following layered model (Figure 6.2.1).

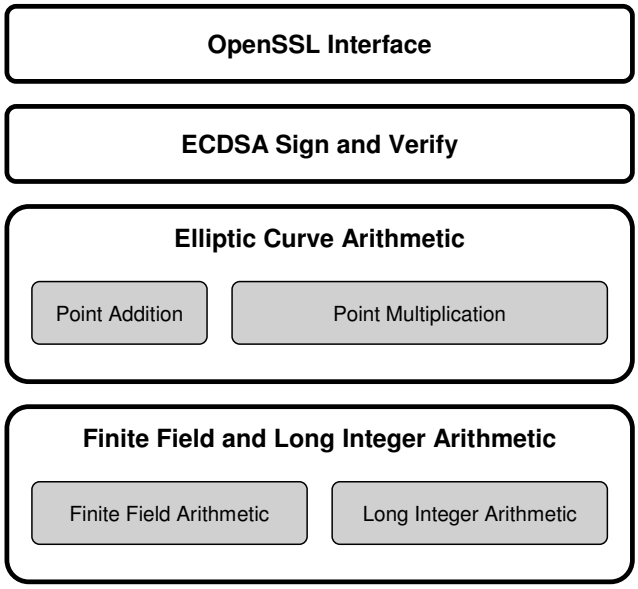
The layers can be described as follows, starting from the lowest:

1. **Finite Field and Long Integer Arithmetic Layer**
   The elliptic curve arithmetic is based on finite field arithmetic and in case of Koblitz curves, it also needs long integer arithmetic. This layer provides an efficient implementation of all necessary methods. It has a key influence on the performance of all higher layers, therefore we decided to tailor our implementation to a particular finite field. More details can be found in Section 6.3.
2. **Elliptic Curve Arithmetic Layer**
   The Elliptic Curve Arithmetic Layer includes implementations of several different methods for scalar multiplication of points on an elliptic curve, the implementation of point addition in different types of coordinates (used by multiplication methods and higher layers) and a couple of helper functions. Section 6.4 contains a detailed description of this layer.

3. **ECDSA Sign and Verify Layer**
   The next layer of our optimized implementation is the ECDSA Sign and Verify Layer. It provides lightweight interfaces for the sign and verify operations as well as initialization and precomputation routines. This layer is described in detail in Section 6.5.

4. **OpenSSL Interface Layer**
   This layer embeds the optimized ECDSA implementation into the existing OpenSSL framework. The OpenSSL framework is organized in a modular manner, therefore the integration of new methods or different implementations of existing methods is quite easy. We describe this process later on in Section 6.6.

Figure 6.2.1 Architecture Overview



The major advantage of implementing the lowest layer first and gradually combining modules is that testing and debugging becomes easier. The modules on the low layer can be tested and debugged separately from other components. Higher layer modules are not developed before the modules on lower layers function correctly according to their specifications. Tests of these higher layer modules can rely on the correctness of the incorporated modules.

From the optimization point of view, the bottom-up approach is also very attractive. The functions on the lower layers of the model form the time critical part of the code, since they are usually executed more often than those on higher layers. We have the possibility to determine the execution times of the isolated modules and can directly measure the influence of optimization approaches on their performance.

6.3. Finite Field Arithmetic in $\mathbb{F}_{2^m}$ and Long Integer Arithmetic

6.3.1. Finite Field Arithmetic

The following explanations and our implementation of finite field arithmetic are mostly based on [19] and [64]. The files `g2fm_inline_gen.h`, `g2fm_inline.c`, `g2fm_gen.h`, `g2fm_gen.c`, `g2fm.h`, and `g2fm.c` contain our implementation. The reasons for this file structure are explained in Section 6.7, which treats our optimization approaches.

Field representation

As mentioned in [19] and in Section 5.1.2 of this document, the polynomial basis representation of $\mathbb{F}_{2^m}$ with a trinomial or pentanomial as a reduction polynomial $f(x)$ appears to yield the simplest and fastest implementation in software. Therefore, we use this representation throughout our implementation of finite field arithmetic.

Consequently, we write an element $A \in \mathbb{F}_{2^m}$ as polynomial $A(x) = \sum_{i=0}^{m-1} a_i x^i$ and store it as binary vector $A = (a_{m-1}, \ldots, a_0)$. How to store these coefficients in computers is straightforward: since they are binary values, one can store each coefficient in one bit of memory. Due to the 32-bit register architecture of the StrongARM CPU, we use an array of $s = \lceil m/32 \rceil$ 32-bit words $(A[s-1], \ldots, A[0])$. The rightmost bit of $A[0]$ corresponds to a_0 and a_{m-1} is part of $A[s-1]$. The bits left of a_{m-1} are set to zero. For $m = 163$ an array of $s = 6$ 32-bit words would be sufficient.

We basically defined two types, namely `gf2mShortElement` and `gf2mLongElement`, that are arrays of s and $2s$ words. The later type holds intermediate results of multiplications or squarings, while the first one is used for all other field elements.

Addition

As stated in Section 5.1.2, addition of field elements is performed by bitwise XOR operations that will be denoted by $\oplus$. The ARM V4 architecture offers native 32-bit XOR instructions, thus addition should only require s operations.

Multiplication

The product $c = a \cdot b$ is computed by first evaluating $c'(x) = a(x) \cdot b(x)$ followed by modular reduction $c(x) \equiv c'(x) \mod f(x)$.

Polynomial Multiplication

To our knowledge, Algorithm 6.3.1 is the fastest method to compute $c = a \cdot b$. It does this by using a window method [38]. In Step 1, polynomials $B_u = u(x) \cdot b(x)$ are

precomputed for $0 \leq u \leq 2^w$ where w is the window size. This is done using a couple of left shifts and cumulative field element additions. In Steps 6 and 10 the m-bit vector B_u is added to C where the rightmost bit of B_u is added to the rightmost bit of $C\{j\}$. Here, $C\{j\}$ denotes the bit vector $(C[2s-2], \ldots, C[j])$ and the symbol $\ll$ denotes a bitwise left shift. The result of the algorithm is not reduced, so $c(x)$ is of degree at most $2m-2$.

Algorithm 6.3.1 Left-to-right comb method with windows of width $w = 4$ [19]

INPUT: Binary polynomials $a(x)$ and $b(x)$ of degree at most $m-1$.
OUTPUT: A binary polynomial $c(x) = a(x) \cdot b(x)$ of degree at most $2m-2$.

1: Compute $B_u = u(x) \cdot b(x)$ for all polynomials $u(x)$ of degree at most 3.
2: $C \longleftarrow 0$.
3: **for** $k = 7$ downto 1 **do**
4: **for** $j = 0$ to $s-1$ **do**
5: Let $u = (u_3, u_2, u_1, u_0)$, where u_i is bit $(4k+i)$ of $A[j]$.
6: $C\{j\} \longleftarrow C\{j\} \oplus B_u$.
7: $C\{j\} \longleftarrow C\{j\} \ll 4$.
8: **for** $j = 0$ to $s-1$ **do** {The case $k = 0$. This saves one comparison in the loop above.}
9: Let $u = (u_3, u_2, u_1, u_0)$, where u_i is bit i of $A[j]$.
10: $C\{j\} \longleftarrow C\{j\} \oplus B_u$.
11: **return** $c(x)$.

Modular Reduction

By choosing the reduction polynomial $f(x)$ to be a low weight polynomial, i.e. one with the least possible number of non-zero coefficients, reduction modulo $f(x)$ becomes a computationally cheap operation [6]. For cases of practical interest, $f(x)$ is a trinomial or pentanomial, i.e. the number of non-zero coefficients is 3 or 5. If the middle terms in $f(x)$ are close to each other, then reduction of $c(x)$ modulo $f(x)$ can be efficiently performed one word at a time. Suppose $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$. Then

$$\begin{aligned} x^{163} &\equiv x^7 + x^6 + x^3 + 1 \mod f(x) \\ &\vdots \\ x^{288} &\equiv x^{132} + x^{131} + x^{128} + x^{125} \mod f(x) \\ &\vdots \\ x^{324} &\equiv x^{168} + x^{167} + x^{164} + x^{161} \mod f(x) \end{aligned}$$

Hence, reduction can be performed by adding to C properly aligned all those $C[j]$ that contain the coefficients $c_k, 163 \leq k \leq 324$.

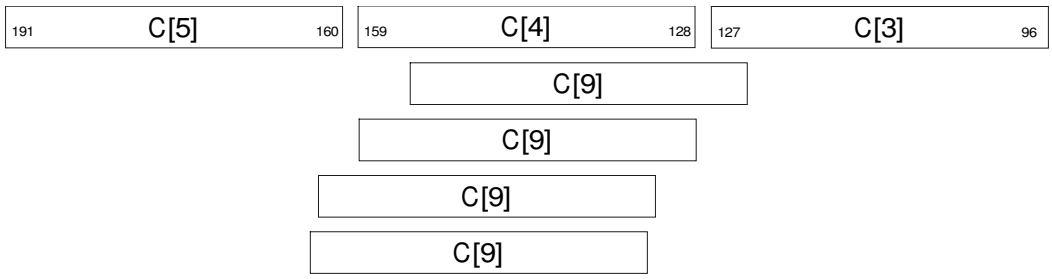
Let us clarify this with an example. Suppose we have already reduced C[10] and now want to reduce C[9], which contains bits 288 through 319. We can write the corresponding

polynomial as

$$\begin{aligned} c^{(9)}(x) &= \sum_{k=0}^{31} c_{k+288} \cdot x^{k+288} = x^{288} \sum_{k=0}^{31} c_{k+288} \cdot x^k \\ &\equiv (x^{132} + x^{131} + x^{128} + x^{125}) \cdot \underbrace{\sum_{k=0}^{31} c_{k+288} \cdot x^k}_{C[9]} \mod f(x) \end{aligned}$$

Now, we add the coefficients of x^i in $c^{(9)}(x)$ to the corresponding coefficient of x^i in C. Figure 6.3.1 shows how $C[9]$ is aligned and how the words $C[5]$ through $C[3]$ of C are affected.

Figure 6.3.1 This figure shows how the four shifted versions of $C[9]$ are aligned before being added to C.



The result of these thoughts is Algorithm 6.3.2. The left shift of bits is denoted by $\ll$ and the right shift by $\gg$. Due to the automatic code generator, our implementation supports different reduction polynomials. The generator automatically outputs the code appropriate for the chosen trinomial or pentanomial (see also Section 6.7.4).

Algorithm 6.3.2 Modular reduction in $\mathbb{F}_{2^m}$ by $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$ [19]

INPUT: A binary polynomial $c(x)$ of degree at most $2m - 2 = 324$.
OUTPUT: A binary polynomial $c(x) \mod f(x)$ of degree at most $m-1$.

1: **for** $i = 10$ downto 6 **do** {Reduce $C[i]$ modulo $f(x)$}
2: $T \longleftarrow C[i]$.
3: $C[i-6] \longleftarrow C[i-6] \oplus (T \ll 29)$.
4: $C[i-5] \longleftarrow C[i-5] \oplus (T \ll 4) \oplus (T \ll 3) \oplus T \oplus (T \gg 3)$.
5: $C[i-4] \longleftarrow C[i-4] \oplus (T \gg 28) \oplus (T \gg 29)$.
6: $T \longleftarrow C[5]$ **and** 0xFFFFFFF8. {Clear bits 0, 1 and 2 of $C[5]$.}
7: $C[0] \longleftarrow C[0] \oplus (T \ll 4) \oplus (T \ll 3) \oplus T \oplus (T \gg 3)$.
8: $C[1] \longleftarrow C[1] \oplus (T \gg 28) \oplus (T \gg 29)$.
9: $C[5] \longleftarrow C[5]$ **and** 0x00000007. {Clear the unused bits of $C[5]$.}

10: **return** $(C[5], C[4], C[3], C[2], C[1], C[0])$.

Squaring

Squaring in $\mathbb{F}_{2^m}$ can be done much faster than multiplying two arbitrary elements. It turns out to be a linear operation, because for $a(x) = \sum_{i=0}^{m-1} a_i x^i$ one finds

$$a^2(x) = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_i a_j x^{i+j} = \sum_{i=0}^{m-1} a_i x^{2i}$$

Hence, squaring can be done by simply inserting a 0-bit between consecutive bits of the binary representation of a. A fast way to do that is to use a table of precomputed values as shown in Algorithm 6.3.3 [54], which is a static table in our implementation. Hence, the precomputation step needs not to be done during run-time. The reduction step is done using Algorithm 6.3.2.

Algorithm 6.3.3 Squaring in $\mathbb{F}_{2^m}$ [54]

INPUT: A binary polynomial $a(x)$ of degree at most $m-1$.
OUTPUT: A binary polynomial $b(x) = a^2(x) \mod f(x)$ of degree at most $m-1$.

1: Precompute for each byte $v = (v_7, \ldots, v_1, v_0)$ the 16-bit vector $T(v) = (0, v_7, \ldots, 0, v_1, 0, v_0)$.
2: **for** $i = 0$ to $s-1$ **do**
3: Let $A[i] = (u_3, u_2, u_1, u_0)$ where each u_j is a byte.
4: $C[2i] \longleftarrow (T(u_1), T(u_0))$, $C[2i+1] \longleftarrow (T(u_3), T(u_2))$.
5: Compute $b(x) = c(x) \mod f(x)$. {using Algorithm 6.3.2.}
6: **return** b(x).

Modular division

Based on a paper by Sheueling Chang Shantz [55] Algorithm 6.3.4 is used to compute the modular division $\frac{a}{b}$ directly, because the running time is roughly the same as the one of the Extended Euclidean Algorithm that computes the modular inverse. By doing so, we save one modular multiplication. The comparison between two elements is done by treating the 32-bit words as unsigned integers where $A[i+1]$ is more significant than $A[i]$.

Algorithm 6.3.4 Modular Division in $\mathbb{F}_{2^m}$ [55]

INPUT: Binary polynomials $a(x)$ and $b(x) \neq 0$ of degree at most $m-1$.
OUTPUT: A binary polynomial $c(x) = \frac{a(x)}{b(x)} \mod f(x)$ of degree at most $m-1$.

```
1: A ⟵ b, B ⟵ f, U ⟵ a, V ⟵ 0.
2: while A ≠ B do
3:    if A is even then
4:       A ≫ 1.
5:       if U is odd then
6:          U ⟵ U ⊕ f.
7:       U ≫ 1.
8:    else if B is even then
9:       B ≫ 1.
10:      if V is odd then
11:         V ⟵ V ⊕ f.
12:      V ≫ 1.
13:   else if A > B then
14:      A ⟵ A ⊕ B
15:      A ≫ 1.
16:      U ⟵ U ⊕ V
17:      if U is odd then
18:         U ⟵ U ⊕ f
19:      U ≫ 1.
20:   else
21:      B ⟵ A ⊕ B
22:      B ≫ 1.
23:      V ⟵ U ⊕ V
24:      if V is odd then
25:         V ⟵ V ⊕ f
26:      V ≫ 1.
27: return U(x).
```

6.3.2. Long Integer Arithmetic

The implementation of point multiplication methods that are optimized for elliptic curves of Koblitz type requires long integer arithmetic. Long integer arithmetic consists of arithmetic operations with integers that are longer than the 32-bit integers that are supported by native CPU operations.

Long integer representation

We use the following structure to represent long integers:

```
typedef struct
{
    Word32 *data;
    bool bNegative;
    unsigned int nWordsUsed;
    unsigned int nWordsAllocated;
}  longint;
```

The member `data` points to an array that contains the absolute value of the long integer. The least significant bit is bit 0 of data[0] and the most significant bit is bit 31 of the last element of the array. For performance reasons, `data` points mostly to statically allocated memory (on the stack), however, sometimes we also dynamically allocate memory on the heap. The boolean `bNegative` contains the sign of the long integer, i.e. it is true if the integer is a negative number. We chose the absolute value / sign representation instead of the two's complement representation, because for a sign inversion, the two's complement representation requires as many inversions as there are words whereas the absolute value / sign representation requires only a change of the sign flag. The two members `nWordsAllocated` and `nWordsUsed` specify how many words of memory have been allocated for the absolute value of the long integer (the chunk of memory that `data` points to) and how much of it is really used. Hence, the last used element of the array is always `data[nWordsUsed - 1]`. This kind of design is necessary, on one hand for compatibility reasons (OpenSSL uses a similar representation) and on the other hand in order to represent long integers with many different lengths as they appear as intermediate results of the Koblitz curve arithmetic.

Addition

We implemented the addition algorithm as it is given in [45], Algorithm 14.7. However, since we have to do signed multiprecision integer arithmetic, our routine also contains a sign check that treats the following two cases separately:

1. *Both operands have the same sign:* Add the absolute values and set the sign of the result to the common sign.

2. *The signs of the operands are different:* Subtract the absolute value of the operand with the greater absolute value from the one with the smaller absolute value and set the sign of the result to the sign of the operand with greater absolute value.

Another difficulty is how to correctly determine carries in the C programming language. It does not offer direct access to the carry flag of the processor, so we use the following algorithm:

Algorithm 6.3.5 Addition with carry detection in the C programming language

INPUT: Two operands a and b, a carry flag c.
OUTPUT: The result of $a+b+c$ and a new carry flag c'.

1: Compute $s = a + b$ and set $c' = 0$.
2: **if** $s < a$ OR $s < b$ **then** {Check for an overflow.}
3: Set $r = s + c$ and set $c' = 1$.
4: **else**
5: **if** $c \neq 0$ **then** {Adding the carry flag could also cause an overflow.}
6: $r = s + c$.
7: **if** $r = 0$ **then** {Carry flag caused an overflow, thus, propagate carry.}
8: $c' = 1$
9: **return** r and c'.

For the subtraction, the situation is somewhat easier, because we can detect the overflow prior to the subtraction by comparing both operands. Here is our algorithm to correctly detect the borrows:

Algorithm 6.3.6 Subtraction with borrow detection in the C

INPUT: Two non-negative words a and b, a borrow flag c.
OUTPUT: The result of $a-b-c$ and a new borrow flag c'.

1: Compute $r = a - b - c$ and set $c' = c$.
2: **if** $a < b$ **then**
3: Set $c' = 1$.
4: **else**
5: **if** $a > b$ **then**
6: Set $c' = 0$.
7: **return** r and c'.

Multiplication

The multiplication algorithm for long integers is based on Algorithm 14.12 in [45]. We implemented a slightly modified version that can be used to only calculate partial results. This significantly increases the performance of the modular reduction algorithm, since it only needs partial results. The algorithm multiplies the absolute values of the long integers, the sign of the result is determined separately.

Algorithm 6.3.7 Long integer multiplication

INPUT: Two non-negative long integers a, b, their sizes (number of words) l_a, l_b, and the range $[l_r^{\text{low}}, l_r^{\text{high}}]$ of words of the result to be calculated.
OUTPUT: The result $r = a \cdot b$, but only the words $(r_{l_r^{\text{low}}}, r_{l_r^{\text{low}}+1}, \ldots, r_{l_r^{\text{high}}})$.

1: $r_k = 0, k \in [0, l_a + l_b - 1]$.
2: **for** $j = 0$ to $l_b - 1$ **do**
3: $\quad c = 0$.
4: $\quad$ **for** $i = \max(0, l_r^{\text{low}} - j)$ to $\min(l_r^{\text{high}} - j, l_a - 1)$ **do**
5: $\quad\quad (uv) = c + a_i \cdot b_j + r_{i+j}$.
6: $\quad\quad c = u$.
7: $\quad\quad r_{i+j} = v$.
8: $\quad$ **if** $i + j < l_r^{\text{high}}$ **then**
9: $\quad\quad r_{i+j} = c$.
10: **return** $(r_{l_r^{\text{low}}}, r_{l_r^{\text{low}}+1}, \ldots, r_{l_r^{\text{high}}})$.

Note that Step 5 of Algorithm 6.3.7 requires the 64-bit result of a multiplication of two 32-bit operands. Fortunately, the C Standard Library supports such an operation. It is also a native operation of the ARM Assembler Instruction set, hence, we do not expect this operation to cause a significant performance deterioration.

Division

We implemented Algorithm 14.20 of [45] for long integer division and also included the optimizations suggested by the authors. The optimized division algorithm looks like that:

Algorithm 6.3.8 Long integer division [45]

INPUT: Two non-negative long integers $a = (a_{l_a-1}, \ldots, a_1, a_0)$, $b = (b_{l_b-1}, \ldots, b_1, b_0)$, and their sizes (number of words) l_a, l_b.
OUTPUT: The result $r = \lfloor a/b \rfloor = (r_{l_a-l_b}, \ldots, r_1, r_0)$.

1: Set $r_j \longleftarrow 0, j \in [0, l_a - l_b]$.
2: **while** $a_{l_a-1} < 2^{30}$ **do** {Normalization}
3: $\quad a = a \ll 1$, $b = b \ll 1$.
4: **while** $a \geq 2^{32(l_a-l_b)}b$ **do**
5: $\quad r_{l_a-l_b} \longleftarrow r_{l_a-l_b} + 1$, $a \longleftarrow a - 2^{32(l_a-l_b)}b$.
6: **for** $i = l_a - 1$ downto l_b **do**
7: $\quad$ **if** $a_i = b_{l_b-1}$ **then**
8: $\quad\quad$ Set $r_{i-l_b} \longleftarrow 2^{32} - 1$.
9: $\quad$ **else**
10: $\quad\quad$ Set $r_{i-l_b} \longleftarrow \lfloor (2^{32}a_i + a_{i-1})/b_{l_b-1} \rfloor$.
11: $\quad$ **while** $r_{i-l_b}(2^{32}b_{l_b-1} + b_{l_b-2}) > 2^{64}a_i + 2^{32}a_{i-1} + a_{i-2}$ **do**

12: $r_{i-l_b} \longleftarrow r_{i-l_b} - 1$.
13: $a \longleftarrow a - r_{i-l_b} \cdot 2^{32(i-l_b)}b$.
14: **if** $a < 0$ **then**
15: Set $a \longleftarrow a + 2^{32(i-l_b)}b$ and $r_{i-l_b} \longleftarrow r_{i-l_b} - 1$.
16: **return** r.

Note that in Steps 10 and 11 of Algorithm 6.3.8, 64-bit multiplication and 64-bit division is used. These operations are supported by the C Standard Library and since long integer division is only rarely used in our routines, there is no optimization necessary.

Modular reduction

We implemented the Barret modular reduction algorithm as it is given in Algorithm 14.42 of [45]. This algorithm needs the precomputation of the quantity $\mu = \lfloor 2^{2\lfloor m/32 \rfloor \cdot 32}/m \rfloor$ where m is the bit size of the modulus. In our case the modulus is always the modulus of the elliptic curve domain, so we only need to compute μ once during the initialization phase of our library. This is the reason why we chose the Barret algorithm as reduction algorithm.

Algorithm 6.3.9 Barret modular reduction of long integers [45]

INPUT: Two positive long integers $x = (x_{2(l_m-1)}, \ldots, x_1, x_0)$, $m = (m_{l_m-1}, \ldots, m_1, m_0)$ (with $m_{l_m-1} \neq 0$), and $\mu = \lfloor 2^{2l_m \cdot 32}/m \rfloor$.
OUTPUT: The result $r = x \mod m$.
1: $q_1 \longleftarrow \lfloor x/2^{32(l_m-1)} \rfloor$, $q_2 \longleftarrow q_1 \cdot \mu$, $q_3 \longleftarrow \lfloor q_2/2^{32(l_m+1)} \rfloor$.
2: $r_1 \longleftarrow x \mod 2^{32(l_m+1)}$, $r_2 \longleftarrow q_3 \cdot m \mod 2^{32(l_m+1)}$, $r \longleftarrow r_1 - r_2$.
3: **if** $r < 0$ **then**
4: $r \longleftarrow r + 2^{32(l_m+1)}$
5: **while** $r \geq m$ **do**
6: $r \longleftarrow r - m$.
7: **return** r.

Modular inversion

The modular inversion algorithm we implemented is a variant of the Extended Euclidean Algorithm (EEA) and is given in [7].

Algorithm 6.3.10 Modular inversion of long integers [7]

INPUT: Two positive long integers $a = (a_{(l_a-1)}, \ldots, a_1, a_0)$, $m = (m_{l_m-1}, \ldots, m_1, m_0)$.
OUTPUT: The result $r = a^{-1} \mod m$.

1: $u \longleftarrow a$, $v \longleftarrow m$, $A \longleftarrow 1$, $C \longleftarrow 0$.
2: **while** $u \neq 0$ **do**
3: **while** u is even **do**
4: $u \longleftarrow u/2$.
5: **if** A is even **then**
6: $A \longleftarrow A/2$.
7: **else**
8: $A \longleftarrow (A + m)/2$.
9: **while** v is even **do**
10: $v \longleftarrow u/2$.
11: **if** C is even **then**
12: $C \longleftarrow C/2$.
13: **else**
14: $C \longleftarrow (C + m)/2$.
15: **if** $u \geq v$ **then**
16: $u \longleftarrow u - v$, $A \longleftarrow A - C$.
17: **else**
18: $u \longleftarrow v - u$, $C \longleftarrow C - A$.
19: **return** $C \mod m$.

6.3.3. Timings

Table 6.3.1 summarizes the execution time of the basic finite field arithmetic we implemented. We use the times obtained for a 163-bit Koblitz curve as an example, since the times for other extension degrees show a similar behavior. Obviously, multiplication and division dominate the execution times of the remaining operations.

Table 6.3.1 Execution times in microseconds of basic finite field arithmetic for $\mathbb{F}_{2^{163}}$ on a Sharp Zaurus at 206MHz and on an Intel Pentium II at 300 MHz.

Arithmetic Operation	Sharp Zaurus 206 MHz	Intel Pentium II 300 MHz
Addition	1µs	$<$ 1µs
Modular Reduction	2µs	1µs
Squaring	2µs	1µs
Multiplication	13µs	6µs
Division	116µs	60µs

6.4. Elliptic Curve Arithmetic

In this section, we describe our implementation of the elliptic curve arithmetic algorithms more in detail. In particular, we mention the difficulties we encountered and explain our design decisions.

6.4.1. Definitions

For our implementation, we define the points with the affine coordinates $(0, 0)$ and with the projective coordinates $(\cdot, \cdot, 0)$, i.e. the projective points with $z = 0$, to be the point at infinity $\mathcal{O}$.

6.4.2. Efficient Point Addition

Addition using affine coordinates

We implemented Equations (5.2) for adding two points that are given in affine coordinates. The implementation is straightforward, once the finite field arithmetic is working properly. However, in order to maximize the speed of the addition operations, we distinguish between the following special cases within our implementation (Figure 6.4.1):

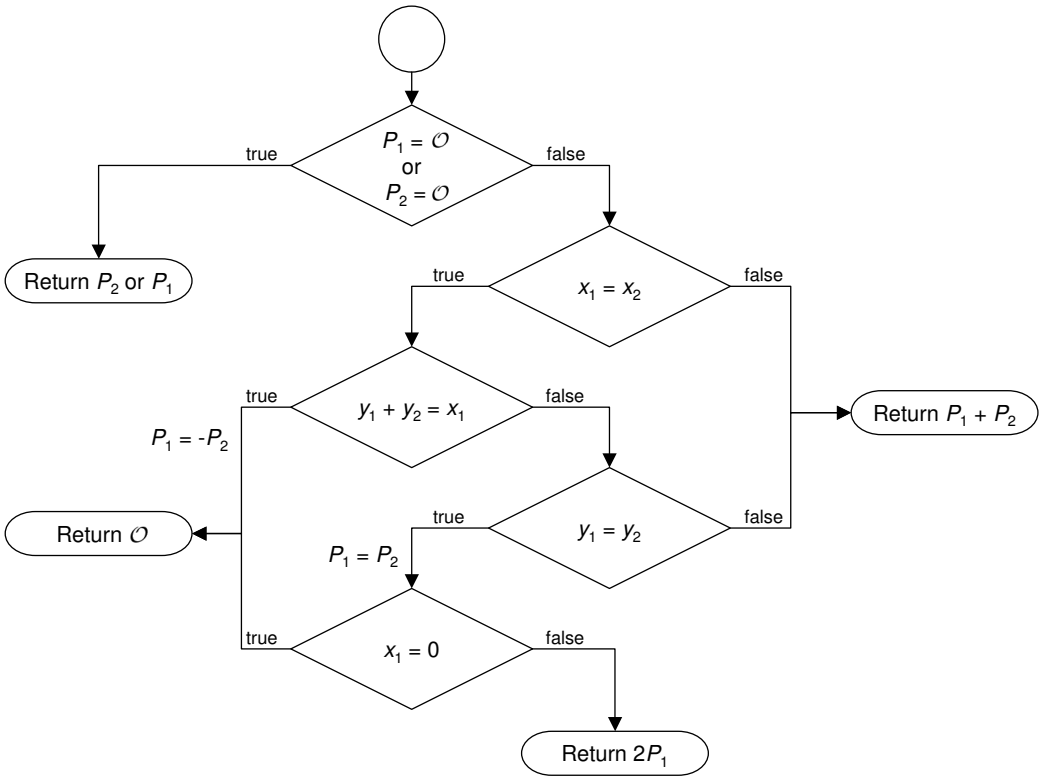
1. $P_1 = -P_2$ (result is $\mathcal{O}$),
2. $P_1 = P_2$ (point doubling, Equation (5.2)),
3. $P_1 = \mathcal{O}$ or $P_2 = \mathcal{O}$ (result is either P_2 or P_1),
4. general point addition (Equation (5.2)).

The third case can be detected relatively easy by comparing the points to our definition of the point at infinity. For $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ the other cases are

1. $P_1 = -P_2$ iff $x_1 = x_2$ and $y_1 = x_2 + y_2$,
2. $P_1 = P_2$ iff $x_1 = x_2$ and $y_1 = y_2$.

In the case of adding two points having the same coordinates, i.e. $x_1 = x_2$ and $y_1 = y_2$, the case of $x_1 = x_2 = 0$ has to be treated separately, because Equations (5.2) are not defined for this case (division by zero). In this case, the result is, by definition, the point at infinity $\mathcal{O}$.

Figure 6.4.1 Flow diagram of the decision process to distinguish the special cases for point addition in affine coordinates.



Addition using mixed coordinates

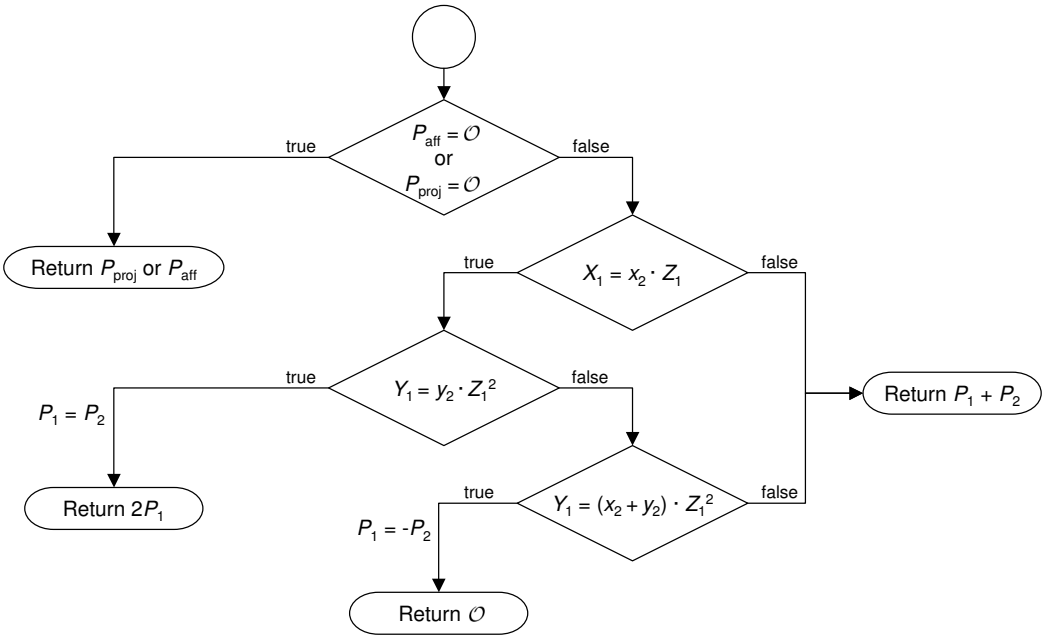
Analogue to the addition using affine coordinates, adding a point given in affine coordinates to a point given in projective coordinates can be realized by simply implementing Equations (5.4). We also distinguish the special cases mentioned above (Figure 6.4.2). Due to the representation in different coordinates, the conditions for these special cases have to be slightly modified. Suppose we are given a point $P_{\text{proj}} = (X_1, Y_1, Z_1)$ given in projective coordinates and a point $P_{\text{aff}} = (x_2, y_2)$ given in affine coordinates. Transferring P_{proj} to affine coordinates, we get $P_{\text{proj}} = (X_1/Z_1, Y_1/Z_1^2)$. Now, we can use the conditions for points in affine coordinates to derive the special cases for mixed coordinates.

These cases are

1. $P_{\text{proj}} = -P_{\text{aff}}$ iff $X_1 = x_2 \cdot Z_1$ and $Y_1 = (x_2 + y_2) \cdot Z_1^2$,

2. $P_{\text{proj}} = P_{\text{aff}}$ iff $X_1 = x_2 \cdot Z_1$ and $Y_1 = y_2 \cdot Z_1^2$.

Figure 6.4.2 Flow diagram of the decision process to distinguish the special cases for point addition in mixed coordinates.



6.4.3. Efficient Point Multiplication

Most of our implementation of the point multiplication methods has already been described in Chapter 5. However, since our sources do not explicitly cover the implementation of the precomputation steps needed for the windowed methods, we present our approaches in the following.

Fixed-base comb method

The very fast scalar point multiplication of the fixed-base comb method (Algorithm 5.3.5) is due to the fact that a great part of the computation is done only once for a base point of the elliptic curve and stored for the actual computation. In our implementation, the precomputation is done during the initialization step within the function `ecdsaInitDomain()`. This way, no further precomputation is necessary for the actual computation of scalar multiplications. The precomputation is done according to Algorithm 6.4.1.

Algorithm 6.4.1 Precomputation steps for the Fixed-base comb method for point multiplication

INPUT: Window width w, $P \in E(GF(2^m))$.
OUTPUT: $P_n = \sum_{i=0}^{w-1} n_i 2^{i \cdot d} P$ for $0 \leq n \leq 2^w - 1$.

1: $P_0 \longleftarrow \mathcal{O}$.
2: $P_1 \longleftarrow P$.
3: $j \longleftarrow 2$.
4: **while** $j \leq 2^{w-1}$ **do**
5: $\quad P_j \longleftarrow 2^d P_{j \gg 1}$.
6: $\quad$ **for** $r = 1$ to $j - 1$ **do**
7: $\quad\quad P_{j+r} \longleftarrow P_j + P_r$.
8: $\quad j \longleftarrow j \ll 1$.
9: **return** P_n for $0 \leq n \leq 2^w - 1$

The computation in Step 5 equals to d doublings of the point $P_{j \gg 1}$. Since doubling in projective coordinates is faster than in affine coordinates, we convert the point to projective coordinates before the computation and convert the result back to affine coordinates after the computation. The point addition in Step 7 is performed in affine coordinates, because the main multiplication routine needs the precomputed points given in affine coordinates.

Window TNAF method

The TNAF method and the window TNAF method for point multiplication on Koblitz curves require some amount of precomputation if several different curves shall be supported. Our implementation contains a function `ecpointInitKoblitzCurve()` that is called within the initialization function `ecdsaInitDomain()` if the chosen curve is a Koblitz curve. This function essentially allocates the required dynamic memory and computes the following quantities needed for the

- Partial Modular Reduction (Algorithm 5.4.2) [59]:
  - V_m,
  - $s_i = \frac{(-1)^i}{f}(1 - \mu U_{m+3-a-i})$ for $i \in \{0, 1\}$.
- Computation of $\text{TNAF}_w(\rho)$ (Algorithm 5.4.4) [59]:
  - U_w,
  - $t_w = 2U_{w-1}U_w^{-1} \pmod{2^w}$.

- Window TNAF method (Algorithm 5.4.5) [59]:
  - U_i for $0 < i < w$,
  - $\tau^i = U_i\tau - 2U_{i-1}$ for $0 < i < w$,
  - α_u and TNAF(α_u) using Algorithm 6.4.2.

In above equations, U_k and V_k are the Lucas sequences that can be computed via the recursion formulas (Equation (5.12)), w is the window width for the window TNAF method, m is the extension degree of the finite field $\mathbb{F}_{2^m}$, f is the cofactor and a is the parameter of the Koblitz curve E_a (see also Table 5.4.1).

Algorithm 6.4.2 Computation of α_u and TNAF(α_u)

INPUT: Window width w
OUTPUT: α_u, $\left(k_{w-1}^{(u)}, k_{w-2}^{(u)}, \ldots, k_0^{(u)}\right) = \text{TNAF}(\alpha_u)$ for $u \in \{1, 3, \ldots, 2^{w-1}-1\}$.

1: $\alpha_0 \longleftarrow 1$.
2: **for** $i = 1$ to 2^{w-2} **do**
3: $\quad \kappa \longleftarrow 2i + 1$ partmod $((\tau^m - 1)/(\tau - 1))$.
4: $\quad \left(k_{w-1}^{(2i+1)}, k_{w-2}^{(2i+1)}, \ldots, k_0^{(2i+1)}\right) \longleftarrow \text{TNAF}(\kappa)$.
5: $\quad \alpha_{2i+1} \longleftarrow 0$.
6: $\quad$ **for** $j = w - 1$ downto 0 **do**
7: $\quad\quad \alpha_{2i+1} \longleftarrow \alpha_{2i+1} + k_j^{(2i+1)} \cdot \tau^j$.
8: **return** α_u and $\left(k_{w-1}^{(u)}, k_{w-2}^{(u)}, \ldots, k_0^{(u)}\right)$ for $u \in \{1, 3, \ldots, 2^{w-1} - 1\}$.

The window TNAF method (Algorithm 5.4.5) requires the precomputation of $P_u = \alpha_u P$ for $u \in \{1, 3, \ldots, 2^{w-1} - 1\}$. In our implementation, we realize this precomputation with Algorithm 6.4.3.

Algorithm 6.4.3 Precomputation steps for the window TNAF method for point multiplication

INPUT: Window width w, $P \in E(GF(2^m))$, $(k_{w-1}^{(l)}, k_{w-2}^{(l)}, \ldots, k_0^{(l)}) = \text{TNAF}(\alpha_{2l+1})$ for $l \in \{0, 1, \ldots, 2^{w-2} - 1\}$
OUTPUT: $P_u = \alpha_u P_u$ for $u \in \{1, 3, \ldots, 2^{w-1} - 1\}$.

1: **for** $l = 0$ to $2^{w-2} - 1$ **do**
2: $\quad P_l \longleftarrow \mathcal{O}$.
3: $\quad$ **for** $j = w - 1$ downto 0 **do**
4: $\quad\quad$ **if** $k_j \neq 0$ **then**
5: $\quad\quad\quad$ **if** $k_j = 1$ **then**
6: $\quad\quad\quad\quad P_{2l+1} \longleftarrow P_{2l+1} + \tau^j P$.
7: $\quad\quad\quad$ **else**

8: $P_{2l+1} \longleftarrow P_{2l+1} - \tau^j P$.
9: **return** P_u for $u \in \{1, 3, \ldots, 2^{w-1} - 1\}$.

6.4.4. Timings

Table 6.4.1 Execution times in microseconds for point arithmetic on the elliptic curve `sect163k1` which is a Koblitz curve over $\mathbb{F}_{2^{163}}$. Times were obtained on a Sharp Zaurus at 206MHz and on an Intel Pentium II at 300 MHz.

Arithmetic Operation	Sharp Zaurus 206 MHz	Intel Pentium II 300 MHz
Point Addition (mixed coordinates)	121µs	64µs
Point Addition (affine coordinates)	158µs	83µs
Point Doubling	61µs	35µs
Point Multiplication Methods for General Curves		
Binary Method	13560µs	7440µs
Window NAF Method ($w = 4$)	10920µs	6000µs
Montgomery Method	8960µs	4840µs
Fixed-Base Comb Method ($w = 9$), without precomputation time	2680µs	1400µs
Point Multiplication Methods for Koblitz Curves		
TNAF Method	6720µs	3080µs
Window TNAF Method ($w = 4$)	5320µs	2400µs
Fixed-base Window TNAF Method ($w = 11$), without precomputation time	3160µs	1440µs

Table 6.4.1 reflects the execution times of point addition and multiplication on our target platform. The window sizes for the windowed multiplication methods correspond to the choices that will be discussed in Section 6.5. The performance gap between addition in affine and in mixed coordinates turns out to be greater than estimated in Section 5.3.1. The remaining times are better than our estimations. Table 6.4.2 shows the influence of different window sizes on the execution times of the fixed-base point multiplication methods. Note, that our implementation also supports curves other than the 163-bit Koblitz curve used in the tables. However, since the results for other curves show a similar behavior, we limit our presentation to the exemplary timings for the 163-bit Koblitz curve.

Table 6.4.2 Execution times in milliseconds of fixed-base point multiplication methods for different window sizes on the elliptic curve `sect163k1` which is a Koblitz curve over $\mathbb{F}_{2^{163}}$. Times were obtained on a Sharp Zaurus at 206MHz.

Window Width	Fixed-base Comb Method		Fixed-base Window TNAF Method	
	Precomputation	Multiplication	Precomputation	Multiplication
4			0.5ms	4.8ms
5			1.5ms	4.4ms
6	21.7ms	3.8ms	3.8ms	4.0ms
7	32.4ms	3.2ms	9.5ms	3.8ms
8	53.2ms	2.9ms	22.6ms	3.6ms
9	94.8ms	2.7ms	52.6ms	3.4ms
10	176.8ms	2.4ms	119.8ms	3.3ms
11	240.1ms	2.2ms	268.6ms	3.2ms
12	664.0ms	2.1ms	598.9ms	3.1ms

6.5. Implementation of the Elliptic Curve Digital Signature Algorithm (ECDSA)

Although the realization of the ECDSA according to Algorithms 3.3.2 (signature generation) and 3.3.3 (signature verification) is straightforward, it involves several design decisions, which we describe in this section. We explain which multiplication methods we chose and which parameter settings lead to the best performance. After introducing the flexible and modular structure of our implementation, we end this section with an elaboration on how strong random numbers for the ECDSA are obtained.

6.5.1. Optimal Point Multiplication Methods

Taking a look at Algorithms 3.3.2 and 3.3.3, one notices that the crucial operations are the scalar point multiplication kG in the first algorithm and the two scalar point multiplications u_1G and u_2Q in the later one. They are crucial, because they dominate the execution time of the algorithms. All other operations such as modular integer arithmetic take only a fraction of the time that scalar point multiplication on an elliptic curve takes.

The algorithms use two different types of scalar point multiplication:

1. multiplication of some scalar k with the fixed base point G
2. multiplication of some scalar l with an arbitrary point Q

For the first type of multiplication, we know the point G already during the initialization phase of our program. This point is declared together with the other domain parameters of the elliptic curve. Obviously, it would be advantageous to use a point multiplication method that provides very fast execution times for the actual multiplication while taking possibly longer to do the necessary precomputation. We have implemented two such multiplication methods, the fixed-base comb method (Algorithm 5.3.5) for general binary curves and the fixed-base window TNAF method (Algorithm 5.4.5) for Koblitz curves.

Figure 6.5.1 Execution times and precomputation times of different fixed-point multiplication methods on the Sharp Zaurus (163-bit Koblitz curve).

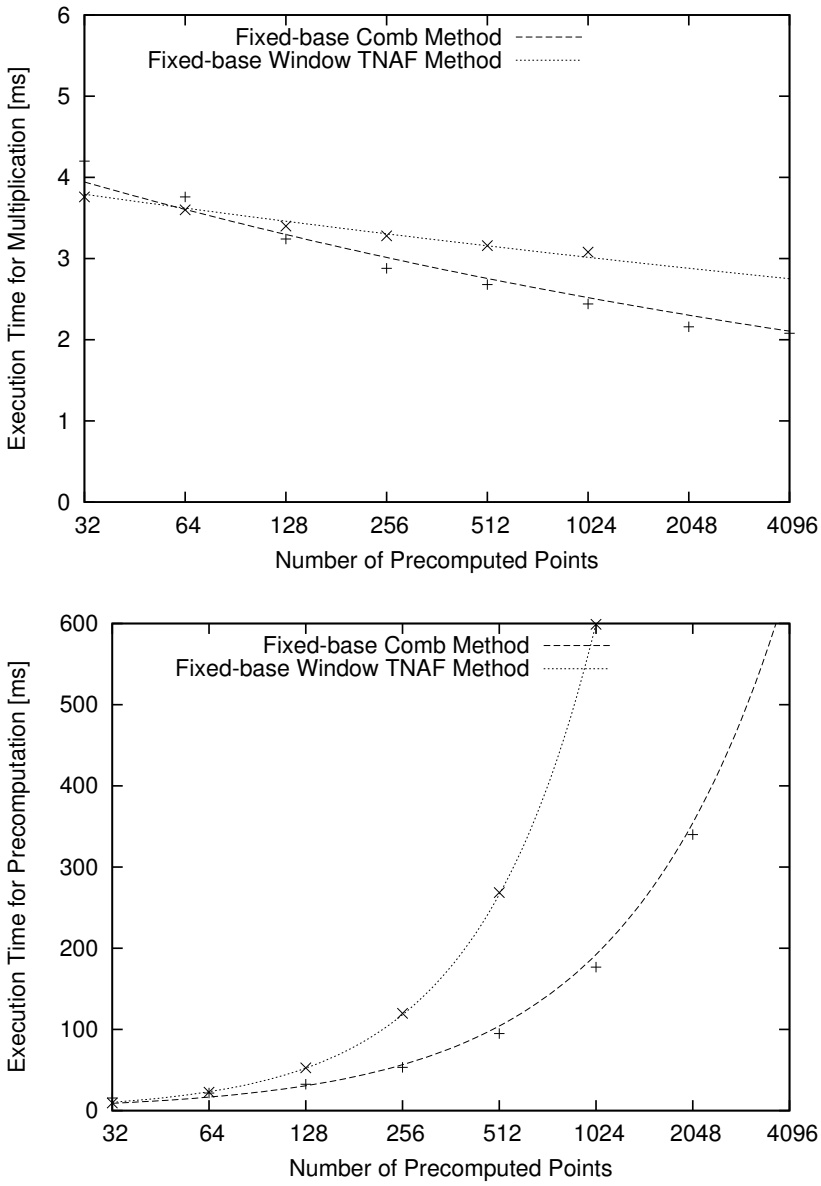
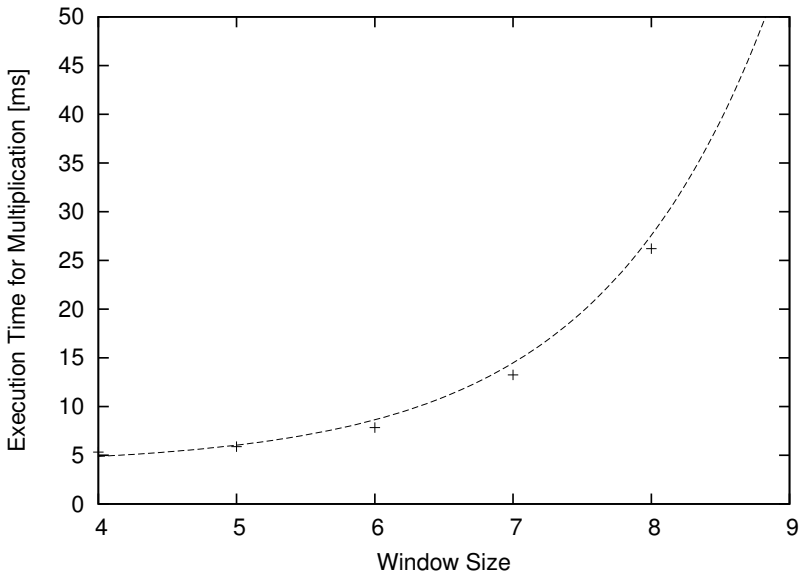


Figure 6.5.1 compares the timings of our implementation of these two methods depending on the number of precomputed points. Note, that the fixed-base comb method requires the precomputation of $2^{w_{\text{comb}}} - 1 \approx 2^{w_{\text{comb}}}$ points for a window width of w_{comb}, and that the fixed-base window TNAF method requires the precomputation of $2^{w_{\text{TNAF}}-2}$ points for a window width of w_{TNAF}. Obviously, the fixed-base comb method outperforms the fixed-base window TNAF method for greater window sizes. The reason for this behavior is that the fixed-base window TNAF method requires the computation of the width-w TNAF, which is computationally more expensive for greater window sizes than the simple bit reordering that the fixed-base comb method applies.

According to the timings of our implementation (see Figure 6.5.1), the fixed-base comb method with a precomputation of 512 points, which corresponds to a window-width w_{comb} of 9 leads to the best performance while still having an acceptable precomputation time. The figure clearly shows that further increasing the window size leads to a 9% faster multiplication time but almost doubles the precomputation time. The size of memory required for the precomputed points is proportional to the precomputation time. Thus, this is another reason for choosing the window-width to be 9.

Figure 6.5.2 Execution times (precomputation + multiplication) of the window TNAF method (163-bit Koblitz curve).



It turns out that for the second type of multiplication there are three possible methods, the Montgomery method, the comb method and the window TNAF method. Figure 6.5.2 demonstrates the execution times (total time of multiplication and precomputation) of the window TNAF method for different window sizes. Obviously, the optimal choice for the window size is 4, which is also the minimum value our implementation supports.

Finally, Table 6.5.1 gives an overview of the execution times of the window TNAF method, the Montgomery method and the comb method. These values suggest that the window TNAF method with a window-width of 4 is the best choice for Koblitz curves and the Montgomery method is the best choice for arbitrary curves.

Table 6.5.1 Execution times of different multiplication methods for arbitrary points on Sharp Zaurus (163-bit Koblitz curve).

Method	Execution Time (for precomputation and main computation)
Window TNAF ($w = 4$)	5.32 ms
Montgomery	8.96 ms
Comb ($w = 2$)	13.68 ms

Finally, we should mention that there is also a method called Shamir's trick [19] for simultaneously computing two point multiplications as they are necessary for the ECDSA verify operation (Algorithm 6.5.1). However, the performance of this method turns out to be worse than doing two multiplications and one addition using the methods discussed above (see Table 6.5.2).

Algorithm 6.5.1 Simultaneous point multiplication (Shamir's trick) [19]

INPUT: Window width w, $k = (k_{s-1}, \ldots, k_1, k_0)_2$, $l = (l_{s-1}, \ldots, l_1, l_0)_2$, P, Q.
OUTPUT: $kP + lQ$.

1: Compute $iP + jQ$ for all $i, j \in [0, 2^w - 1]$.
2: Write $k = (k^{d-1}, \ldots, k^1, k^0)$ and $l = (l^{d-1}, \ldots, l^1, l^0)$ where each k^i and l^i is a bitstring of length w, and $d = \lceil t/w \rceil$.
3: $R \longleftarrow \mathcal{O}$.
4: **for** $i = d - 1$ downto 0 **do**
5: $\quad R \longleftarrow 2^w R$.
6: $\quad R \longleftarrow R + (k^i p + l^i Q)$.
7: **return** R.

Table 6.5.2 Execution times of verify operation with Shamir's trick on Sharp Zaurus (163-bit Koblitz curve).

Window Width	Execution Time
without Shamir's trick	11.64 ms
$w = 2$	25.96 ms
$w = 3$	23.26 ms
$w = 4$	22.38 ms
$w = 5$	23.42 ms
$w = 6$	27.10 ms

Table 6.5.3 summarizes which multiplication methods we chose for which case together with the optimal window sizes.

Table 6.5.3 Overview of the used multiplication methods

	General Curves	Koblitz Curves
Arbitrary Point	Montgomery Method	Window TNAF method ($w = 4$)
Fixed Point	Fixed-base Comb Method ($w = 9$)	

6.5.2. Modular Architecture

In order to offer the chance to use different multiplication methods for different types of curves, we organize the ECDSA layer of our implementation in a modular way. This is done with a structure of type `ecdsaDomain` that is initialized during run-time with values tailored to the elliptic curve to be used. Here is how this structure looks like:

```
typedef struct
{
    EllipticCurvePoint G;
    gf2mShortElement n;
    gf2mLongElement mu;
    EllipticCurve *curve;

    // Precomputed Points
    EllipticCurvePoint *FixedBasePrecomputedPoints;
    EllipticCurvePoint *SimultaneousPrecomputedPoints;

    // Function pointers to the arithmetic methods to be used
```

```
    void (*ecpointMultArbitrary)();
    void (*ecpointMultGenerator)();
    void (*ecpointMultSimultaneous)();
} ecdsaDomain;
```

It contains some elliptic curve domain parameters, the precomputed value `mu` that is used for long integer modular reduction (see Section 6.3.2 for details), two arrays of precomputed points for the multiplication methods that use precomputation and three function pointers. In the function `ecdsaInitDomain()`, these function pointers are set to point to the optimal multiplication methods depending on the type of elliptic curve as discussed above. The precomputation for these methods is done in the initialization function. The memory for the arrays of precomputed points is allocated dynamically. We provide the function `ecdsaFreeDomain()` to free the allocated memory.

6.5.3. Random Number Generation

The signature generation and certainly the key generation according to ECDSA require the generation of random numbers. These random numbers must be strong, i.e. an attacker should not be able to predict the random number or parts of it. Otherwise the signature system would be vulnerable to attacks as described in [26]. Since the OpenSSL library already contains a well-designed random number generator, we use the output of the OpenSSL function `RAND_bytes()` for our random numbers. To ensure that the obtained number is not greater than the modulus n, we perform a modular reduction step.

6.6. Integration into OpenSSL

For the integration into OpenSSL we used the following OpenSSL struct, which is defined in the file `openssl/ec.h`:

```
typedef struct ecdsa_method
{
    const char *name;
    ECDSA_SIG *(*ecdsa_do_sign)(const unsigned char *dgst, int dgst_len,
                                EC_KEY *eckey);
    int (*ecdsa_sign_setup)(EC_KEY *eckey, BN_CTX *ctx, BIGNUM **kinv,
                            BIGNUM **r);
    int (*ecdsa_do_verify)(const unsigned char *dgst, int dgst_len,
                           ECDSA_SIG *sig, EC_KEY *eckey);
    int flags;
    char *app_data;
} ECDSA_METHOD;
```

Our implementation of this struct as well as the interface functions can be found in the file `openssl_interface.c`. Here is how it looks like:

```
static ECDSA_METHOD rubcosy_ecdsa_meth = {
    "RubCoSy ECDSA method",
    rubcosy_ecdsa_do_sign,
    rubcosy_ecdsa_sign_setup,
    rubcosy_ecdsa_do_verify,
    0,    /* flags    */
    NULL  /* app_data */
};
```

The declarations of the functions we implemented are in the file `ecdsa.h`. We implemented our own version of the `ecdsa_do_sign()`-method, the `ecdsa_sign_setup()`-method, and the `ecdsa_do_verify()`-method. The implementation basically converts the OpenSSL types `EC_KEY`, `BIGNUM`, and `ECDSA_SIG` to the corresponding lightweight types of our ECDSA implementation and calls the corresponding functions of our implementation. It also calls the initialization routines whenever necessary.

Figure 6.6.1 Call Graph for the `ECDSA_do_sign()`-method.

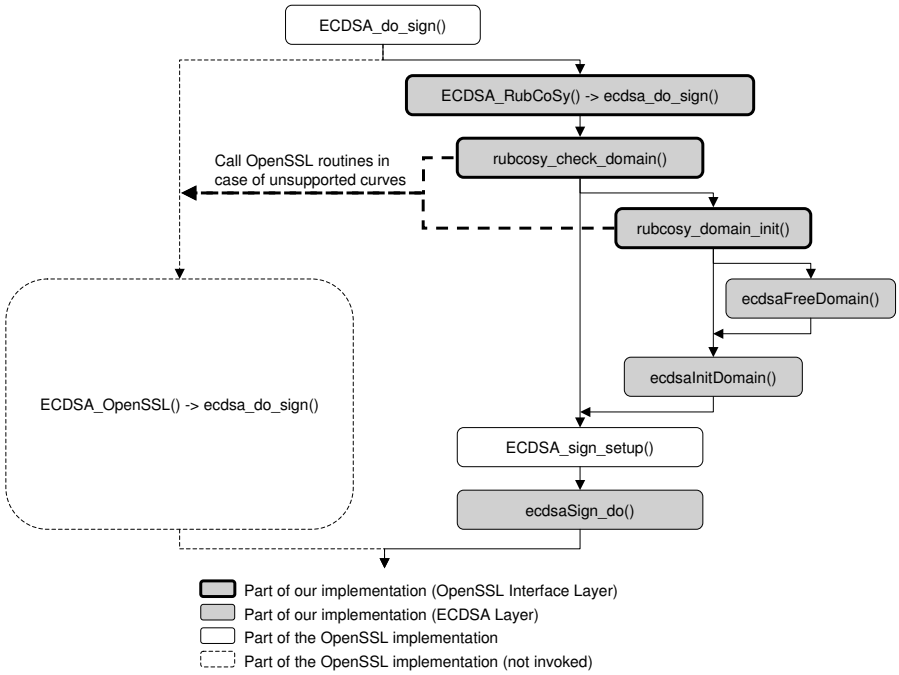
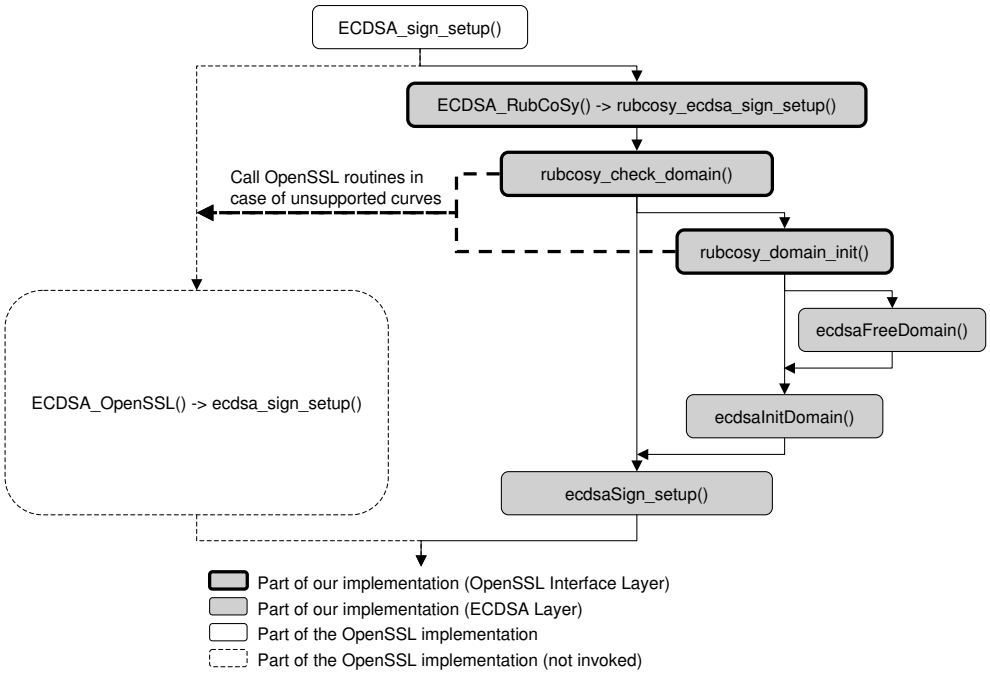


Figure 6.6.1 illustrates what happens when the OpenSSL framework invokes the ECDSA signature generation. The OpenSSL function `ECDSA_do_sign()` calls the member function `ecdsa_do_sign()` of the `ECDSA_METHOD`-struct. If we replace the default struct with

our implementation, our version of the `ecdsa_do_sign()`-function is invoked. Then, the functions `rubcosy_check_domain()` and `rubcosy_domain_init()` check whether our implementation supports the requested elliptic curve. If not, the corresponding member function of the standard OpenSSL method struct is invoked. However, in case we do support the elliptic curve, the domain parameters are initialized (i.e. dynamic memory is allocated, precomputation is done). Then, our implementation behaves similar to OpenSSL and first calls the OpenSSL function `ECDSA_sign_setup()`, followed by the `ecdsaSign_do()`-method, which is part of our ECDSA Sign and Verify Layer.

Figure 6.6.2 Call Graph for the `ECDSA_sign_setup()`-method.



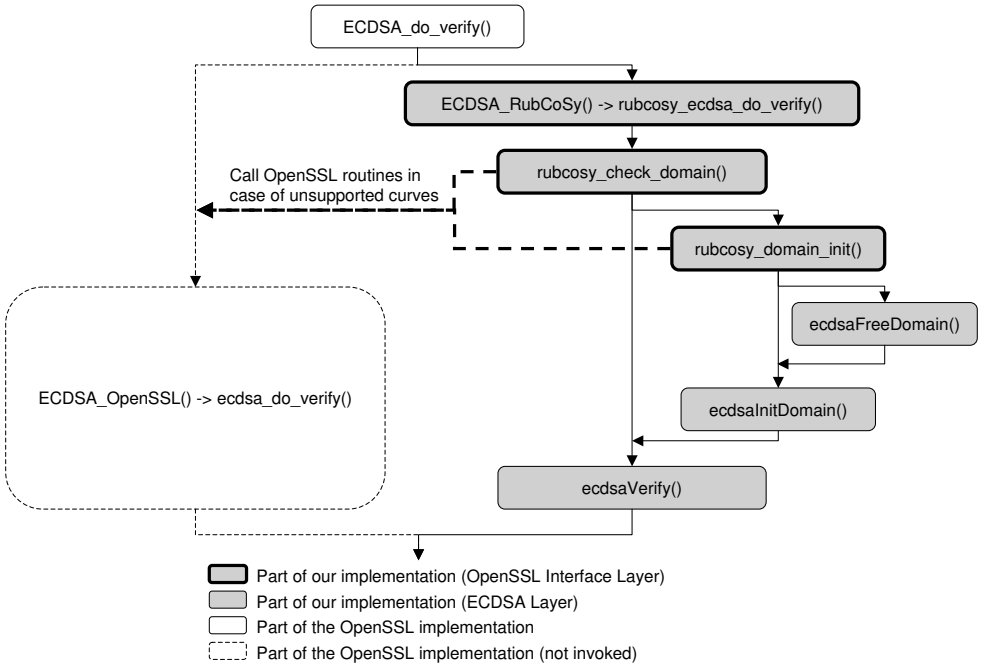
The OpenSSL function `ECDSA_sign_setup()` behaves similar to `ECDSA_do_sign()` (Figure 6.6.2) and calls, in case the elliptic curve is supported, the `ecdsaSign_verify()`-method, which is again part of our ECDSA Sign and Verify Layer.

Finally, Figure 6.6.3 describes which functions are called when the OpenSSL framework performs a signature verification. The actual verification is done using our `ecdsaVerify()`-method.

How can a program tell OpenSSL to use our optimized algorithms instead of the OpenSSL algorithms? We provide the function `ECDSA_RubCoSy()` which returns a pointer to the `ECDSA_METHOD` struct mentioned before. This pointer can be passed to the OpenSSL function `void ECDSA_set_default_method(const ECDSA_METHOD *)` in order to inform the OpenSSL framework that our optimized implementation shall be used instead of the OpenSSL implementation. This can be done in the global initialization section of the

program. The function `rubcosy_domain_free()` should be called in the global finalization section of the program in order to free all memory dynamically allocated by our implementation.

Figure 6.6.3 Call Graph for the `ECDSA_do_verify()`-method.



6.7. Optimizations

To increase the performance of our implementation, we primarily tried three different optimization methods. Of course, we also paid attention to producing efficient and low-overhead code throughout the whole implementation. However, we expected the three approaches

- inline functions
- loop unrolling
- inline assembler

to significantly increase the execution speed of our verify and sign operations. In this chapter we describe these three approaches in detail. Additionally, we present our automatic source code generator, which enables us to generate source code that is tailored to specific elliptic curves with specific key sizes. Hence, our implementation is not tied to one particular curve but can be configured to support any binary curve without loss of performance.

6.7.1. Inline Functions

Inline functions are an extension to the C programming language and allow the programmer to direct the compiler to integrate the function's code into the code of its callers. This eliminates the function-call overhead and might also improve the performance of the compiler-optimized code. The trade-off is, however, that the object code may become larger. Actually, the only difference to the conventional macro method which uses `#define`-statements to replace normal functions is that the type checking during compilation is not disabled. Furthermore, debugging the code is much easier, because the inlining can be disabled with a compiler switch leading to normal function code.

In our implementation, source files with the term `inline` within their filename contain the source code for inline functions. It was necessary to put them into separate files, because the compiler needs to see their source while compiling the source of the callers, i.e. the source of the inline functions has to be included in all source files that contain functions that call these inline functions.

The typical functions that we declared as inline functions are utility functions, e.g. functions that assign the coordinates of a point to another point or functions that compare the absolute values of two long integers. Nevertheless, using our automatic source code generation tool, we could compare the speed of a number of combinations and choose the optimal combination (see also Section 6.7.4).

6.7.2. Loop Unrolling

The largest gain in performance can be achieved with so-called loop unrolling. With this method, loops in the source code like

```
for (i = 0; i < 10; i++)
    foo[i] = 0;
```

are unrolled, i.e. their code is written as

```
foo[0] = 0;
foo[1] = 0;
foo[2] = 0;
foo[3] = 0;
...
foo[9] = 0;
```

In modern processors with pipeline architecture short loops significantly deteriorate the execution speed. The reason for this is that a loop is essentially a number of instructions

and a conditional branch instruction at the end. The conditional branch evaluates an expression that changes within the loop body. In our example above, the branch instruction checks whether the variable `i` is less than 10 after each loop cycle. In a pipeline architecture the processor cannot load the next instruction into the pipeline as long as the expression is not evaluated, because it does not know which will be the next instruction (the first instruction of the loop or the instruction following the loop).

Imagine, for example, we have a 4-stage pipeline with the following stages:

1. **Fetch (F):**
   The next instruction is fetched from the instruction buffer.

2. **Decode (D):**
   The instruction is decoded, i.e. the processor examines which operands will be needed and what shall be done with them.

3. **Execute (E):**
   The processor executes the instruction, e.g. adds two operands.

4. **Write back (W):**
   The result of the execution step is written to the register file and is now available for following instructions.

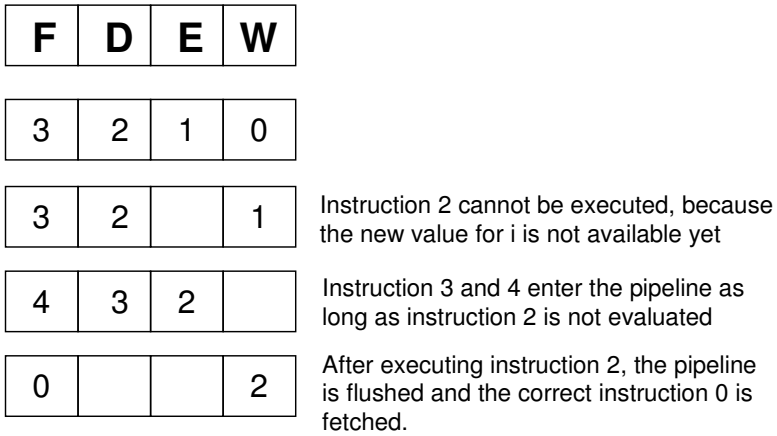
The CPU instructions implementing the loop above written in pseudocode might be

```
0: foo[i] = 0;
1: i = i + 1;
2: if (i < 10) goto 0;
3: ...
4: ...
```

As Figure 6.7.1 illustrates, the conditional jump in instruction 2 cannot be executed before the result of instruction 1 is written back to the register file. Moreover, it depends on the result of instruction 2 whether instruction 3 or instruction 0 is executed next. Hence, instruction 0 will be fetched after instruction 2 is executed and two holes (`NOP`-operations) are inserted into the pipeline.

Unrolling the loops reduces the number of branch instructions within the code, so that the pipeline can be filled continuously with instructions. Hence, the instruction throughput and therefore the execution speed of the operations rises.

Figure 6.7.1 Stalled pipeline, because the result of a previous instruction is not available on time.



In our case, unrolling the loops more than doubled the execution speed of the signature and verify operations. However, we did not unroll every loop. For example, in the algorithm for multiplying two elements of $\mathbb{F}_{2^m}$ (Algorithm 6.3.1), there are two nested loops (Steps 3 and 4). It turns out, that unrolling the inner loop leads to an 18% performance gain whereas unrolling both loops deteriorates the performance. For this reason, we used our automatic source code generation utility to determine the level of unrolling that leads to the best overall performance.

Together with the inline functions, the loop unrolling optimizations sped up the execution of the Montgomery point multiplication method from originally 25.2ms on the Sharp Zaurus to 8.96ms. This is equivalent to an acceleration factor of 2.8. This demonstrates the significance of this optimization method.

6.7.3. Inline Assembler

The compiler we utilized, the Gnu GCC compiler, supports so-called inline assembler statements, which embed assembler statements directly in the source code. This enables the programmer to directly access native processor commands that are not supported by the C programming language, e.g. bit shift operations or add-with-carry operations. Additionally the programmer has direct control over the compiler output and may hand-optimize the order of instruction execution.

Figure 6.7.2 Multi-word bitwise right shift.

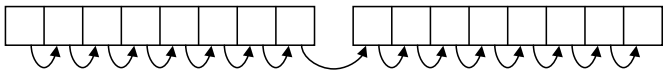


Table 6.7.1 Implementation of the multi-word right shift operation in C and Assembler.

C Source Code	Compiler Generated Assembler	Hand-written Assembler

`a[0] = a[0] >> 1;`
`a[0] = a[0] ^`
`(a[1] << 31);`
	`ldr r3, [%0]`
	`ldr r2, [%0,#4]`
	`mov r3, r3, LSR #1`
	`eor r3, r3, r2, LSL #32`
	`str r3, [%0]`
		`ldr r0, [%0,#20]`
		`movs r0, r0, LSR #1`
		`str r0, [%0,#20]`

`a[1] = a[1] >> 1;`
`a[1] = a[1] ^`
`(a[2] << 31);`
	`ldr r1, [%0,#8]`
	`mov r2, r2, LSR #1`
	`eor r2, r2, r1, LSL #32`
	`str r2, [%0,#4]`
		`ldr r0, [%0,#16]`
		`movs r0, r0, RRX`
		`str r0, [%0,#16]`

`a[2] = a[2] >> 1;`
`a[2] = a[2] ^`
`(a[3] << 31);`
	`ldr r3, [%0,#12]`
	`mov r1, r1, LSR #1`
	`eor r1, r1, r3, LSL #32`
	`str r1, [%0,#8]`
		`ldr r0, [%0,#12]`
		`movs r0, r0, RRX`
		`str r0, [%0,#12]`

`a[3] = a[3] >> 1;`
`a[3] = a[3] ^`
`(a[4] << 31);`
	`ldr r2, [%0,#16]`
	`mov r3, r3, LSR #1`
	`eor r3, r3, r2, LSL #32`
	`str r3, [%0,#12]`
		`ldr r0, [%0,#8]`
		`movs r0, r0, RRX`
		`str r0, [%0,#8]`

`a[4] = a[4] >> 1;`
`a[4] = a[4] ^`
`(a[5] << 31);`
	`ldr r1, [%0,#20]`
	`mov r2, r2, LSR #1`
	`eor r2, r2, r1, LSL #32`
	`str r2, [%0,#16]`
		`ldr r0, [%0,#4]`
		`movs r0, r0, RRX`
		`str r0, [%0,#4]`

`a[5] = a[5] >> 1;`	`mov r1, r1, LSR #1`
	`str r1, [%0,#20]`
		`ldr r0, [%0]`
		`movs r0, r0, RRX`
		`str r0, [%0]`

We developed hand-written assembler code for some functions that we supposed to perform better when written in assembler. Examples for such functions are shift operations for finite field elements. While performing a multi-word bitwise right shift, one wants to shift all bits in all words right by one bit and the least significant bit of the more significant words shall be inserted as most significant bit of the less significant words (Figure 6.7.2). There is an ARM assembler instruction for such an operation, the rotate right instruction (`RRX`). Table 6.7.1 contains an implementation of the right shift operation in C, the assembler output of the compiler and a hand-written assembler implementation. Ob-

viously, the compiler-generated code consists of 23 statements whereas the hand-written code consists of only 18 statements. Hence, we could theoretically increase the execution speed by about 20%.

However, it turned out that the performance achieved by using native assembler instructions for such operations is worse than the performance the compiler optimized code achieves. The reason for this is that the compiler does a lot of code reordering, i.e. it changes the order of execution of some statements, and thereby optimizes register usage and memory access. Moreover, the compiler does not only consider the relatively short function that implements the shift operation, but also the longer function that calls the shift function. Remember, we use inline functions, therefore the compiler looks at the whole code including the replaced versions of the shift function during optimization. Unfortunately, using inline assembler prevents the compiler from performing such an optimization, because the hand-written assembler sections are included after the optimization stage.

Hence, in order to further speed up the code, it would be necessary to hand-write larger parts of the code in assembler and hand-optimize register usage and memory access. We believe that the amount of time necessary to do this is not justified by the expected gain of speed and loss of portability. Consequently, our implementation does not use any inline assembler.

6.7.4. Automatic Source Code Generator

The reasons for us to develop an automatic source code generator are essentially the following:

- Loop unrolling for different key sizes cannot be done in C (for example, with `#define`-macros).
- Different target platforms or different curves might require different levels of loop unrolling or inline functions for maximum execution speed.
- The modular reduction algorithm in $\mathbb{F}_{2^m}$ (Algorithm 6.3.2) heavily depends on the reduction polynomial, which varies for different curves.

Our generator creates the source files `gf2m_gen.h`, `gf2m_gen.c` and `gf2m_inline_gen.c`. These files contain the definitions for the finite field arithmetic such as the reduction polynomial to be used or the extension degree m. Moreover, all finite field arithmetic operations are generated, since they contain loops that are unrolled. The generator also creates a version of the modular reduction algorithm (Algorithm 6.3.2) that is tailored to a specific reduction polynomial. The tool can be configured via command line options or via the configuration file `ecclib.conf`. For more details about the usage of the generator and the meaning of the parameters, we refer to Appendix A.1.2.

6.8. Known Limitations

For performance reasons our implementation currently has the following limitation:

- Our implementation only supports one particular curve at a time. However, if signature operations using an unsupported curve are requested via the OpenSSL interface, the arguments are passed to the original OpenSSL functions, which should support the requested curve.

  Moreover, the code generation tool can be used to change the curve to be supported before compilation.

6.9. ECDSA Timings

In this section, we present the execution times for ECDSA signature generation and verifation of our implementation. The measurements are obtained using the C function `clock()`. We call this function before and after a block of functions to be measured. The execution time in microseconds can be calculated by dividing the difference Δt_{Clock} by the constant `CLOCKS_PER_SEC`.

Table 6.9.1 presents the execution times of our implementation compared to the OpenSSL implementation. Obviously, our approach of using efficient algorithms tailored to a particular curve yields a performance gain of a factor of about 4–6 for the signature generation and 3–6 for the signature verification.

Table 6.9.1 Execution times in milliseconds for signature operations obtained on a Sharp Zaurus at 206MHz. The OpenSSL times were obtained using the OpenSSL development snapshot of the OpenSSL crypto library from 20-Dec-2002.

Curve Name	Curve Type	Key Size	Our Implementation		OpenSSL Implementation	
			Sign	Verify	Sign	Verify
`sect113r1`	random	113bit	2.8ms	7.5ms	12.1ms	22.9ms
`sect131r1`	random	131bit	3.8ms	11.5ms	22.1ms	43.4ms
`sect163r1`	random	163bit	5.7ms	17.9ms	28.8ms	55.9ms
`sect193r1`	random	193bit	7.6ms	26.0ms	41.6ms	80.9ms
`sect233r1`	random	233bit	10.1ms	37.3ms	56.3ms	111.1ms
`sect283r1`	random	233bit	15.4ms	59.6ms	103.3ms	205.5ms
`sect163k1`	Koblitz	163bit	5.4ms	11.7ms	28.8ms	55.9ms
`sect233k1`	Koblitz	233bit	10.1ms	21.8ms	56.3ms	111.1ms
`sect239k1`	Koblitz	233bit	10.5ms	22.7ms	57.7ms	113.9ms
`sect283k1`	Koblitz	283bit	15.5ms	33.7ms	103.3ms	205.5ms

7. Previous Work

In the past years since Miller and Koblitz proposed Elliptic Curve Cryptosystems various papers dealt with the efficient implementation of ECC on different platforms. In this section, we present some of this work together with the execution times on different platforms.

In [54] the authors describe an efficient software implementation of elliptic curves that is optimized for processors with 64-bit word size. The paper mentions a couple of useful tricks for an optimized implementation. The execution time for a scalar point multiplication on an elliptic curve over $\mathbb{F}_{2^{155}}$ is 124ms on a Sun SPARC IPC at 25MHz and 9.9ms on a DEC Alpha 3000 at 175MHz.

An extensive study of software implementation of the NIST-recommended elliptic curves over binary fields can be found in [19] by Hankerson, Hernandez and Menezes. We took most algorithms from this paper. The authors treat general elliptic curves over $\mathbb{F}_{2^m}$ as well as Koblitz curves whose special structure can be exploited for faster arithmetic. They obtained the execution times for scalar point multiplications on a Pentium II 400MHz workstation. See Table 7.0.2 for an overview of the results.

Table 7.0.2 Execution times for scalar point multiplication on a Pentium II at 400MHz using different algorithms [19].

	Underlying field		
	$\mathbb{F}_{2^{163}}$	$\mathbb{F}_{2^{233}}$	$\mathbb{F}_{2^{283}}$
Random Curves			
Binary method (affine coordinates)	9.178 ms	21.891 ms	34.845 ms
Binary method	4.716 ms	10.775 ms	16.123 ms
Binary NAF method	4.002 ms	9.303 ms	13.896 ms
Window NAF method with $w = 4$	3.440 ms	7.971 ms	11.997 ms
Montgomery method	3.240 ms	7.697 ms	11.602 ms
Fixed-base comb method with $w = 4$	1.683 ms	3.966 ms	5.919 ms
Koblitz Curves			
TNAF method	1.946 ms	4.349 ms	6.612 ms
Window TNAF method with $w = 5$	1.442 ms	2.965 ms	4.351 ms
Fixed-base window TNAF with $w = 6$	1.176 ms	2.243 ms	3.330 ms

Weimerskirch, Paar and Shantz implemented elliptic curves over binary fields on a PalmOS device with Motorola Dragonball CPU running at 16 MHz [64]. They chose the

NIST-recommended random curves and Koblitz curves over $\mathbb{F}_{2^{163}}$. The execution times are summarized in Table 7.0.3.

Table 7.0.3 Execution times for scalar point multiplication on a Dragonball CPU at 16 MHz using different algorithms [64]. The underlying field is $\mathbb{F}_{2^{163}}$.

Random Curves	
Addition-subtraction method	3310 ms
Sliding windows method ($w = 4$)	3070 ms
Width-w addition-subtraction method	2960 ms
Montgomery method	2730 ms
Fixed-base comb method with $w = 4$	1430 ms
Fixed-base comb method with $w = 8$	790 ms
Koblitz Curves	
TNAF method	1670 ms
Window TNAF method with $w = 4$	1510 ms
Window TNAF method with $w = 5$	1680 ms
Fixed-base window TNAF with $w = 6$	1080 ms
Fixed-base window TNAF with $w = 10$	870 ms

In September 2002, Sun Microsystems donated an elliptic curve implementation to the OpenSSL project [3], a programming group that works on an open-source version of the Secure Sockets Layer (SSL) encryption system. So far, the code is not optimized for particular platforms or elliptic curves. Currently the Montgomery method and the Window NAF method are used for scalar point multiplication on curves over binary fields. In [18] execution times for the ECDSA sign and verify operation on a Yopy PDA with StrongARM CPU at 200 MHz and an Sun Ultra$^{\text{TM}}$-80 server equipped with a 450 MHz UltraSPARC II processor are presented. The execution times of the ECDSA sign and verify operations are summarized in Table 7.0.4.

Table 7.0.4 Execution times of ECDSA operations on a StrongARM CPU at 200 MHz (Yopy) and on an UltraSPARC II processor at 450 MHz.

Platform	ECDSA-Operation	163-bit	193-bit
Ultra-80	Verify	6.8 ms	9.2 ms
	Sign	13.0 ms	18.1 ms
Yopy	Verify	46.5 ms	76.6 ms
	Sign	24.5 ms	39.0 ms

8. Summary and Conclusions

In this thesis, we carefully analyzed the secure charging protocol with respect to the use of different digital signature schemes. An important part of this analysis was the recommendation of key sizes to be used. Due to the micro payment character of this application, we believe that key sizes below the commonly recommended 1024-bit RSA are possible. According to our examinations, the required level of security, which is that — with a financial effort proportionate to the expected gain of about 200€ — it should not be possible to break the keys in less than 24 hours, can be achieved with 704-bit RSA keys (respectively, 131-bit ECDSA over binary curves) until the year 2006. For a sufficient protection until the year 2015, we recommend larger RSA keys with a size of at least 1024 bits (respectively, 163-bit ECDSA over binary curves). Of course, these recommendations must be treated with care, since we cannot fully anticipate the effects of progress in cryptanalysis. We therefore recommend to continuously monitor the progress in this area and to adapt the key sizes when necessary.

Having determined the required level of security, we evaluated the performance of the RSA signature scheme and the ECDSA signature scheme on the basis of the execution times on our target platform. We proposed a new measure that takes into account the special organization of state-of-the-art communication protocols such as the secure charging protocol. In these protocols, a device has to perform signature verification, signature generation or both — depending on its role in the communication process. As a result of our examination we can say that signature schemes based on elliptical curves outperform classical signature schemes such as RSA in particular for high levels of security. With respect to the required storage space and transmission bandwidth, ECC is clearly the better choice. Nevertheless, in some applications that predominantly require signature verification or need only very low levels of security, RSA may still be a good choice — even for applications in the area of wireless communications.

We implemented digital signatures based on general elliptic curves over binary finite fields as well as Koblitz curves. During this process we implemented several different point multiplication methods and determined the optimal choice for our application and target platform. Using the fixed-base comb method for multiplying the base point of the curve and the Montgomery method for multiplying arbitrary points, a signature generation on a StrongARM CPU clocked at 206 MHz takes 5.7ms and a signature verification takes 17.9ms for 163-bit ECDSA. In case of Koblitz curves, we use the window TNAF method, which enables us to verify a signature on a 163-bit Koblitz curve in 11.7ms. This is a speed up of 500% compared to the current ECDSA implementation using curves over $\mathbb{F}_{2^m}$ of the OpenSSL project.

Although our implementation supports only one particular curve during run-time, we

provided a tool that allows easy adaptation to different curves during compile time. Finally, the integration into the OpenSSL framework allows easy reuse of the implementation in future communication protocols or other applications.

We integrated our ECDSA implementation into the prototype implementation of the Secure Charging Protocol, which is currently developed by Lamparter, Paul, and Westhoff. A test run of the protocol prototype showed that ping packets between different nodes of a test network could be successfully transmitted. Unfortunately, the current prototype does not allow any statistics about the network performance and especially the performance of the digital signature operations. Hence, an interesting objective for future work would be to examine the network performance and the computational load for the network nodes caused by the signature operations in a realistic network scenario.

A. Appendix

A.1. Ecclib Manual

A.1.1. Configuration with `eccdefs.h`

The file `eccdefs.h` contains a number of `#define`'s which influence the behavior of the Ecclib code:

- `#define USE_OPENSSL_RNG`
  This `#define` should always be enabled, because it tells the compiler to use the random number generator shipped with the OpenSSL library. If the `#define` is disabled, simple calls to `rand()` are used to generate the random secrets needed for the signature generation process.
- `#define USE_OPENSSL_FOR_UNSUPPORTED_FIELD`
  This `#define` tells our implementation to call the OpenSSL default routines for signature generation and verification if it does not support the requested curve. Remember that within OpenSSL the type of curve that shall be used for the signature operations can be dynamically chosen, therefore it might be the case that our implementation has not been compiled for the requested curve.
- `#define SKIP_DGST_LEN_CHECK`
  OpenSSL usually checks if the length of the digest of the message to be signed is greater than the size of the key in bytes. This causes problems when the key size is less than 160-bits (20 bytes). By enabling this switch one can tell our implementation to skip the check. This `#define` is also enabled by default.
- `#define TEST_EXEC_TIME`
  This defines how the execution times of the self-test routines are obtained. If the switch is enabled, the self-test routines do not check if the results of the operations are correct, but only measure the execution times. The screen output is kept to a minimum in this mode. This `#define` is disabled by default.
- `#define USE_MEMxxx_FUNCTIONS`
  This `#define` determines whether our implementation uses functions like `memcpy()` or `memset()` or replaces them with loops. We did not notice any influence on the performance, so the switch is disabled by default.
- `#define AUTOSKIP_LEADING_ZEROS`
  This switch should always be disabled.

- `#define USE_SHAMIRS_TRICK`
  Enabling this switch tells the compiler to use Shamir's trick (Algorithm 6.5.1) of simultaneous point multiplication for the signature verification. As stated in Section 6.5.1, this deteriorates the overall performance, so this switch should always be disabled.

A.1.2. Code Generator `EcclibCodeGen` and Configuration `ecclib.conf`

As mentioned in Section 6.7.4, `EcclibCodeGen` generates the source files `gf2m_gen.h`, `gf2m_gen.c`, and `gf2m_inline_gen.c` during the make process. The tool can be configured with the configuration file `ecclib.conf`.

The options within `ecclib.conf` are the following:

- `CURVE=`⟨*OpenSSL Curve Identifier String*⟩
  This option determines which elliptic curve is supported by our library. The string ⟨*OpenSSL Curve Identifier String*⟩ is used within OpenSSL to denote the curve and can be found in the OpenSSL header file `openssl/obj_mac.h`. Examples for such strings are `sect113r1`, `sect113r2`, `sect131r1`, `sect131r2`, `sect163r1`, `sect163r2`, `sect163k1`, `sect193r1`, `sect193r2`, `sect233r1`, and `sect233k1`. The names are the same as the names of the curves standardized by SECG in [44]. Note, that only curves over $\mathbb{F}_{2^m}$ are supported.

- `COMBWINDOW=`$\langle w_{\text{comb}} \rangle$
  This specifies the size of the window used for the fixed-base comb method for point multiplication (Algorithm 5.3.5). Note, that our implementation demands a lower bound of $w_{\text{comb}} \geq 2$.

- `TNAFWINDOW=`$\langle w_{\text{TNAF}} \rangle$
  The size of the window used for the window TNAF method (Algorithm 5.4.5) is determined by this option. Note, that our implementation demands a lower bound of $w_{\text{TNAF}} \geq 4$.

- `SHAMIRWINDOW=`$\langle w_{\text{Shamir}} \rangle$
  Since we also implemented simultaneous point multiplication (Algorithm 6.5.1), this option determines the window size for this algorithm.

- `MAKEINLINE=1111100111101101`
  This option determines which of the 16 automatically generated functions shall be inline functions and which not. Each digit corresponds to one function. A value of 1 means that the function shall be generated as inline function and a value of 0 means that it shall be a normal function. Normally, these settings should not be changed, because they were determined with intensive tests. However, if the target platform differs from the original Zaurus PDA platform, changing this setting might improve the performance.

- `UNROLLLEVEL=2222222122212222`
  Our source code generator can be configured to support different levels of loop unrolling. Each digit corresponds to one of the 16 automatically generated functions. The values determine whether

  (0) functions like `memcpy()` shall be used,

  (1) simple `for`-loops shall be used,

  (2) most loops shall be unrolled,

  (3) all loops shall be unrolled or

  (4) assembler implementations shall be used.

  Note, that not every function supports all options. Please consider the source code of the generator for details. Normally, these settings should not be changed, because they were determined with intensive tests. However, if the target platform differs from the original Zaurus PDA platform, changing this setting might improve the performance.

- `HEADER=`⟨*Filename*⟩
  Defines the name of the file that shall contain source code for the generated header file. The default filename is `gf2m_gen.h`.

- `SOURCE=`⟨*Filename*⟩
  Defines the name of the file that shall contain source code for the generated source file. The default filename is `gf2m_gen.c`.

- `INLINE=`⟨*Filename*⟩
  Defines the name of the file that shall contain source code for the generated source file containing the inline function. The default filename is `gf2m_inline_gen.c`.

Finally, note that the `CURVE=`⟨*OpenSSL Curve Identifier String*⟩ option can also be passed via command line parameter. The code generator then uses the passed setting and updates the `ecclib.conf` file.

A.1.3. Self-Test-Routines

With `make test` the executable `ecclib.out`, which contains a number of self-test routines, is built. Which self-test routines are executed can be determined in the source file `main.cpp`. Here is a list of the tests available together with the corresponding `#define`:

- `#define TEST_BASIC_ARITHMETIC`
  This test routine checks whether the finite field and long integer arithmetic works correctly. In particular, the finite field multiplication, addition and squaring is checked. Moreover, the routine tests the elliptic curve point addition in affine and also in mixed coordinates. The test determines whether elliptic curve point doubling works properly. Finally, almost all long integer routines are checked.

- `#define TEST_POINT_MULTIPLICATION`
  This `#define` enables the elliptic curve point multiplication tests. All methods that are implemented are called with the same input parameters and checked for correctness. This routine also determines the execution times of the different point multiplication methods.

- `#define TEST_ECCLIB_SIGNATURE`
  This test checks whether signature generation and signature verification works correctly. The tests are performed on the ECDSA Sign and Verify Layer, hence, the OpenSSL interface is not involved. Consequently, the execution times obtained with this test are slightly better due to the reduced overhead.

- `#define TEST_OPENSSL_COMPATIBILITY`
  By enabling this `#define`, the compatibility of the signatures of our implementation to the OpenSSL signature routines is checked. This is done by generating a signature with our routines and verifying it with the OpenSSL routines. The test function also generates signatures with the OpenSSL routines and verifies them with our routines.

- `#define TEST_OPENSSL_SIGNATURE`
  The integration into OpenSSL is checked with this test routine. Consequently, the signing and verifying is performed by invoking the OpenSSL interface. In addition to this, the execution times are measured.

- `#define TEST_COMPARE_RSA_VS_ECDSA`
  This test compares the execution times of our ECDSA implementation with the execution times of the OpenSSL implementation of the RSA signature scheme. The bit sizes of the keys are chosen so that both schemes provide a comparable level of security.

A.2. Manual of the Certificate Tool `CertGen`

The `CertGen`-Tool can be used to generate ECDSA keys and certificates. It supports three different functions:

1. Generate a self-signed certificate for the Certificate Authority (CA) containing the CA public key and a separate file that contains the CA private key
2. Generate a CA-signed certificate for a node containing the node's public key and a separate file that contains the node's private key
3. Verify a certificate, i.e. check if the CA signature is valid and the certificate has not expired

These functions use our ECDSA implementation for the signature generation and verification and the OpenSSL routines for the certificate handling.

A.2.1. CA Certificate Generation

In order to generate a set of CA keys and a certificate, `CertGen` should be invoked with the following command:

`CertGen generate ca` ⟨*ca-arguments*⟩

where ⟨*ca-arguments*⟩ includes the following mandatory parameters:

- `ISSUER=`⟨*name of the issuer of the certificate*⟩
- `SUBJECT=`⟨*name of the subject of the certificate*⟩
- `SERIALNUMBER=`⟨*serial-number of the certificate*⟩
- `CA-CERTFILE=`⟨*name of the file that shall contain the certificate to be created*⟩
- `CA-KEYFILE=`⟨*name of the file that shall contain the private key to be created*⟩

A.2.2. Node Certificate Generation

In order to generate a set of node keys and a certificate, `CertGen` should be invoked with the following command:

`CertGen generate node` ⟨*node-arguments*⟩

where ⟨*node-arguments*⟩ includes the following mandatory parameters:

- `ISSUER=`⟨*name of the issuer of the certificate*⟩
- `SUBJECT=`⟨*name of the subject of the certificate*⟩
- `SERIALNUMBER=`⟨*serial-number of the certificate*⟩
- `CA-KEYFILE=`⟨*name of the file that contains the private key to sign the node certificate*⟩
- `CERTFILE=`⟨*name of the file that shall contain the node certificate to be created*⟩
- `KEYFILE=`⟨*name of the file that shall contain the private key to be generated*⟩

A.2.3. Certificate Verification

In order to verify a certificate, `CertGen` should be invoked with the following command:

`CertGen verify` ⟨*verify-arguments*⟩

where ⟨*verify-arguments*⟩ includes the following mandatory parameters:

- `CA-CERTFILE=`⟨*file name of the CA certificate*⟩
- `CERTFILE=`⟨*file name of the certificate to be verified*⟩

A.3. OpenSSL Command Line Parameters

We used OpenSSL to generate RSA keys and RSA certificates as well as RSA signatures. In this section, we shortly describe the necessary OpenSSL command line options.

A.3.1. Generate RSA private key file

The following command generates a file containing a private RSA key with a size of *keysize* bits:

`openssl genrsa -out` ⟨*keyfile*⟩ ⟨*keysize*⟩

To convert the file *keyfile* to the ASN.1 distinguished encoding rules (DER), we used the following command:

`openssl rsa -outform DER -out` ⟨*name of DER encoded file*⟩ `<` ⟨*keyfile*⟩

A.3.2. Generate RSA certificate

Once a private key has been generated, the following OpenSSL commands can be used to generate a DER encoded certificate file with the name *certfile*:

```
openssl req -new -key ⟨keyfile⟩ -out ⟨certificate request file⟩
openssl x509 -req -in ⟨certificate request file⟩ -signkey ⟨keyfile⟩
             -out ⟨certfile⟩ -outform DER
```

A.3.3. Generate RSA signature

To generate a RSA signature of the input file *infile* using the private key stored in the file *keyfile* and save it to the file *sigfile*, the following command can be used:

`openssl rsautl -inkey` ⟨*keyfile*⟩ `-in` ⟨*infile*⟩ `-out` ⟨*outfile*⟩ `-raw`

References

[1] Aodv. Available from http://moment.cs.ucsb.edu/AODV/aodv.html.

[2] Sharp zaurus - developer site.
http://docs.zaurus.com/linux_compiler_setup_howto.shtml.

[3] The openssl project, 1998–2000.
For further information, see http://www.openssl.org.

[4] air-lan der wlan-zugang am flughafen münchen, March 2003. More information at http://air-lan.munich-airport.de.

[5] ARM Limited. *ARM Architecture Reference Manual*, June 2000.

[6] I. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, 1999.

[7] M. Brown, D. Hankerson, J. L. Hernandez, and A. Menezes. Software implementation of the nist elliptic curves over prime fields. *Lecture Notes in Computer Science*, 2020:250–265, 2001. Available from http://citeseer.nj.nec.com/brown01software.html.

[8] S. Buchegger and J.-Y. L. Boudec. The selfish node: Increasing routing security in mobile ad hoc networks. Technical Report RR 3354, IBM Research Report, 2001.

[9] S. Buchegger and J.-Y. L. Boudec. Performance analysis of the confidant protocol: Cooperation of nodes - fairness in dynamic ad-hoc networks. In *Proceedings of IEEE/ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHOC)*, June 2002.

[10] S. Buchegger and J.-Y. Le Boudec. Nodes bearing grudges: Towards routing security, fairness, and robustness in mobile ad hoc networks. In *Proceedings of the Tenth Euromicro Workshop on Parallel, Distributed ad Network-based Processing*, pages 403–410. IEEE Computer Society, January 2002.

[11] L. Buttyán and J.-P. Hubaux. Enforcing service availability in mobile ad-hoc wans. In *Proceedings of IEEE/ACM Workshop on Mobile Ad Hoc Networking and Computing (Mobi-HOC)*, August 2000.

[12] L. Buttyán and J.-P. Hubaux. Stimulating cooperation in self-organizing mobile ad hoc networks. Technical Report DSC/2001/046, EPFL-DI-ICA, August 2001.

[13] S. Das, C. E. Perkins, and E. M. Rover. Ad hoc on demand distance vector (aodv) routing. Internet-Draft, *Mobile Ad-hoc Network (MANET) Working Group, IETF*, October 1999.

[14] H. Deng, W. Li, and D. P. Agrawal. Routing security in wireless ad hoc networks. *IEEE Communications Magazine*, 40(10):70–75, October 2002.

[15] W. Diffie and M. E. Hellman. Multiuser cryptographic techniques. In *Proceedings of AFIPS National Computer Conference*, pages 109–112, 1976.

[16] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, 1976.

[17] T. ElGamal. A public key cryptosystems and a signature scheme based on discrete logarithms. In *IEEE Transactions on Information Theory*, 31, pages 469–472, 1985.

[18] V. Gupta, S. Gupta, S. Chang, and D. Stebila. Performance analysis of elliptic curve cryptography for ssl. In *Proceedings of the ACM workshop on Wireless security*, pages 87–94. ACM Press, 2002.

[19] D. Hankerson, J. L. Hernandez, and A. Menezes. Software implementation of elliptic curve cryptography over binary fields. *Lecture Notes in Computer Science*, 1965:1–24, 2001. Available from http://citeseer.nj.nec.com/hankerson00software.html.

[20] heise online CeBIT special. Wlan-angebot von t-online, March 2003. Available from http://www.heise.de/newsticker/data/uma-07.03.03-000/.

[21] Y. Hu, A. Perrig, and D. Johnson. Ariadne: A secure on-demand routing protocol for ad hoc networks. In *8th ACM International Conference on Mobile Computing and Networking*, September 2002. Available from http://citeseer.nj.nec.com/hu02ariadne.html.

[22] J.-P. Hubaux, L. Buttyán, and S. Čapkun. The quest for security in mobile ad hoc networks. In *Proceeding of the ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHOC)*, 2001.

[23] D. Johnson, D. A. Maltz, and J. Broch. The dynamic source routing protocol for mobile ad hoc networks. Internet-Draft, *Mobile Ad-hoc Network (MANET) Working Group, IETF*, October 1999.

[24] D. Johnson, A. Menezes, and S. Vanstone. The elliptic curve digital signature algorithm (ecdsa). A Certicom Whitepaper, 2001. Available from Certicom website at http://www.certicom.com/resources/w_papers/w_papers.html.

[25] U. Jönsson, F. Alriksson, T. Larsson, P. Johansson, and G. Q. Maguire Jr. Mipmanet - mobile ip for mobile ad hoc networks. In *MobiHoc 2000*, 2000.

[26] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Cryptanalytic attacks on pseudorandom number generators. In *Fast Software Encryption - FSE '98*, LNCS 1372, pages 168–188. Springer-Verlag, 1998.

[27] N. Koblitz. Elliptic curve cryptosystems. In *Mathematics of Computation*, 48, pages 203–209, 1987.

[28] N. Koblitz. Cm-curves with good cryptographic properties. In *Advances in Cryptology - Crypto '91*, pages 279–287. Springer-Verlag, 1992.

[29] J. Kong, P. Zerfos, H. Luo, S. Lu, and L. Zhang. Providing robust and ubiquitous security support for mobile ad-hoc networks. In *International Conference on Network Protocols (ICNP)*, pages 251–260, 2001.

[30] B. Lamparter, K. Paul, and D. Westhoff. Charging support for ad hoc stub networks. In *Journal of Computer Communication, Special Issue on "Internet Pricing and Charging: Algorithms, Technology and Applications"*. Elsevier Science, 2003.

[31] B. Lamparter, M. Plaggemeier, and D. Westhoff. About the impact of co-operation approaches for ad hoc networks, extended abstract. In *The Fourth ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc'03)*, June 2003.

[32] Y. W. Law, S. Dulman, S. Etalle, and P. Havinga. Assessing security-critical energy-efficient sensor networks, June 2002.
Available from http://citeseer.nj.nec.com/539921.html.

[33] A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 14(4):255–293, 2001.

[34] C. H. Lim and P. J. Lee. More flexible exponentiation with precomputation. In *Advances in Cryptography - Crypto '94*, LNCS 839, pages 95–107, 1994.

[35] Y.-D. Lin and Y.-C. Hsu. Multihop cellular: A new architecture for wireless communications. In *INFOCOM 2000*, 2000.

[36] J. López and R. Dahab. An overview of elliptic curve cryptography. Available from http://citeseer.nj.nec.com/333066.html.

[37] J. López and R. Dahab. Improved algorithms for elliptic curve arithmetic in $gf(2^n)$. In *Selected Areas in Cryptography - SAC '98*, LNCS 1556, pages 201–212. Springer-Verlag, 1999.

[38] J. López and R. Dahab. High-speed software multiplication in $\mathbb{F}_{2^m}$. in IC Technical Report IC-00-09, Institute of Computing, University of Campinas, May 2000. Available from http://www.dcc.unicamp.br/ic-main/publications-e.html.

[39] S. Marti, T. J. Giuli, K. Lai, and M. Baker. Mitigating routing misbehavior in mobile ad hoc networks. In *6th International Conference on Mobile Computing and Networking (MOBICOM'00)*, pages 255–265, August 2000.

[40] Certicom Corp. Current public-key cryptographic systems. A Certicom Whitepaper, April 1997.

[41] Certicom Corp. Remarks on the security of the elliptic curve cryptosystem. A Certicom Whitepaper, September 1997. Available from Certicom website at http://www.certicom.com/research/wecc3.html.

[42] Certicom Corp. Certicom ecc challenge, since 1997. Available from Certicom website at http://www.certicom.com/resources/ecc_chall/challenge.html.

[43] RSA Laboratories. Factoring challenge. Available from http://www.rsasecurity.com/rsalabs/challenges/factoring/.

[44] Standards for Efficient Cryptography Group. Sec 2: Recommended elliptic curve domain parameters. Certicom Research, September 2000. Available from Standards for Efficient Cryptography Group (SECG) website at http://www.secg.org.

[45] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.
For further information, see http://www.cacr.math.uwaterloo.ca/hac.

[46] V. Miller. Uses of elliptic curves in cryptography. In *Advances in Cryptology: Proceedings of Crypto'85*, LNCS 218, pages 417–426. Springer-Verlag, 1986.

[47] P. Montgomery. Speeding up the pollard and elliptic curve methods of factorization. In *Mathematics of Computation*, 48, pages 243–264, 1987.

[48] F. Morain and J. Olivos. Speeding up the computations on an elliptic curve using addition-subtraction chains. In *Informatique théorique et Applications*, 24, pages 531–544, 1990.

[49] Smart sensor networks. Wireless Ad Hoc Networks Project at the National Institute of Standards and Technology (NIST).
Available from http://w3.antd.nist.gov/wctg/manet/index.html.

[50] P. Papadimitratos and Z. J. Haas. Secure routing for mobile ad hoc networks. In *SCS Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS 2002)*, January 2002.

[51] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[52] M. J. B. Robshaw and Y. L. Yin. Elliptic curve cryptosystems. RSA Laboratories Technical Note, June 1997.

[53] I. RSA Data Security. Des challenge iii, January 1999. More information at http://www.rsasecurity.com/rsalabs/challenges/des3/.

[54] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck. Fast key exchange with elliptic curve systems. In *Advances in Cryptology - Crypto '95*, volume LNCS 963, pages 43–56. Springer-Verlag, 1995.

[55] S. C. Shantz. From euclid's gcd to montgomery multiplication to the great divide, June 2001.

[56] Sharp Electronics (Europe) GmbH. *Documentation for the Sharp Zaurus Developer CD.*

[57] Sharp Electronics (Europe) GmbH. *SL-5500G Personal Mobile Tool - Technische Daten.*

[58] B. Shrader. A proposed definition of 'ad hoc network'. Course project report, Royal Institute of Technology (KTH), Stockholm, Sweden, May 2002.
Available from http://www.s3.kth.se/ brooke/Reports/.

[59] J. A. Solinas. Efficient arithmetic on koblitz curves. In *Designs, Codes and Cryptography*, 19, pages 195–249, 2000.

[60] F. Stajano and R. Anderson. The resurrecting duckling: Security issues for ad-hoc wireless networks. In *Security Protocols, 7th Int'l Wksp. Proc.*, LNCS, 1999.

[61] W. Stallings. *Network and Internetwork Security.* IEEE Press, 2nd edition edition, 1995.

[62] G. Stoneburner. Underlying technical models for information technology security. Recommendations of the National Institute of Standards and Technology, NIST Special Publication 800-33, December 2001.

[63] S. Vanstone. Responses to nist's proposal. *Communications of the ACM*, 35:50 – 52, July 1992.

[64] A. Weimerskirch, C. Paar, and S. C. Shantz. Elliptic curve cryptography on a palm os device. In *Information Security and Privacy, 6th Australasian Conference, ACISP 2001*, LNCS 2119, pages 502–513. Springer-Verlag, July 2001. Available from http://www.crypto.rub.de/Publikationen/texte/weika_eccpalm.pdf.

[65] M. J. Wiener. Performance comparison of public-key cryptosystems. *CryptoBytes, Technical Newsletter of RSA Laboratories*, 4(1):1, 3 – 5, Summer 1998.

[66] M. G. Zapata. Secure ad hoc on-demand distance vector (saodv) routing. Internet-Draft, draft-guerrero-manet-saodv-00.txt, October 2001.

[67] Y. Zhang and W. Lee. Intrusion detection in wireless ad-hoc networks. In *Proceedings of MOBICOM 2000*, pages 275–283, 2000.

[68] L. Zhou and Z. J. Haas. Securing ad hoc networks. *IEEE Network Magazine*, 13(6), November/December 1999.