TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Systems Architecture

Chair of Distributed and Networked Systems

Implementation and Evaluation of Clause Blind Schnorr Signatures on Constrained Systems

Max Karl Stolze

Born on: 19th July 1993 in Pößneck
Course: Distributed Systems Engineering
Matriculation number: 5151867
Matriculation year: 2023

Master Thesis

to achieve the academic degree

Master of Science (M.Sc.)

First referee
Prof. Dr. Matthias Wählisch
Second referee
Dr.-Ing. Stefan Köpsell
Supervisor
M.Sc. Mikolai Gütschow

Submitted on: 3rd December 2025

Statement of authorship

I hereby certify that I have authored this document entitled *Implementation and Evaluation of Clause Blind Schnorr Signatures on Constrained Systems* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, 3rd December 2025

Max Karl Stolze

Abstract

In this work the Clause Blind Schnorr (CBS) signature scheme is implemented in the context of constrained systems and the GNU Taler (Taler) payment system. The goal is to evaluate whether the signature scheme offers benefits over the existing RSA signature scheme that is already implemented for a Taler wallet application for embedded systems. The implementation contains a hardware-accelerated and a software-based version of the scheme. To implement the hardware-accelerated version, the used driver had to be enhanced with support for the twisted Edwards curve version of Curve25519. Different versions of scalar multiplication have been implemented and a window based approach with precalculations and combined multiplications is found to perform best. The evaluation of the blinding scheme implementation compares the performance of blind signature primitives and finds that the CBS scheme does not perform blinding and verification faster than the RSA scheme. Unblinding is executed faster with the CBS scheme. This is the case for both the hardware-accelerated and the software-based implementation. Enhancing the performance is likely possible with by further improvements regarding the hardware-accelerated approach or a different library choice for the software-based solution. The evaluation further shows that the CBS scheme needs less storage space and bandwidth for signature storage and creation.

Kurzfassung

In der vorliegenden Arbeit wird eine Implementierung des Clause Blind Schnorr (CBS) Schemas für blinde Signaturen im Kontext beschränkter Systeme und des GNU Taler (Taler) Zahlungssystems implementiert. Ziel der Arbeit ist es, zu ermitteln, ob das Signaturschema Vorteile gegenüber dem bestehenden Blindsignaturverfahren auf Basis von RSA mit sich bringt. Die Implementierung beinhaltet eine hardwaregestützte sowie eine rein softwarebasierte Version des Verfahrens. Zum Zwecke der hardwaregestützten Implementierung wurde der Treiber für die Kryptographie-Peripherie um die twisted Edwards Variante von Curve25519 ergänzt. Es wurden verschiedene Lösungen für Skalarmultiplikation implementiert und ein window basierter Ansatz mit Vorberechnungen und kombinierten Multiplikationen als beste Lösung identifiziert. Die Evaluation vergleicht die Leistung der beiden Systeme hinsichtlich der Primitiven für Blindsignaturverfahren und stellt fest, dass Verblindung und Verifizierung mit dem implementierten CBS Schema langsamer sind als mit dem bestehenden RSA Schema. Entblindung ist hingegen schneller mit dem CBS Schema. Die Evaluation zeigt weiterhin, dass mit dem CBS Schema weniger Speicherplatz und Bandbreite für die Speicherung und Erstellung der Signaturen benötigt wird.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Goal . . . 2
- 1.3 Strategy . . . 2

2 Background **3**
- 2.1 Mathematical Foundations . . . 3
  - 2.1.1 Groups . . . 3
  - 2.1.2 Finite Fields . . . 4
  - 2.1.3 Elliptic Curves . . . 5
- 2.2 Cryptographic Basics . . . 6
  - 2.2.1 Cryptography . . . 6
  - 2.2.2 Cryptographic Hash functions . . . 6
  - 2.2.3 Key Derivation Functions . . . 7
  - 2.2.4 Public-Key Cryptography . . . 7
  - 2.2.5 Signatures . . . 8
  - 2.2.6 Blind Signatures . . . 9
- 2.3 Blind Signature Schemes in Relevant Crypto Systems . . . 11
  - 2.3.1 RSA . . . 11
  - 2.3.2 Schnorr Signatures . . . 13
- 2.4 Constrained Systems . . . 20
  - 2.4.1 Overview . . . 21
  - 2.4.2 RIOT OS . . . 21
  - 2.4.3 nrf52840DK . . . 22
- 2.5 GNU Taler . . . 23
  - 2.5.1 Overview . . . 23
  - 2.5.2 Withdrawal . . . 24

3 Related Work **28**
- 3.1 Recent Developments in Blind Signatures . . . 28
- 3.2 Cryptography on Embedded Systems . . . 29

4 Design and Implementation **31**
4.1 Overview . . . 31
4.2 Design . . . 32
4.2.1 Integration in Wallet Application . . . 32
4.2.2 Software-Based Approach . . . 34
4.2.3 Hardware-Accelerated Approach . . . 35
4.2.4 Testing . . . 35
4.3 Implementation . . . 35
4.3.1 Software-Based Approach . . . 36
4.3.2 Hardware-Accelerated Approach . . . 36
4.3.3 Integration in Wallet Application . . . 43

5 Evaluation **45**
5.1 Setup . . . 45
5.2 Scalar Multiplication Approaches . . . 46
5.3 Comparing RSA and CBS . . . 49
5.3.1 Time Measurements . . . 51
5.3.2 Memory Footprint . . . 53
5.3.3 Security Level . . . 56

6 Conclusion **58**
6.1 Summary . . . 58
6.2 Future Work . . . 59

List of Abbreviations **61**

List of Figures **61**

List of Tables **62**

Bibliography **64**

1 Introduction

This introductory chapter will first present the motivation for this thesis. This is followed by the overall goal and strategy to achieve it.

1.1 Motivation

Paying electronically for goods and services is convenient as it can happen instantly and removes the necessity of carrying and handling cash money. Whether the payment happens in an online shop over the internet or via card payment in a physical shop, the bank and involved payment providers learn about the time and amount of the payment as well as about the transaction partners, which is in stark contrast to cash. Evidence that this situation did not appear unanticipated can be found in a paper of David Chaum from 1983. In [22], he reflects on developments in the payment landscape and explains that payment information can reveal a lot about commonly visited locations, lifestyle and associations of an individual. To counter that and allow a more privacy-preserving form of digital payments, he introduces an anonymous payment system based on a new cryptographic concept that he calls blind signatures. In principle, blind signatures allow a requester to ask a signer for a digital signature on a message that the signer can not see and thus needs to sign it blindly.

This concept is still of interest today and used, for example, by GNU Taler (Taler). Taler is a payment system that allows anonymous payments for customers as well as transparent and taxable income for merchants. It offers two different options for the issuance of blind signatures: RSA blind signatures [22] and Clause Blind Schnorr (CBS) signatures [29, 38]. The former are based on the RSA crypto system [75], the latter on the Schnorr signature system [77] on elliptic curves.

Cryptography on elliptic curves is of interest for the Internet as a whole as it allows smaller key and signature sizes. This becomes even more interesting for devices that are constrained in their computational power, memory, storage or energy capabilities. In the context of the Taler payment system, such devices could serve as hardware for wallets that are used to handle funds and authorize payments. This thesis will evaluate the use of the Clause Blind Schnorr signature scheme on such constrained devices in the context of a wallet for the GNU Taler payment system.

1.2 Goal

A version of the GNU Taler wallet for constrained devices is being developed at the Chair of Distributed and Networked Systems of the Dresden University of Technology. Taler supports two blind signature systems for the purpose of cipher agility: RSA as well as the Clause Blind Schnorr signature system. More precisely, the RSA blind signatures scheme builds on RSA-FDH [12] where FDH stands for full domain hash. The wallet software in scope does not support the CBS signature scheme but only RSA-FDH. The CBS implementation for Taler makes use of elliptic curves and promises smaller key and signature sizes. Hence, less data needs to be transmitted and the keys and signatures occupy less storage space on the participating devices, which directly translates to a higher possible amount of coins on a device in the case of Taler. The work that initially added CBS to Taler [29] found that it performs better than RSA on their test systems, which were desktop and notebook systems. It shall be investigated whether their findings translate to the constrained systems that are in scope here and how CBS performs in comparison to the RSA-FDH implementation with fixed 2048-bit key size that is available in this wallet.

1.3 Strategy

A literature review regarding the security of the CBS scheme and the application of cryptography on constrained devices shall be performed. In order to do the evaluation of the CBS approach an example implementation shall be developed for the wallet software. The implementation shall be able to make use of a hardware accelerator for cryptographic operations if available and otherwise fall back to a cryptographic software library. This shall resemble the capabilities of the current RSA-FDH implementation. To implement the hardware-accelerated version, the driver for the crypto-peripheral needs to be enhanced to support the twisted Edwards curve version of Curve25519. For the software-based solution, a suitable cryptographic software library must be chosen.

2 Background

This chapter shall cover the basics needed to understand the remainder of the thesis. At first mathematical foundations and the basics of public key cryptography as well as digital signatures are explained. Then the topic of blind signatures in general and the RSA as well as Clause Blind Schnorr scheme are explained in detail. Next, the specific ecosystem in which the schemes shall be used is introduced. This begins with constrained devices as the hardware in focus and continues with the payment system GNU Taler.

2.1 Mathematical Foundations

This section introduces the mathematical basics used throughout the thesis. The reader is expected to have basic understanding of modular arithmetic. The sources for this section, Sections 4.3, 8.2 and 9.1 of [63] and [17], give more details and in-depth explanations. Here, just the most important parts and representations are repeated.

2.1.1 Groups

A *group* is a tuple $(G, \circ)$ of a set G and a binary operation $\circ : G \times G \to G$. In order to be a group, it must fulfill the following requirements or axioms:

- Closure: $\forall x, y \in G : (x \circ y) \in G$
- Associativity: $\forall x, y, z \in G : (x \circ y) \circ z = x \circ (y \circ z)$
- Existence of exactly one identity element:
  It exists $e \in G$ that fulfills $\forall x \in G : x \circ e = e \circ x = x$
- Existence of an inverse element:
  $\forall x \in G$ exists an $i \in G : x \circ i = i \circ x = e$

The group is called *abelian* if it additionally fulfills the requirement

- Commutativity: $\forall x, y \in G : x \circ y = y \circ x$

Usually, the operations for groups are called addition, ($+$) or multiplication ($\cdot$). The related groups are then called *additive* or *multiplicative* groups. Table 2.1 summarizes the common notations for those groups.

	Additive	Multiplicative
Group Operation	+	-
Inverse of $x \in G$	$-x$	x^{-1}
Neutral Element	0	1
apply $\circ$ m times to $x \in G$	$mx = x + x + ... + x$	$x^m = x \cdot x \cdot ... \cdot x$

Table 2.1: Common notations for additive and multiplicative groups

The number of elements in the set G is called the *order* or *cardinality* of the group and in case G has a finite number of elements, the groups is a *finite* group. It is denoted as $|G|$. An example for a finite abelian group, commonly denoted as $\mathbb{Z}_p$, would be the set of integers $\{0, 1, ..., p-1\}$, with $p > 1$ and cardinality $|\mathbb{Z}_p| = p$ in combination with the group operation $+$ defined as addition modulo p, i.e. for $x, y \in \mathbb{Z}_p : x+y = x+y \mod p$ and neutral element 0. Similarly, if $p > 1$ is a prime number, then $\mathbb{Z}_p^*$ denotes the set of integers $\{1, ...p-1\}$ with $|\mathbb{Z}_p^*| = p - 1$ in combination with the group operation $\cdot$ defined as $x, y \in \mathbb{Z}_p^* : x \cdot y = x \cdot y \mod p$ and neutral element 1, which forms a finite abelian group.

The *order of a single element* x of a group (ord(x)) is the smallest positive integer k such that $k \cdot x = 0$, when 0 is the neutral element of the group. If a group G contains an element x with ord(x)=$|G|$, then G is said to be *cyclic* and x is called a *generator*. Each element of G can be represented as multiple of x in an additive group or power of x in a multiplicative group. The introduced group $\mathbb{Z}_p^*$ forms a finite cyclic group for every prime p. Further, if G is a finite cyclic group, then each element $x \in G$ in operation with itself $|G|$ times, yields the neutral element and also the order of x divides the order of G. Lastly, if $|G|$ is prime, all elements other than the neutral element are generators.

Considering a group $(G, \circ)$ and a subset $H \subset G$, that also fulfills all group axioms under operation $\circ$, then $(H, \circ)$ is called a *subgroup* of $(G, \circ)$. If G is a cyclic group, then each $x \in G$ generates a cyclic subgroup H with $|H| = ord(x)$. Further, the order of the subgroup H divides the order of the group G.

2.1.2 Finite Fields

A set F together with two binary operations, usually $+$ and $\cdot$, is called a *field* if the following properties are fulfilled:

- $(F, +)$ is an abelian group with neutral element 0
- $(F \backslash \{0\}, \cdot)$ is an abelian group with neutral element 1
- Distributivity holds: $\forall x, y, z \in F : z \cdot (x + y) = zx + zy$

Note, $F\backslash\{0\}$ denotes the set F without 0.

The field is called finite if the set F is finite. The amount of elements in a finite field is again called the order of the field. Every finite field has a prime power order, which means that the order can be expressed as follows: let q be the order of a finite field F, then $q = p^m$, with p being a prime number an $m \in \mathbb{N}, m \geq 1$. A field is called a *prime field* if $m = 1$ and an *extension field* if $m \geq 2$. We shall focus on prime fields. F_p denotes a finite field of prime order p, consisting of the set of numbers $\{0, 1, ..., p-1\}$ and the operations $+$ and $\cdot$ performed modulo p. Another common name for finite fields is *Galois field* and the corresponding notation is $GF(p)$.

2.1.3 Elliptic Curves

An *elliptic curve* can be defined by Equation (2.1), defined over a prime field F_p with $p > 3$, together with the two parameters $a, b \in F_p$ where $4 \cdot a^3 + 27 \cdot b^2 \neq 0 \mod p$.

$$E : y^2 = x^3 + ax + b \mod p \tag{2.1}$$

This form is called Weierstrass representation of an elliptic curve.

A point on such a curve is a pair (x, y) with $x, y \in \mathbb{Z}_p$, where $\mathbb{Z}_p$ is the additive group of F_p, that satisfies Equation (2.1). There is one additional point on the curve, the so-called point at infinity, denoted as $\mathcal{O}$ or just 0. The set of all points on E over a certain finite field F_p is denoted as $E(F_p)$.

Under an appropriate operation, the points on the curve form a group. The operations on the curve are defined in additive notation, so the operation is $+$. A point consists of two coordinates $P = (x, y), x, y \in \mathbb{Z}_p$. The point at infinity serves as neutral element in the group operation and thus $P + \mathcal{O} = \mathcal{O} + P = P$. Addition of two points, $P = (x_1, y_1), Q = (x_2, y_2)$ yields a third point $R = (x_3, y_3)$, i.e. $R = P + Q$ or $(x_3, y_3) = (x_1, y_1) + (x_2, y_2)$. Doubling of a point, i.e. $R = P + Q$ with $P = Q$, yields a second point as well and can be denoted as $R = 2P$. Adding a point to itself more often is called scalar multiplication and denoted as nP for scalar n, i.e. $P + P + P + P + P = 5P$.

Weierstrass curves need case differentiations for the addition formulas. Those are not detailed here but can be found in the referenced literature. Instead, another curve type is presented for which addition is possible without differentiations in a single formula. This different form to represent elliptic curves is the *twisted Edwards* representation [17]. Such curves are also defined over a finite field F_p with $p > 3$ and described with Equation (2.2), with nonzero $d, a \in \mathbb{Z}_p$.

$$E_{Ed} : ax^2 + y^2 = 1 + dx^2y^2 \tag{2.2}$$

The group operation for two points $P = (x_1, y_1), Q = (x_2, y_2)$ is again $+$ and defined as seen in Equation (2.3). It can be used for doubling as well as addition.

$$P + Q = \left(\frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2}\right) \tag{2.3}$$

The neutral element $\mathcal{O}$ is the point $(0, 1)$ and the inverse of any point (x_1, y_1) is $(-x_1, y_1)$.

Arithmetic with that formula can be further enhanced by use of projective coordinates (x, y, z) [17] or extended coordinates (x, y, z, t) [48] instead of the presented affine (x, y) coordinates. The reader is referred to the original sources for the exact formulas.

2.2 Cryptographic Basics

This section gives an overview over cryptographic building blocks used throughout the thesis and a general introduction to cryptography. This includes an overview over public key cryptography, digital signatures and blind signatures. Lastly, the relevant blind signature systems are introduced.

2.2.1 Cryptography

The authors of [63] introduce cryptography as the science of securing communication against adversaries. While commonly associated with the encryption of plaintext messages to some sort of cipher code, modern cryptography aims to fulfill a diverse set security goals.

Such security goals are effectively services that a security system shall offer. The most common ones are sometimes referred to as CIA:

- **C**onfidentiality: information is kept secret from unauthorized parties
- **I**ntegrity: messages have not been modified while being sent
- **A**uthenticity: the sender of the message is who he claims to be

More security goals exist and the ones that are most important for this thesis are:

- Non-repudiation: message senders can not refute message creation
- Anonymity: protection against the discovery of one's identity

Cryptographic algorithms can be split in two major groups: symmetric and asymmetric. Symmetric is the easy-to-imagine option of two people sharing a common secret and thus being able to communicate securely over an insecure medium. Here we deal with the other option, asymmetric cryptography, also known as public-key cryptography which will be explained in more detail.

2.2.2 Cryptographic Hash functions

The source used to describe this building block is section 14.1 of [80]. A hash function H takes an input of arbitrary length and generates a fixed length output. As such, it maps from $\{0, 1\}^*$ to $\{0, 1\}^t$ where t is the bit length of the output. This output is usually called hash value or digest. A hash function is called a cryptographic hash function if it fulfills three different properties:

- Preimage resistance
- Second preimage resistance
- Collision resistance

Preimage resistance means that, given H and a digest y, it is hard to find x, such that $H(x) = y$, i.e. it should be hard to find a message that yields a given hash value. Second preimage resistance means that, given H and x_1, it should be hard to find x_2, with $x_2 \neq x_1$, such that $H(x_1) = H(x_2)$, i.e. it should be hard to find a second messages that yield the same hash value as a given message. Collision resistance means that, given just H, it should be hard to find two messages x_1, x_2 with $x_1 \neq x_2$, such that $H(x_1) = H(x_2)$, i.e. it should be hard to find any two messages with the same hash value given nothing else than the hash function. The last property is the hardest when looking at the domain (possible input set) and codomain (set containing the possible outputs) of a hash function as stated above. The set of arbitrary inputs $\{0,1\}^*$ is larger than the set of possible outputs of fixed length $\{0,1\}^t$. Hence, collisions are unavoidable, but it should at least be hard, meaning computationally infeasible, to find any. The source for this subsection includes a longer discussion on infeasibility of finding collisions.

2.2.3 Key Derivation Functions

The source for the following description is RFC 5869 [55]. The basic idea of a key derivation function is to derive strong cryptographic key material from some initial input material, the input keying material (IKM). Deriving key material from a given input is useful if a deterministic output is desired and purely random generated values are not. RFC 5869 specifies a KDF based on HMAC [54] intended as a basic building block for cryptographic applications, called HKDF. It follows an extract-and-expand mechanism, meaning that first the input material is used to extract a cryptographically strong pseudorandom key and second this key is expanded. Expansion creates the actual output, multiple additional pseudorandom keys of which a desired amount of bytes is outputted as keying material. The first phase is necessary when the input material is not cryptographically strong. If it is, the extract phase can be skipped and the input material can be expanded directly. An instantiation of HKDF needs a hash function for the used HMAC functionality. Next to the IKM a random salt and an info string are optional parameters. A random salt makes the output independent of the IKM, if that is desired. Using an info string makes it possible to bind the key material to a context. That is, one can use the same IKM with different info elements to obtain different keying material for different sections or modules of an application. Lastly, an output length for the produced keying material must be specified. The source for this section specifies the exact functionality of the extract and expand phases.

2.2.4 Public-Key Cryptography

This subsection is based on chapter 6 of [63]. As mentioned above, symmetric cryptography has the requirement of a shared key between communication partners. Sharing this secret

is the first hurdle in secure communication. Some prominent figures who dealt with this during the advent of public-key cryptography were Merkle, Diffie and Hellman. In 1976 Diffie and Hellman published a solution [33] to obtain such a shared secret electronically. They developed the concept of public-key or asymmetric cryptography to achieve that. It leaves behind the assumption of a pre-shared secret between two communication partners that was transmitted securely. Rather, it envisions the existence of a pair of keys for each party. One key is private and shall be kept secret, the other is public and can be shared with everyone even over an insecure channel. This setup is now asymmetric in two senses. First, both parties now hold different keys and not copies of the same key. Second, the capabilities of the keys have changed. In the sense of encryption and decryption the public-key can only be used to encrypt messages. To decrypt the message again, the private key is needed. In order to build such a system, the common foundation is a one-way function. That is a function that is easy to compute but hard to reverse, if only the output is known. Hardness usually means computational infeasibility, i.e. that there is no fast solution known to such problems that is more efficient than applying brute force techniques. For this thesis, the relevant problems are the factoring problem and the discrete logarithm problem, more specifically the elliptic curve discrete logarithm problem. Both of them will be introduced in section 2.3.

2.2.5 Signatures

The purpose of public-key cryptography which is most important in this thesis are digital signatures and specifically blind signatures which will be explained in the next subsection. Source of this subsection is chapter 10 of [63]. In a general sense digital signatures can be seen as the digital counterpart to physical signatures on physical documents. When signing a document with pen and paper, the signature signals that the signer is agreeing with the content of the document. At the same time it is considered as unique and thus identifies the signer. Realistically, a signature can be forged and a document can be altered once the signature is on it. However, those risks are accepted in everyday live or are mitigated by justice or moral. When moving that idea to the digital realm, the very basic idea is to use the private key of a person to sign a message and its public key to verify the authenticity of the signature. This motivates two foundational functions in a digital signature scheme: signing and verifying. A signer applies the sign function to a message while using his private key to obtain a signature. This is sent to a recipient along with the message. A verifier can verify the signature using the public key of the signer. The verification yields either true or false as result, meaning that the signature is valid for message and key or that it is not valid. A key generation function is needed to create the public and private keys used during sign and verify. Hence, a digital signature system usually includes operations for key generation, signing and verification. This conceptional process is displayed in figure Figure 2.1.

Looking back at the concept of security goals, briefly introduced in Section 2.2.1, this scheme aims to achieve the security goals of authenticity, non-repudiation and validity. Authenticity is given because it is assumed that the public key used for verification matches exactly one private key, that of the signer. The step of linking a private key to a specific human being or entity is something that has to be done prior and is usually achieved by the means of certificates, which will not be further explained here. Non-repudiation, the

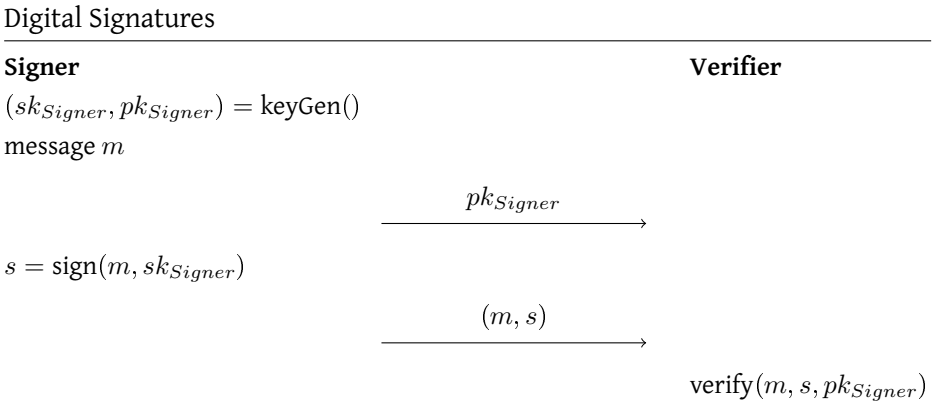

Figure 2.1: Concept of digital signatures. The signer generates a public (pk) and a private or secret key (sk) and holds a message he wants to send. First the public key is shared with the verifier. Then, the message is signed using the private key. The message is sent to the verifier together with the generated signature. The receiver verifies the signature using the public key received prior.

inability of the sender to refute that he has sent the message, is achieved by the same means. Only he should have access to the private key, so only he could have generated the signature. Validity is obtained by verifying the message together with its signature. In case the message would have been altered during transit to the recipient, i.e. its validity would have been violated, the verification process would no longer yield true for the combination of message, signature and public key. It should be noted, however, that the verification will fail if any of the three elements has been modified on its own. The signature systems of RSA and Schnorr will be explained in the sections 2.3.1 and 2.3.2.

2.2.6 Blind Signatures

The concept of blind signatures was first introduced by Chaum in [22], with the analogy of envelopes which are lined with carbon paper. If a document is placed in such an envelope and handed to someone, that person could sign the outside of the envelope and, given the carbon paper lining, the signature would be copied onto the document inside the envelope. The interesting feature of this concept is, that the signer never saw the document he signed via the carbon paper. The original paper gives a detailed explanation of how an anonymous voting system could be established with such envelopes and the reader is encouraged to get further details from there. Chaum explains further how this concept of blindly signing documents, when implemented digitally, can be used to create an untraceable payment system that keeps the payer anonymous. A user of the system would create a digital document and a secret value. He would alter that document using the secret and send it to a bank in order to get a signature of the bank. Such a signed document would be worth an agreed amount, for example 1€. The bank would debit the user's account the specified amount when issuing

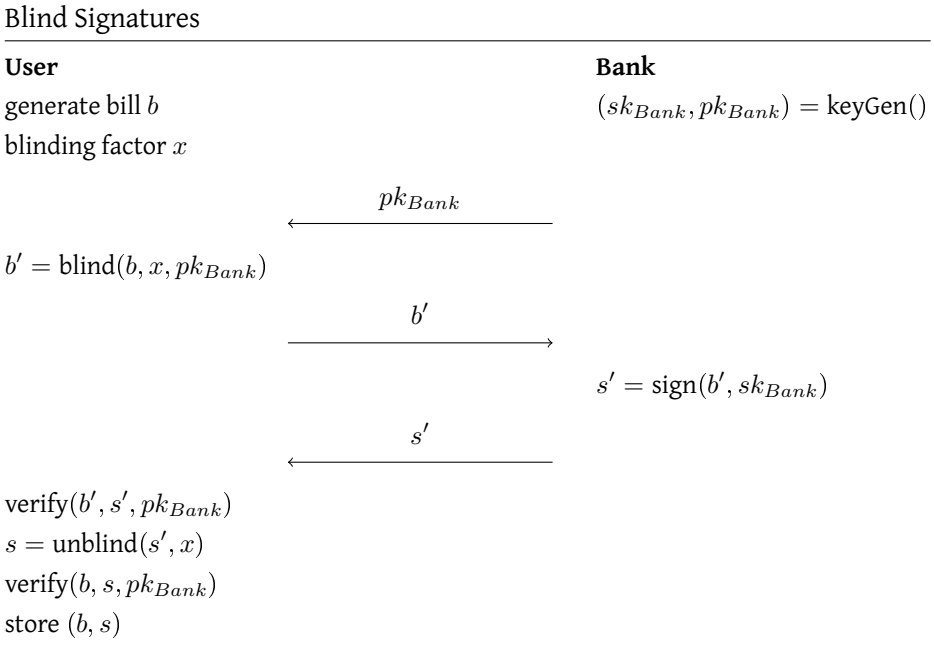

Figure 2.2: Basic concept of blind digital signatures in an anonymous payment scenario.

a signature. Using the secret the user holds, he could now change the signature in such a fashion that it is valid for the original unchanged document. This way, he would have created a document that the bank has never seen and gained a valid signature of the bank for that document. Later, he could use that document to pay a vendor. The vendor could then exchange the signed document for regular currency at the bank. As the bank has never seen that document before, it could only verify that it carries a valid signature but would not be able to link it to the payer. Figure 2.2 gives an overview over the explained scheme. This concept is also explained in greater detail in the original paper. The short description given here shall be sufficient to point out the main features and requirements of blind digital signatures.
The main properties of the blind signature scheme are:

- Blindness: the signer shall not be able to make a connection between the original and the altered document
- One-More Unforgeability: the user shall not be able to produce more valid signatures than blinded ones obtained from the signer

Just as with general digital signatures, operations for key generation, signing and verification are needed. For blind signatures, additional functions for blinding and unblinding are necessary. The blinding function uses a blinding factor to alter the message that the user wants to have a signature for and generates the blinded message. The unblinding function

uses the signature on the blinded message that is obtained from the signer and generates a valid signature for the original message.

2.3 Blind Signature Schemes in Relevant Crypto Systems

The two blind signature schemes that are relevant in this thesis are RSA-FDH and Clause Blind Schnorr (CBS). Hence, the following will be an explanation of the RSA-FDH signature and blind signature scheme as well as the Schnorr signature scheme and the Clause Blind Schnorr signature scheme. Mathematical details are given as needed and for more details the reader is referred to the sources mentioned in the single sections.

2.3.1 RSA

The following brief description is based on Chapter 7 of [63]. RSA, getting its name from its inventors Rivest, Shamir and Adleman, was proposed in 1978 [75] and became the most used public-key crypto system since. It can be used for the encryption of data as well as for digital signatures and key exchange. Fundamentally, it is based on the mathematical problem of integer factorization.

Definition 1 (Integer Factorization Problem)
Given n, find prime numbers p, q such that $p \cdot q = n$

The multiplication of two prime numbers is easy. Factoring the resulting number, however, is a very hard problem. More information on the hardness follows below.

A participant of the system creates a private and a public key, based on the following steps.

- pick large primes p, q
- compute $n = p \cdot q$
- compute $\phi(n) = (p-1)(q-1)$
- select the public exponent $e \in \{0, ...\phi(n) - 1\}$, with e coprime to $\phi(n)$
- compute the private key $d \cdot e \equiv 1 \bmod n$

e can then be published and used by others to **e**ncrypt data, while d is kept private to **d**ecrypt data. To be precise, the public key consists of the exponent e as well as the modulus n, so the tuple (e, n). For the private key it would technically be sufficient to store it as (d, n), but usually it is stored as a larger set of data. The factorization of n is what makes RSA secure. Factoring n is only hard if it is sufficiently large. This attribute changes over time as with growing computational power the time needed to factor large numbers decreases. According to the German Federal Office for Information Security (BSI) [49], n is advised to be at least 3000 bit long, at the time of writing. That means that p and q should be at least about 1500 bit each.

RSA-FDH Signature Scheme

Signer		**Verifier**
knows :		knows :
FDH, $(sk_{Signer}, pk_{Signer}) = \text{keyGen}()$		FDH, $pk_{Signer} = (e, n)$
message m		
$h = \text{FDH}(m)$		
$s = h^d \mod n$		
	$\xrightarrow{(s,m)}$	
		$s' = s^e \mod n$
		$h' = \text{FDH}(m)$
		if $h' == s'$
		accept signature as valid
		else signature is invalid

Figure 2.3: RSA Full Domain Hash Signature scheme. Signer sends a message together with the related signature to the verifier. It is assumed that both parties know and apply the same FDH hash function.

RSA Signatures

Information about RSA signatures is obtained from chapter 15.2 of [80]. Multiple schemes exist, here RSA-FDH is explained. This scheme uses a cryptographic hash function to hash the message before signing it. FDH is the abbreviation of full domain hash and means that the employed hash function must have $\{0, ..., n-1\}$ as codomain, where n is an RSA modulus. The source for this subsection delivers more details on the FDH concept, here it is simply assumed that the employed hash function has the requested codomain. Figure 2.3 gives an overview over the signature scheme.

The sender first hashes the message he wants to send by use of a FDH hash function that must be known to him as well as to the sender. This message hash is then signed using the private key of the sender and sent to the recipient together with the message. To verify the received signature, the recipient first applies the public key of the sender to the signature to obtain the hash created by the sender. Second, he hashes the received message himself using the same hash function as the sender. The verification is successful if the received hash and the self generated hash are equal. If the values differ, the verification fails. Using this scheme, the three security goals mentioned in Section 2.2.5 can be achieved. Integrity is given through the matching hash values generated by sender and receiver. The hashes can only match if the private key used to sign the message is the matching counterpart to the public key known to the verifier. This ensures that the owner of the private key has created the signature.

It should be mentioned, that the use of private and public keys in this scheme is in essence the same as in the RSA encryption scheme. This is explained in more detail in the source used for the section. Equation (2.4) shows how the verification works, assuming the right private and public keys. Key part of the functionality is $(h^d)^e \mod n = h^{de} \mod n = h^1 \mod n$. The last step of the key generation outlined above guarantees that. A full proof can be found in Section 7.3 of [63].

$$\begin{aligned} h' &= s' \\ \text{FDH}(m) &= s^e \mod n \\ \text{FDH}(m) &= (h^d)^e \mod n \\ \text{FDH}(m) &= h^n \mod n \\ \text{FDH}(m) &= h \\ \text{FDH}(m) &= \text{FDH}(m) \end{aligned} \tag{2.4}$$

Blind RSA Signatures

The introduced RSA-FDH signature scheme can be turned into a blind signature scheme as introduced in Section 2.2.6. Sections 5.3 and 23.12 of [76] are used as source for the explanation.

Figure 2.4 shows the protocol between a user and a signer. The message to sign is first blinded by use of a blinding factor and the signers public key. It is then sent to the signer who signs it with his private key just as in the regular RSA-FDH scheme and returns the signature to the user. This signature is unblinded under use of the blinding secret created prior. Verification is done in the same way as in the RSA-FDH scheme. The validity of the unblinded signature can be seen when one follows the calculations done during the protocol as shown in Equation (2.5).

$$\begin{aligned} s = s'/x \mod n &= s' \cdot x^{-1} \mod n \\ &= (h'^d) \cdot x^{-1} \mod n \\ &= ((h \cdot x^e)^d) \cdot x^{-1} \mod n \\ &= h^d \cdot x^{ed} \cdot x^{-1} \mod n \\ &= h^d \cdot x^1 \cdot x^{-1} \mod n \\ &= h^d \mod n \end{aligned} \tag{2.5}$$

This protocol gives the user a valid signature of the signer on the hash h of message m of his choice, without the signer knowning m.

2.3.2 Schnorr Signatures

The focus now changes to the other signature system of interest in this thesis, Schnorr signatures. The section will again start with an overview over the used one-way function that

RSA-FDH Blind Signature Scheme

User		**Signer**
knows:		knows:
FDH, message m,		FDH, $(sk_{Signer}, pk_{Signer}) = \text{KeyGen}()$
	$pk_{Signer} = (e, n)$ $\longleftarrow$	
blind:		
generate blinding factor x		
$h = \text{FDH}(m)$		
$h' = h \cdot x^e \mod n$		
	h' $\longrightarrow$	
		sign:
		$s' = h'^d \mod n$
	s' $\longleftarrow$	
unblind:		
$s = s'/x \mod n$		
verify:		
if $h == s^e$		
accept signature as valid		
else signature is invalid		

Figure 2.4: RSA-FDH blind signature scheme between a user and a signer.

makes this signature system secure, together with the related key generation operation. Next, the general signature scheme as well as the Clause Blind Schnorr (CBS) signature scheme will be introduced.

Schnorr signatures are named after their inventor Claus Peter Schnorr, who published the scheme in 1991 [77]. He explains that the security of the scheme relies on the hardness of the discrete logarithm problem. This problem is displayed in Definition 1.

Definition 2 (Discrete Logarithm Problem)
Given p, y, α with p prime, find x such that: y = $\alpha^x \mod p$

Schnorr mentions further that the scheme can also be implemented with elliptic curves, which is the setting used in this thesis. The problem ensuring security of the scheme then changes to the elliptic curve discrete logarithm problem (ECDLP).

Definition 3 (Elliptic Curve Discrete Logarithm Problem)
Given $E(F_p), Q, P$, with p prime, find x such that: $Q = x \cdot P$

In order to use the scheme, a set of parameters must be publicly known to all participants.

- A prime field F_p
- An elliptic curve E defined over the prime field F_p
- A generator point $G \in E$ that creates a cyclic subgroup of order q
- A hash function H

The key generation algorithm is then:

- Pick a scalar value $k \in F_p, k < q$
- Calculate $K = kG$

The scalar k serves as secret key and the point K serves as public key. Note, that the notation for elliptic curve arithmetic is used. Capital letters (X, R) signify points on the curve while small letters (x, r) mean scalars. The group operation is addition and repeated addition of a point is depicted like multiplication (e.g. $P + P + P = 3P$).

The signature generation scheme is displayed in Figure 2.5. The signer picks a random scalar r and uses it to calculate a point R. This point is concatenated with the message to send (denoted as $m||R$) and hashed. Next, the scalar s is calculated under use of scalar r, the hash value c and the signer's private key. The signature consists of the pair (s, R) that is sent to the verifier together with the message m. The verifier hashes message and R to calculate his own hash value. If message and signature are correct, the verification will succeed. Equation (2.6) shows why the verification works when $c == c'$.

$$\begin{aligned} sG &= R + c' \cdot kG \\ sG &= rG + c' \cdot kG \\ (r + c \cdot k)G &= rG + c' \cdot kG \\ (r + c \cdot k)G &= (r + c' \cdot k)G \end{aligned} \tag{2.6}$$

Schnorr Signature Scheme on Elliptic Curves

Signer		**Verifier**
knows:		knows:
$k_{Signer}, K_{Signer} = \text{keyGen}()$		K_{Signer}
message m		
sign:		
pick random $r \in \mathbb{Z}_q$		
$R = rG$		
$c = \text{H}(m\|\|R)$		
$s = r + c \cdot sk_{Signer} \mod q$		
	$\xrightarrow{m, (s, R)}$	
		verify:
		$c' = \text{H}(m\|\|R)$
		if $sG == R + c'K_{Signer}$
		accept signature as valid
		else signature is not valid

Figure 2.5: The Schnorr signature scheme on elliptic curves. Generator G and order q are known to both parties, just as hash function H.

Blind Schnorr Signatures

The introduced signature scheme can be modified to become a blind signature scheme. This blind signature scheme was first published by Chaum and Pedersen in [23], this section, however, uses [38] as its basis. The scheme is depicted in Figure 2.6.

The user chooses two blinding parameters α, β and receives a nonce point R. Note, that there needs to be an initial message from the user to the signer in order to initiate the protocol. This is left out in the figure and the first action is the signer sending R to the user. The blinding factors are used in the calculation of R' and c. The latter is sent to the signer, which applies the same signature approach as in Figure 2.5. The signature is returned and unblinded by the user. The verification is again the same as for regular Schnorr signatures (see Equation (2.6)). It should be noted, however, that also the blinded signature can be verified. Doing so would avoid the effort to unblind the signature and only then recognize it as invalid. That the signer can not correlate the unblinded signature with the blind signature is proven in [23].

Clause Blind Schnorr Signatures

The Clause Blind Schnorr (CBS) signature scheme was introduced by Fuchsbauer et al. in 2019/2021 in [38]. Classical blind Schnorr signatures rely on an additional hard problem, the *ROS problem*. This problem was introduced by Schnorr in 2001 [78] and a sub-exponential algorithm to solve it was published just slightly later by Wagner [93]. A new attack on the ROS problem was published by Benhamouda et al. [13] in 2020, that builds on the results of Wagner and delivers even faster solutions to the ROS problem. The scheme of Fuchsbauer et al. is based on the so called *modified* ROS problem or short *mROS*, which is not affected by the improved attack of Benhamouda et al. Their modification to the blind Schnorr signature protocol is small but more importantly, it does not change the signatures and is therefore compatible with systems that already use Schnorr signatures.

The signature scheme gets the "clause" part of its name from the idea that multiple runs are started (R_1, R_2) and just one is finished $(R_1 \vee R_2)$ - i.e. one clause finishes. The signer decides which run gets finished on a random basis. The protocol is shown in figure 2.7 with $t = 2$ runs at a time. In the original paper, it is pointed out that larger t increase the hardness of the mROS problem. Since nothing is changed about the signature as such, the verification as shown in Equation (2.6) stays the same and the validity and blindness of the signatures are unchanged. The paper gives explicit proofs of that, which are omitted here.

EdDSA - Ed25519

Lastly, a signature scheme shall be introduced that builds on Schnorr signatures. Schnorr patented his work which kept it from use in open source and open standards. This patent expired in 2008 and since then the scheme enjoys popularity. The Edwards-Curve Digital Signature Algorithm (EdDSA) introduced by Bernstein et al. [16] uses a variant of the Schnorr signature scheme. It is specified in RFC 8032 [50] and is part of NIST's Digital Signature Standard [62]. Ed25519 is one specific instantiation of EdDSA using twisted Edwards version

Blind Schnorr Signature Scheme

Signer		**User**
knows:		knows:
$k_{Signer}, K_{Signer} = \text{keyGen}()$		K_{Signer}
pick random $r \in \mathbb{Z}_q$		pick random $\alpha, \beta \in \mathbb{Z}_q$
$R = rG$		
	$\xrightarrow{R}$	
		blind:
		$R' = R + \alpha G + \beta K_{Signer}$
		$c' = \text{H}(R' \| m)$
		$c = c' + \beta \mod q$
	$\xleftarrow{c}$	
sign:		
$s = r + c \cdot k_{Signer} \mod q$		
	$\xrightarrow{s}$	
		unblind:
		$s' = s + \alpha \mod q$
		verify:
		if $s'G == R + c' \cdot K_{Signer}$
		accept signature as valid
		else signature is not valid

Figure 2.6: The blind Schnorr signature scheme. Generator G and order q are known to both parties, just as hash function H.

Clause Blind Schnorr Signature Scheme

Signer		**User**
knows:		knows:
$k_{Signer}, K_{Signer} = keyGenSchnorr()$		K_{Signer}
pick random $r_1, r_2 \in \mathbb{Z}_q$		pick random $\alpha_1, \beta_1, \alpha_2, \beta_2 \in \mathbb{Z}_q$
$R_1 = r_1 G, R_2 = r_2 G$		
	$\xrightarrow{R_1, R_2}$	
		$R_1' = R_1 + \alpha_1 G + \beta_1 K_{Signer}$
		$R_2' = R_2 + \alpha_2 G + \beta_2 K_{Signer}$
		$c_1' = H(R_1', m)$
		$c_2' = H(R_2', m)$
		$c_1 = c_1' + \beta_1 \mod p$
		$c_2 = c_2' + \beta_2 \mod p$
	$\xleftarrow{c_1, c_2}$	
pick random $b \in \{0, 1\}$		
$s = r_b + c_b \cdot k_{Signer} \mod p$		
	$\xrightarrow{s, b}$	
		verify: $sG \neq R_b + c_b \cdot K_{Signer}$
		if false abort
		$s' = s + \alpha_b \mod p$

Figure 2.7: The Clause Blind Schnorr signature scheme with two runs.

Parameter	Explanation	Ed25519
p	odd prime power for finite field GF(p)	$2^{255} - 19$
b	with $2^{b-1} > p$ for bit length of key and signature	256
H	Hash function with output size 2b	SHA-512
G	base point for curve calculations	B in [50] 5.1
$c \in \{2, 3\}$	base-2 logarithm of cofactor	3
$L, L \cdot G = 0$	$2^c \cdot L$ must be number of curve points	L in [50] 5.1

Table 2.2: EdDSA Parameters and explicit Ed25519 values.

of Curve25519 (edwards25519) as its curve. The most important parameters for EdDSA, used in the key generation, sign and verify operations and their specific values for Ed25519 are visible in Table 2.2.

The key generation algorithm works as follows:

- generate random bit string of length b
- calculate private key s as b least significant bits of $H(k)$
- calculate publick key A as $A = sG$

A signer must do the following to create a signature σ on message m:

- calculate o as b most significant bits of $H(k)$
- calculate $r = H(o||m)$ and interpret r as integer
- calculate $R = rG$
- calculate $s = (r + H(R||A||m) \cdot s) \mod L$
- $\sigma = (R, s)$

Finally, to verify the signature using $\sigma = (R, s), m$ and A, the verifier must do:

- interpret R as curve point
- interpret s as integer and check that $s < L$
- calculate $u = H(R||A||m)$
- accept if $(2^c \cdot s)G = 2^c R + (2^c \cdot u)A$

2.4 Constrained Systems

The section first gives a brief overview about constrained systems before introducing the operating system and hardware used throughout the thesis.

2.4.1 Overview

A constrained system or device is a device with limitations in regard to CPU, memory, power resources, code complexity, available interfaces or other properties. Such constrained devices are used to fulfill tasks in specific environments or settings they are designed for. They can be in charge of collecting information about their environment, forwarding that information to other devices, displaying it or acting on it by performing physical actions. If such systems are capable of network connectivity, a single constrained device in a network becomes a constrained node. RFC 7228 [19] defines terminology for constrained-node networks. It also introduces the possibility of devices interacting and thus benefiting of other "things" close by or somewhere on the internet. The connection of a growing set of diverse devices leads to the Internet of Things (IoT), that is built of uniquely addressable objects (things).

The RFC further delivers a definition for constrained nodes, which characterizes them in contrast to more powerful systems on the internet. It highlights the limitations in size, weight, available power or energy as characteristic that are taken for granted in other devices connected to the internet. Furthermore, examples for limitations are given and devices are grouped in tree classes. A successor to RFC 7228 is currently in draft [20] and enhances this list. It further differentiates the devices in two groups: microcontrollers and general purpose devices. Each group is then categorized in five classes. The full list can be found in the draft.

The communication within a network of constrained nodes should be as secure as within a network of capable nodes. This poses the problem of additional load on the device through the involved cryptographic functionality. RFC 7228 points to RFC 8576 [40] for a broader overview about that topic. There, the authors give a larger overview over the problem field and also the challenges for secure IoT. They mention the concept of *cryptographic agility*, which describes that a device should be capable of reacting to changes in regard to cryptographic assumptions. That means, for instance, that a protocol must be changed because a design flaw was found or that a cryptographic algorithm is no longer deemed secure due to advances in research. A device vendor must be able to react on that by updating the device when necessary. It is further pointed out that especially frequent use of resource-intensive public-key cryptography is a problem if computational power is limited. In [52], the authors present three common options how constrained IoT can handle those limitations. The first option is to use software libraries that are adapted to the circumstances of the constrained devices. The next option is to use a microcontroller that includes peripherals especially for the purpose of cryptographic operations. Lastly, it is also possible to connect specialized external cryptographic hardware to a microcontroller. Hence, possible solutions for secure communication with constrained devices exist, but they need extra effort.

2.4.2 RIOT OS

RIOT [9, 73] is an operating system (OS) for IoT devices. As pointed out in [52], the IoT ecosystem has a very heterogeneous hardware landscape. To build portable software, the use of operating systems for constrained devices grows in popularity. Other such operating systems are, for instance, Contiki [25] or Zephyr [66]. The software handled in this thesis is written for the RIOT operating system.

RIOT OS adapts to constrained systems with different architectures (8 to 32-bit) and has no specific limitations to any hardware platform. It aims to minimize resource usage for RAM and ROM as well as power consumption. Real-time capabilities are provided and code portability across supported hardware is possible. To offer a large degree of flexibility, the system is designed with modularity in mind. Further, it allows the easy inclusion of third party software via a package system, that imports such components during the build process.

In [52], a crypto-subsystem design for the operating system was introduced that handles vendor driver integration, context abstraction, concurrent access as well as power management and state handling of crypto-functions. To further ease the use of cryptographic operations, the OS aims to integrate the PSA Certified Crypto API (psa-crypto) [2]. This is a specification which defines implementation-independent interfaces for different cryptographic operations. Support for that on the OS level makes it possible for users to implement applications against the psa-crypto API and the operating system takes care of the selection of hardware-accelerators or suitable software libraries.

2.4.3 nrf52840DK

The constrained device used throughout this thesis is a nRF52840 Development Kit (nRF52840DK) [8], sold by the company Nordic Semiconductor. It can be categorized in class 4 of the microcontroller group of [20]. The nRF52840DK is a development board with 1 MB flash memory and 256 kB RAM and uses an ARM Cortex-M4 32-bit processor, operating at a clock speed of 64MHz. According to the specifications, the device can be used in computer peripherals, wearables, entertainment devices and the Internet of Things. The device contains a set of sensors, multiple communication and connection options and a set of peripherals. Among others, it contains the ARM Cryptocell-310, a security peripheral to enhance cryptographic operations, which is used throughout the thesis.

CryptoCell-310

The ARM Cryptocell-310 (CC-310) is an ARM intellectual property. It is sold to hardware manufacturers that can implement it in their boards. In the documentation [3], Nordic describes the CC-310 as “a security subsystem providing root of trust (RoT) and cryptographic services for a device.” and mentions that “The following cryptographic features are among the functionality that can be supported: [...] Elliptic curve cryptography (ECC): [...] Edwards/Montgomery curves: Ed25519, Curve25519”. Furthermore, it is stated in the section about the PKA engine that “The PKA engine can be used to hardware accelerate various arithmetic regular and modular mathematical operations involving very large numbers which are used in both RSA and Elliptic Curve Cryptographic (ECC) public-key cryptosystems” and that it supports operand sizes between 128 and 3136 bits. However, the PKA (Public Key Accelerator) engine is effectively a big number ALU (Arithmetic Logic Unit) for modular arithmetic. The support for the mentioned elliptic curve algorithms exists only in the form of the size of numbers the PKA engine can process.

Making use of that has to be done in software. The Nordic SDK [5, 6] offers the functionality described in the documentation as proprietary software that interacts with the CC-310. An

open-source alternative exists in form of the cc3xx driver of Trusted Firmware-M (TF-M) [84]. This driver, however, does not yet support all features of the CC-310. In particular, it does not support twisted Edwards version of Curve25519, which is needed for this thesis. Interacting with the PKA engine is done by writing values into special memory mapped registers and executing operations via opcodes. The CC-310 accesses a special section of RAM, the PKA SRAM (Static RAM), in which all data for the cryptographic operations, except from immediate values, must be stored. Writing to SRAM directly is not possible, which is why all writing must be done via opcodes.

The documentation [3] describes a virtual memory mapping system that must be configured depending on the size of the operands which shall be used. The user defines a certain operand size and depending on that, the available 4KiB of SRAM are split into virtual registers. To access the single registers there are 32 memory mapped registers which contain the start addresses of the virtual registers. The user can only work with those 32 registers at a time. However, the (potential) remainder of the SRAM can be used as well. The SRAM thus works as a private memory pool.

2.5 GNU Taler

Taler is a digital payment system that makes use of the cryptographical basics described so far. Implementation and evaluation of the Clause Blind Schnorr signature scheme in this thesis is done in the context of a Taler version for constrained systems. Thus, an overview over the system as a whole and the process of coin withdrawal explicitly is given.

2.5.1 Overview

The following high-level introduction is adapted from [34]. Taler is a digital payment system that aims to achieve payer anonymity and vendor taxability. It builds on the idea of Chaumian ecash and the concept of blind signatures [22]. Contrary to the current trend of cryptocurrencies based on distributed ledger technologies, Taler does not try to introduce a new form of currency but to tokenize existing currencies. It integrates in existing payment landscapes, aims for easy taxability of income and allows access for auditors.

Figure 2.8 shows the relevant participants in the Taler system as well as their relations. A customer in the system obtains the right to receive coins by transferring fiat currency to an exchange via bank transfer. The exchange handles incoming payments and makes sure that only who transferred money in the first place is eligible to withdraw coins. The customer may withdraw coins by use of a wallet application that handles their receiving, spending and safekeeping. Spending coins happens upon interaction with a vendor that accepts such coins as a means of payment. When a customer agrees to pay with coins he uses the coins to confirm a payment towards the vendor, which then can verify the validity of the coins with the exchange that handed them out in the first place. Upon confirmation the vendor accepts the payment and can now ask the exchange for a payout of the coins in the form of fiat currency to his business account with a bank. Thus, the exchange works as an escrow that keeps customers money until it is spent and then hands it to the appropriate recipient.

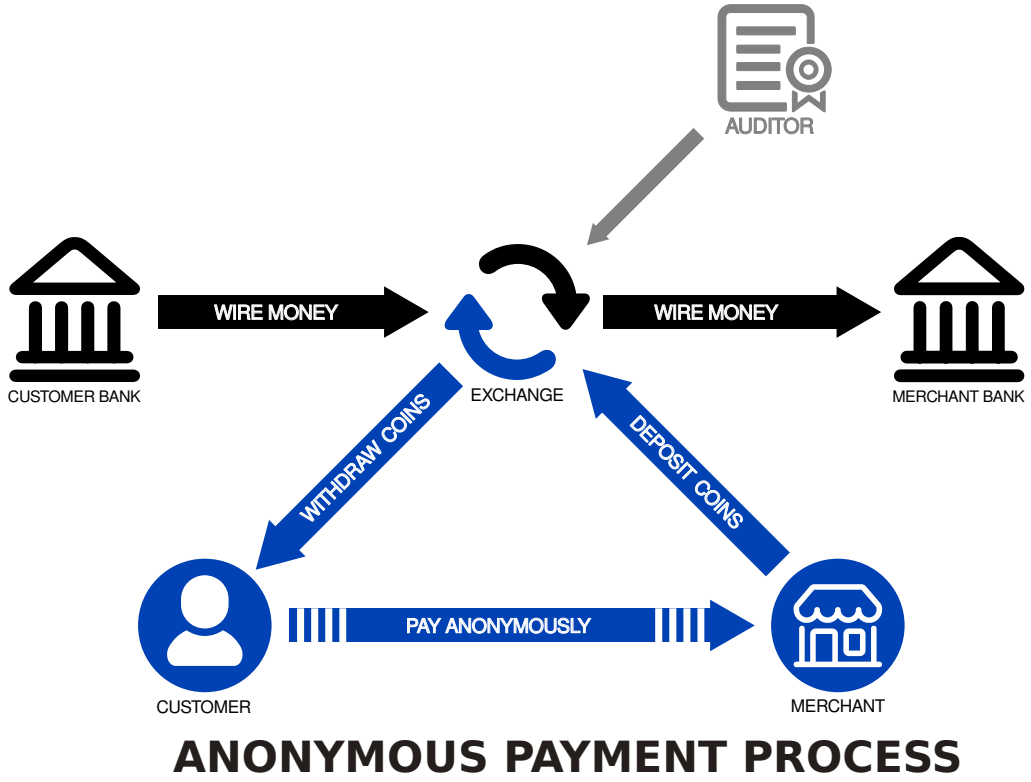

Figure 2.8: Overview over the Taler payment process and participants. Source: [37]

Multiple exchanges can exist and the user is free to choose any of them. In order to ensure trust in those exchanges, they are auditable. Auditors should be trusted entities like financial regulation authorities. The GNU Taler system is aware of important parts of modern payment landscapes like Anti Money Laundy (AML) and Know Your Customer (KYC) processes. Details about those are omitted here as they are of less importance to the following thesis but can be found in [34]. The most relevant part for this thesis is how the coins come into existence and how value is attributed to them.

2.5.2 Withdrawal

The process of obtaining valid coins from an exchange, called withdrawal, happens after a customer has sent funds to an exchange. The funds are associated with a so-called reserve, which is an Ed25519 key pair w_s, W_p generated by the customer. With the money transfer to the bank the customer also informs the exchange about the public key W_p. Withdrawing a coin from the exchange starts with the creation of another Ed25519 key pair c_s, C_p. Next, the customer decides about what value the coin shall have. To do so he picks a denomination public key D_p. An exchange offers a list of such denomination keys and thus decides which denominations (e.g. 0.5€, 1€, 10€) it offers. Under use of D_p and a random secret, the public key C_p is blinded by the customer and sent to the exchange in a withdrawal request. This

request further contains the reserve public key W_p, and the chosen denomination public key D_p. The exchange checks the request and, if valid, deducts the appropriate amount from the customer's reserve, signs the blinded coin private key and returns the signature. Upon reception, the user unblinds the signature and stores the coin key pair, the denomination public key and the unblinded signature. This set forms the coin that can later be used for payment with a merchant. The customer is the sole owner of the private key and can use it to prove ownership. The unblinded signature can be used to prove validity of the coin towards a vendor.

Using a blind signature scheme for the withdrawal achieves payer anonymity as the exchange will not be able to make a connection between the coin withdrawn by the user and the coin used to pay at the merchant, other than that it has a valid signature. Anonymity towards the merchant depends on the circumstances of the buying process. Buying something in a shop where an identification is needed will of course reveal the customer's identity towards the merchant but still not towards the exchange. If no identification is needed, the merchant learns only that the payer obtained the coin from the exchange that issued the signature.

In [34], the blind signature system of choice is the RSA-FDH scheme and the exact protocol is visible in Section 4.7.2 of that work. [29] adds Clause Blind Schnorr (CBS) as an alternative scheme to Taler. This alternative shall offer cipher agility for Taler and is found to be faster while saving storage space and bandwidth. The choice of the blind signature scheme plays a role in more aspects than the withdrawal process and both works cover those in more detail. Here, the focus remains on the withdrawal protocol, specifically the customer side of it. Thus, Figures 3.1 and 3.2 of [29], depicting the withdrawal process with CBS, are replicated in Figure 2.9, with reduced information about the exchange side.

It should be noted that this protocol is designed with *abort idempotency* in mind. That means, that in case of connection issues, aborts or timeouts, it should be possible to restart the communication with the same values. To achieve this, the only randomly generated value in the protocol is the withdraw secret ω. All following values are derived from that and other values generated throughout the withdrawal process. The exchange uses the withdraw nonce n_ω (derived from ω by the customer) for the derivation of the R values as well as the signature generation (see Figures 3.1 and 3.2 of [29] for details). Hence, the complete interaction can be recreated from the withdraw secret. The blinding, unblinding and verification primitives are the same as shown in Figure 2.7, just that the blinding values are also derived and not randomly generated. The message to blind is the public key of the coin C_p and the denomination public key D_p is used as public key of the signer. Note also, that the derivation of the blinding secrets $\alpha_0, \alpha_1, \beta_0, \beta_1$ with in a single derivation is depicted as four single derivations in the original work but summarized here for brevity.

The decision whether coins signed with the CBS approach can be withdrawn lies with the exchange. For use of the CBS scheme the denomination public keys must be Ed25519 public keys, for use of the RSA-FDH scheme they must be RSA keys. It is possible to make a choice for one of the schemes or offer both of them at the same time. A customer can only choose between the denominations offered by the exchanges. It should be noted, however, that in [29] the integration of CBS in the wallet is noted under future work and at the time of writing of this thesis this still remains future work. In the exchange software, offering CBS denominations is supported.

Further details about the workings of the payment system, how double spending is prohibited, how change is issued and how the other participants can interact can be found in [34]. [29] gives more insight in the integration of the CBS scheme in the other parts of the system as well. Lastly, the project website [36] gives the latest information about the current state of the payment system, its availability and further documentation.

GNU Taler Withdrawal Protocol using Clause Blind Schnorr

Customer		**Exchange**
knows:		knows:
reserve keys w_s, W_p		reserve public key W_p
denomination public key D_p		denomination keys d_s, D_p
generate withdraw secret:		
$\omega = randombytes(32)$		
derive withdraw nonce:		
$n_\omega = \text{HKDF}(256, \omega, "n")$		
	$n_\omega, D_p \longrightarrow$	
		generate R values
	$\longleftarrow R_0, R_1$	
derive coin key pair:		
$c_s = \text{HKDF}(256, \omega \vert\vert R_0 \vert\vert R_1, \text{"cs"})$		
$C_p = \text{Ed25519.GetPub}(c_s)$		
blind:		
$b_s = \text{HKDF}(256, \omega \vert\vert R_0 \vert\vert R_1, \text{"b-seed"})$		
$\alpha_0, \alpha_1, \beta_0, \beta_1 = \text{HKDF}(1024, b_s, \text{"alphabeta"})$		
$R'_0 = R_0 + \alpha_0 G + \beta_0 D_p$		
$R'_1 = R_1 + \alpha_1 G + \beta_1 D_p$		
$c'_0 = H(R'_0, C_p)$		
$c'_1 = H(R'_1, C_p)$		
$c_0 = c'0 + \beta_0 \mod p$		
$c_1 = c'1 + \beta_1 \mod p$		
sign and send blinded values:		
$\rho_W =< n_\omega, D_p, c_0, c_1 >$		
$\sigma_W = \text{Ed25519.Sign}(w_s, \rho_W)$		
	$W_p, \sigma_W, \rho_W \longrightarrow$	
		decide for one run, sign
	$\longleftarrow b \in \{0,1\}, s$	
verify signature:		
$sG = R_b + c_b D_p$		
unblind:		
$s' = s + \alpha_b \mod p$		
verify unblinded signature:		
$s'G = R'_b + c'_b D_p$		
$\sigma_C =< R'_b, s' >$		
store coin: c_s, C_p, σ_C, D_p		

Figure 2.9: Clause Blind Schnorr Withdrawal process adapted from Figures 3.1 and 3.2 in [29]. The exchange side is kept very short as it is not implemented here.

3 Related Work

Two aspects were relevant for the review of related works: the security of the Clause Blind Schnorr (CBS) scheme and the use of cryptography on constrained systems. Reviewed work for both aspects will be presented in the following.

3.1 Recent Developments in Blind Signatures

The core question in this aspect of literature review was, whether the CBS signatures are still considered to be safe to use. Demarmels and Heutzeveld [29] already pointed out that CBS and the mROS problem are relatively young and not as well examined as RSA. Moreover, the authors of the CBS scheme updated their paper [38] with a note in which they point out that they are not vulnerable to a new attack [13] that was found roughly at the same time. This seems reason enough to investigate whether new attacks were found that can attack CBS.

In 2025 Joux et al. [51] published an improved attack on the ROS problem. This is effectively an improvement on the attack by Benhamouda et al. which was already mentioned not to affect the mROS problem and thus CBS. The authors of the improved attack list mROS as one of the variations of the ROS problems that is not affected. While this is a signal for the current safety of mROS and thus clause blind Schnorr, it shows at the same time that the search for vulnerabilities is ongoing.

Tessaro and Zhu in 2022 [82] introduced a new blind signature scheme together with a new problem on which their proofs rely, the *weighted fractional* ROS (WFROS) problem. Their scheme allows higher security levels than CBS or rather the underlying mROS problem. They also point out that CBS was mentioned as the only plausible alternative to RSA blind signatures in a draft for a blind signature RFC [30] This draft became RFC 9474 in 2023 [31] but lost the mentions of alternative schemes after version 7 [21, 32]. Up until then the authors listed the Clause Blind Schnorr signatures as well as the WFROS based blind signatures as alternatives that need further experimentation and analysis. Regarding the security level of CBS they point out that a security parameter of 512-bit (which translates to a 512-bit curve) is needed to achieve a security level of 128-bit in all cases and that their scheme allows higher security levels. They list the original paper as source for the security argument. Indeed, looking at Figure 11 in [38], it is visible that for security parameter 256-bit and small amounts

of parallel sessions (around 10), the complexity of the best attacks drops to around 70.

This observation is repeated by Fuchsbauer, one of the original authors of CBS, in a work published together with Wolf in 2024 [39]. Here they also mention that the scheme is still secure but achieves only 70-bit security when used with 256-bit curves. It is additionally pointed out that the amount of parallel signing sessions (parameter t in the explanation in Section 2.3.2) could be increased to make the mROS problem harder to solve. However, they also state that the amount of $t > 2^{14}$ sessions that would be needed to achieve a 128 bit security level would tremendously increase the communication complexity.

The same is stated in 2024 by Harding and Xu in [47](p.3) even more explicitly for the circumstances examined in this thesis: "Considering compatibility with existing standards, only two of the aforementioned schemes are at all relevant: Clause Blind Schnorr and Fuchsbauer-Wolf. The former appears incompatible with Ed25519 because it only achieves roughly 70 bits of security when instantiated with a 256-bit curve.".

Summarizing the papers mentioned above, the consensus seems to be that Clause Blind Schnorr offers insufficient security levels with 256-bit curves and at least 512-bit curves are needed to achieve 128 bits of security. As this is already stated in the original paper, this does not seem to be a new development, it is rather stated more explicitly and the search for blind signatures on standard 256-bit curves seems to be ongoing.

3.2 Cryptography on Embedded Systems

This section covers related work that handles the topic of cryptography on embedded and constrained systems. The Internet of Things (IoT) has a reputation of being insecure and a gateway for security problems into people's homes. Usually the main selling point of IoT devices is their functionality, not their security. Nevertheless, vendors should be interested in designing secure products as their reputation suffers when their devices get hacked and such knowledge becomes published.

A direct predecessor of this thesis is [44] by Gütschow and Wählisch from 2025. Here, the exact system setup as used in this thesis is used to showcase the practicability of running GNU Taler on constrained devices. The work contains a comparison of the necessary cryptographic operations implemented in hardware and software. RSA-FDH blind signatures are computed with the software library RELIC [67] or hardware-accelerated with the crypto peripheral of the nRF52840DK, [7] a ARM Cryptocell-310 [3, 59]. Elliptic curve non-blind signature creation and verification is done using the Ed25519 signature scheme. For that, the software library in use is c25519 [11], the hardware accelerator is the same as used for RSA-FDH. The results show that the hardware-accelerated versions of the crypto schemes are one to two orders of magnitude faster, which highlights the need for hardware-acceleration on constrained devices. All operations for the Ed25519 signature scheme takes around 20ms when executed with hardware-accelleration. Those findings are in line with those of the next work.

In 2021, Kietzman et al. [52] presented an overview over crypto-hardware performance for IoT devices. Their report contains a detailed review of elliptic curve cryptographic operations on an nRF52840DK board in combination with the RIOT operating system [73]. As this is a similar system setup (apart from the version of RIOT) as in this thesis, the results should

serve as data for comparison. A difference is that the operations examined in the overview are the cryptographic primitives of ECDSA [64] key generation, signing and verification as well as ECDH [58] key exchange over curve NIST P-256 [24]. This thesis works directly with elliptic curve arithmetic as the CBS scheme is not available as cryptographic primitive. Also, the curve used here is twisted Edwards version of Curve25519 (edwards25519). Their comparison includes the cryptographic libraries RELIC [67] and uECC [60] as well as the hardware accelerators of the nRF52840DK (CC-310) [3] and the ATECC608A (an external crypto chip). In the paper, the differences in the implementation of the elliptic curve arithmetic in the software libraries as well as the hardware drivers are considered. Their finding is that the hardware-accelerated operations perform faster than the pure software-based versions and that the nRF52840DK using the CC-310 outperforms the external crypto chip. It is found that all measured operations take around 20 ms on the nRF52840DK, which is one order of magnitude below the software. While the curve and operations examined in the paper are different from this thesis, it is still reasonable to assume that the hardware-accelerated implementation will outperform the software implementation.

In [18] from 2021, Bisheh-Niasar et al. describe the design of an Ed25519 implementation using FPGAs. This is the same signature system as used in [44]. Their results of execution times as low as 0.18 ms for a single signature creation show that specialized implementations can achieve higher speeds than those found in the two works presented prior. However, the implementation effort of creating a custom-built FPGA is significantly higher and has a different scope. For this work, only commercial of-the-shelve devices are considered.

In [43] from 2004, Gura et al. compare RSA and elliptic curve cryptographic computations on 8-bit processors. Their findings are that elliptic curve cryptography (ECC) performs faster than RSA, especially when the word size of the processor shrinks. This is of interest in so far as that it compares RSA and ECC, just as is done here in a special use case. The curves in scope are different again and here really just single operations are compared and reviewed. It does give, however, an interesting insight in the implementation approach taken by the authors.

4 Design and Implementation

As explained in Sections 1.2 and 1.3, the goal of this thesis is to implement and evaluate Clause Blind Schnorr (CBS) signatures for a GNU Taler (Taler) wallet application for low-end IoT devices. This chapter will start with an overview of the system setup and initial situation. Next, the design of the missing functionality and the integration in the existing wallet software will be described. Lastly, details of the implementation steps will be presented.

4.1 Overview

The Taler on RIOT (taler-riot) [87] wallet is an application written for the RIOT [73] operating system at the Chair of Distributed and Networked Systems of the Dresden University of Technology. It has been implemented as a demonstrator, to showcase that the payment system can be run on low-end IoT software and can handle withdrawals and payments. However, it is not possible to run the application solely on a microcontroller by now. In order to make a connection to a Taler exchange, it must be connected to a computer via USB. It then sends data via CoAP and CBOR over the USB connection to a proxy, running on the computer, which in turn forwards the data to the Taler Exchange instance via http(s) and JSON. In order to withdraw Taler tokens, the software contains a pure software-based implementation of RSA-FDH blind signatures as well as a hardware-accelerated version. The hardware-accelerated version is restricted to a limited number of boards as it expects an CC-310 [59]. That peripheral is available on the nRF52840 Development Kit (nRF52840DK) [3, 7], which is the board used for this implementation as well. Section 2.4.3 gives an overview over the board as well as the relevant features and functionality of the CC-310. To make use of the CC-310, the *cc3xx* driver is used, which is part of the open-source Trusted Firmware-M (TF-M) project [84]. This driver aims to support the CC-310 as well as its successor, the ARM Cryptocell-312 (CC-312) [4, 59]. The pure software-based version uses the library *RELIC* [67]. In general, the taler-riot software aims to use the psa-crypto [1] as much as possible. This specification defines implementation-independent interfaces for different cryptographic operations. The RIOT operating system integrates psa-crypto on an OS level [71] to ease the use of cryptographic operations and automatically pick a suitable hardware or software backend. However, blind signatures are not part of psa-crypto, and thus they must be

implemented under direct use of cryptographic libraries.

In order to have a use-case that can be tested and compared to the existing solution, this implementation focuses on the withdrawal process as introduced in Section 2.5.2. The blind, unblind and verify operations are depicted in the overall withdrawal protocol in Figure 2.9. The taler-riot software shall be enhanced with an implementation of the CBS scheme that contains the necessary operations for the withdrawal protocol.

4.2 Design

Three parts are relevant to design: the overall integration of the functionality into the wallet, the software-based and the hardware-accelerated CBS implementation.

4.2.1 Integration in Wallet Application

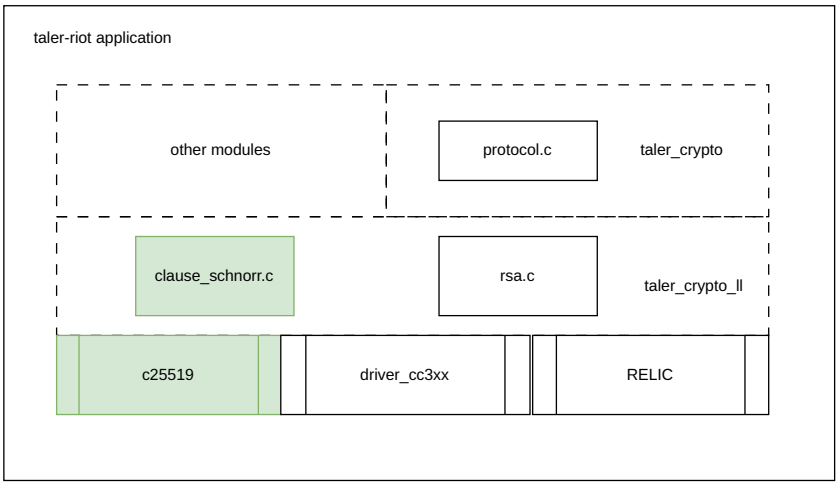

Figure 4.1: Taler-RIOT architecture overview. Green boxes symbolize parts that are added for the CBS scheme.

The additional functionality for CBS is added to the existing software with as little changes as necessary. Figure 4.1 depicts the architecture of the relevant modules including the necessary additions. The taler-riot application is built in a modular fashion. Overall orchestration of the withdrawal process is handled by the protocol compilation unit in the `taler_crypto` module. To do so, it calls the blind, unblind and verify operations of the RSA compilation unit in the `crypto_low_level` module. Communication and persistence is handled in other modules that are also called from the protocol unit during a withdrawal execution. The CBS implementation is added to the `crypto_low_level` module, next to the RSA implementation. From an interface perspective, the necessary operations for both blind signature schemes are the same as explained in Section 2.3: key generation, blind, unblind and verify. For CBS it

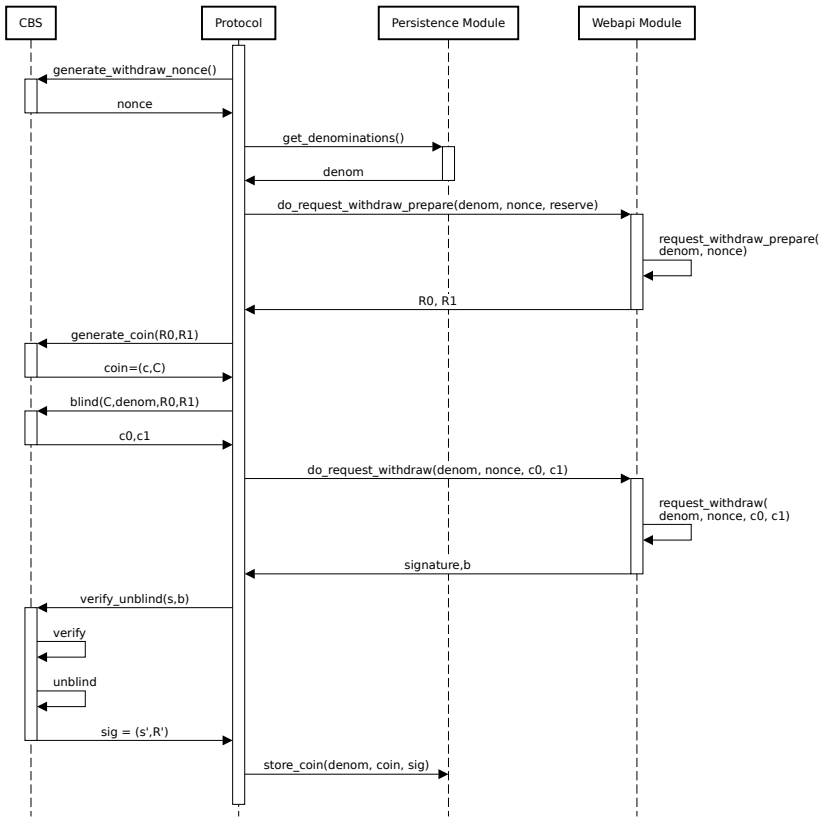

Figure 4.2: Simplified Withdrawal with Participants. Self-interactions of the Webapi Module signify communication with an exchange.

is additionally necessary to have a method to generate the withdraw secret ω and nonce n_ω to initiate the communication with the exchange. Figure 4.2 visualizes the interaction of the protocol unit with the other parts of the software in a simplified manner.

The authors of [29] picked edwards25519 as underlying curve and the same is done here. This curve is also the basis for the Ed25519 signature system which is used to create the reserve (w_s, W_p) as well as coin (c_s, C_p) key pairs and to sign messages between wallet and exchange. An Ed25519 implementation exists already and will not be re-implemented. The signer's public key used in the CBS blinding process is the denomination public key D_p created by the exchange. The only necessary change for the customer regarding the coin generation is the derivation of the coin's private key. Obtaining the public key can be done with the existing Ed25519 implementation. Functionality to execute the HMAC-based key derivation function (HKDF) to derive the secret key from the randomly generated withdraw secret ω also exists already in taler-riot. Obtaining ω, that is, the generation of random values, is also already implemented. Blinding consists of the derivation of the blinding values $\alpha_1, \alpha_2, \beta_1, \beta_2$ and the calculation of the R', c' and c values. The blinding values are derived from a blinding-seed b_s which is in turn derived from the blinding secret and the R values received from the exchange. c' is the result of a hash function. More explicitly, a Full Domain

Hash (FDH) (see Section 2.3.1) with codomain matching order of the elliptic curve group of edwards25519. A FDH is already implemented for the RSA-FDH implementation and can be reused.

The calculations of the R', c' and c values as part of the blinding operation does not differ from the proposed scheme of [38]. The same is true for unblinding and verification. The operations consist of actual calculations on the elliptic curve and must be done by use of the cryptographic libraries. Table 4.1 summarizes the single values needed during the withdrawal protocol, what functionality is needed to generate them and whether it is necessary to implement the functionality. The missing parts are the elliptic curve operations point addition and scalar multiplication and the addition of scalar values, which effectively is a modular big number addition. Those functions should be delivered by cryptographic libraries, if possible. To make use of external libraries, the RIOT module system shall be used.

Value / Calculation	Necessary functionality	Implementation necessary
ω	generation of random bytes	no, reuse existing
$n_\omega, c_p, b_s,$ α, β	HKDF	no, reuse existing
C_p	Ed25519.getPub()	no, reuse existing
c'	FDH	no, reuse existing
$R', sG, R + c \cdot D_p$	scalar multiplication, point addition	yes
$s + \alpha \mod p,$ $c' + \beta \mod p$	scalar value addition	yes

Table 4.1: Summary of necessary implementations ordered by the single steps of the withdrawal protocol.

4.2.2 Software-Based Approach

A cryptographic library must be chosen and added to the software project that offers the needed functionality. Two choices of cryptographic libraries appear obvious. RELIC [67] is the library already in use for the RSA software-based implementation. The second choice is c25519 [11] as it is the library used in the predecessor of this work [44]. Both support the edwards25519 curve, offer the necessary operations for curve points and scalars and are available as RIOT modules [69] [72]. The choice is made for c25519 as it is specialized to the needed curve and does not add unnecessary features or requires configuration. Also, given the use in the former work, the results found there could be used for comparison. Integrating the library in the taler-riot software is easily done via the RIOT build system and does not require further work. As it is already used to create the key pairs for coins and reserves it already in use and has no negative impact on the memory requirements.

4.2.3 Hardware-Accelerated Approach

The hardware-based approach shall make use of the CC-310 to implement the necessary operations. One driver, already existing as RIOT module [68], is part of the Nordic nRF5 SDK [6]. It offers hardware-accelerated Ed25519 primitives [5] for key generation, signing and verification. What it does not offer is access to the necessary arithmetic operations on the edwards25519 curve. Further, it is not open-source and can not be modified. Hence, this driver can not be used to implement the CBS scheme. The second available option is the cc3xx driver of TF-M [84] already used in the hardware-acellerated RSA blind signature scheme. This driver contains a general elliptic curve arithmetic interface that can be used with different elliptic curve types. However, only Weierstrass curves are supported and no twisted Edwards curves as needed here. As this driver is open-source it can be modified and the functionality can be added. No other driver could be found that offers the required functionality and thus the cc3xx driver seems to be the sole option. Hence, the elliptic curve operations on edwards25519 are implemented for this driver. As it is already in use in the taler-riot software there is no integration effort for the driver as such.

4.2.4 Testing

Both approaches need to be tested for correct calculations and operations. To obtain test values for the overall CBS operation, two sources are used. The first, is the test suite of GNUnet [42], which contains the implementation of cryptographic operations used by the Taler exchange software. The tests in `test_crypto_cs.c` [41] are enhanced with print statements and then executed to obtain test vectors from the resulting test log. The second source are the tests in the wallet software repository [81]. That repository includes a set of test vectors useable for one complete withdrawal run. Those are encoded with Crockford's Base32 variant [26] according to the exchange API specification [35]. Having a set of test vectors in encoded form, as expected from a real communication with the exchange, is a good second set of values. Code for decoding is already available in the taler-riot application and can be reused.

In order to test the proper implementation of the hardware-accelerated approach it is necessary to test the elliptic curve implementation. The tests must ensure that scalar multiplication, point addition and addition of scalar values is executed correctly. To do so, existing libraries, of which correct calculations can be expected, shall be used to create test vectors. Tests can be taken from the tests contained in the source code of c25519 as it also contains tests regarding proper arithmetic. As a second source of test vectors the python library LightECC [79] is chosen, that allows easy interactive testing and calculations.

4.3 Implementation

Specific aspects of the implementation will be described here, beginning with the pure software-based approach. Next, the hardware-accelerated approach will be described and lastly the integration of both in the existing software. This follows the bottom-up approach in which the implementation took place.

In terms of general project setup, the software for the thesis was simply added to the GitHub repository of taler-riot on a new branch [88]. The c25519 library was already available as RIOT module and was simply added via the appropriate Makefile. To be able to easily make changes to the TF-M cc3xx driver, its original repository [84] was forked [85] and the dependency in the taler-riot repository was changed to use the fork.

4.3.1 Software-Based Approach

As c25519 includes all necessary operations for the withdrawal operation, the main task was to get data into and out of the application. Scalar values can be handed to the library in form of `uint8_t` arrays and do not need any conversion. Addition of scalar values can be done by use of the functions for the underlying field $GF(2^{255} - 19)$ and subsequent reduction by the curve order. Points on the elliptic curve can be handed to the application directly as x and y coordinate pair, where each coordinate is again a `uint8_t` array. If the point is given in compressed form, `ed25519_try_unpack` can be used to unpack the point to such a pair. Point compression and decompression is specified in 5.1.3 of RFC 8032 [50] and is done as such in the library. In order to make computations with the points, they must be turned into extended coordinates (x, y, t, z) (see Section 2.1.3) via the `ed25519_project` function. This yields a `struct ed25519_pt` which is the structure expected by the functions for scalar multiplication and point addition. Once computations are done, the point can be turned back into a (x, y) pair of coordinates via `ed25519_unproject` and then this pair can be compressed by use of `ed25519_pack`. By use of those functions, the CBS functionality could be implemented. To simplify the input and output of with the library a wrapper function was written to bundle the (un)packing and (un)projection function calls. Tests to verify the correctness of the implementation were created by use of test vectors obtained from the taler-wallet-core repository and GNUnet as described in Section 4.2.4. They were added to the existing test structure of the taler-riot application.

4.3.2 Hardware-Accelerated Approach

For the hardware-accelerated approach, all functionality for curve arithmetic and point compression had to be added to the cc3xx driver. The driver already offered options to use the PKA engine of the ARM Cryptocell-310 (CC-310) (see Section 2.4.3) as well as a general framework for elliptic curves. Supporting the edwards25519 curve in the driver is the necessary foundation to use it in the taler-riot application and implement the Clause Blind Schnorr (CBS) signature scheme. Hereinafter these points will be explained: the elliptic curve framework of TF-M, the integration of the new curve type and the implementation of the arithmetic operations.

Elliptic Curve Framework

The cc3xx driver already contained functionality to use Weierstrass type curves for elliptic curve cryptography. This is embedded in a framework for general elliptic curve cryptography and arithmetic. A user is supposed to interact with an interface that delegates the requested

operations to the matching curve implementation. Figure 4.3 shows a simplified interaction of a user with the interface. The depicted use-case is a scalar multiplication that includes most relevant operations needed later on.

The interaction starts with an initialization action that takes care of the initialization of the PKA engine as well as the writing of curve parameters into the PKA SRAM. This initializes a `cc3xx_ec_curve_t` structure that is needed for future operations. This object is held by the user and must be handed to the EC interface for each operation.

A structure `cc3xx_ec_point_affine` to handle points consisting of (x, y) coordinate pairs (affine coordinates) are offered by the interface. This structure contains only register IDs. Allocation of space for points in the PKA SRAM is handled by the EC interface. As depicted, the user can either provide coordinates that will be written directly to the SRAM or just request a point object with empty coordinates. Scalar values, on the other hand, must be handed to the PKA by the user. The EC interface accepts them in form of register IDs.

To execute elliptic curve operations, the user needs to supply the curve object, scalars and point objects including the result location. The calculation of the result is handled by the specific curve implementation. It employs the PKA to calculate the single steps of an operation. Final results must be transformed back into affine coordinate pairs if the curve implementation used different coordinates for calculations. A user can finally read the (x, y) values from the PKA SRAM via the matching calls to the PKA engine and the affine point structure. After use, the PKA engine must be uninitialized, which is done via the matching call to the EC interface.

Accordingly, the first necessary addition to the driver is the new curve type in form of parameters for initialization and later use, as well as the concrete edwards25519 curve information set. Then, according to the interface each curve type should support the operations:

- Point addition $(R = P + Q)$
- Point doubling $(R = P + P)$
- Scalar multiplication of a point and a scalar $(R = sP)$
- Scalar multiplication of two points followed by addition $(R = aP + bQ)$

To implement addition and doubling, a representation of extended coordinates must be implemented (see also Section 4.3.1). This must be able to return affine coordinates at the end of any calculation. In order to actually make use of the PKA engine, addition and doubling must be implemented as direct interaction with the PKA engine. Once done, scalar multiplication can be added as sequence of additions and doublings of points. The last operation, not part of the interface but necessary for the integration, is point compression and decompression for points on curve edwards25519.

Adding Twisted Edwards Curves

Multiple sources were used as blueprint and support to implement the new curve type. The Weierstrass implementation that already existed in the driver was used to understand how a curve is reflected as data structure and which elements are needed. The second source was

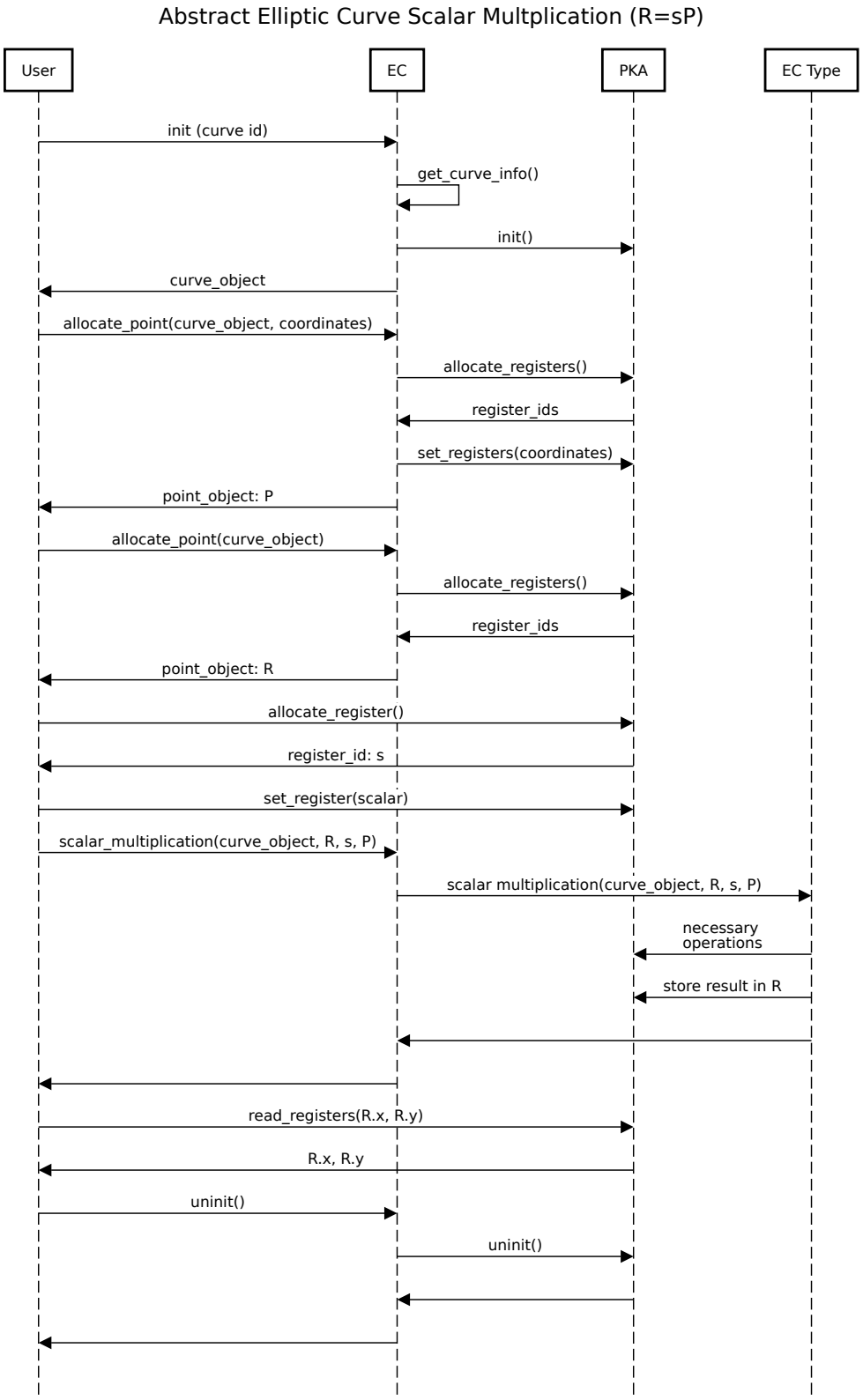

Figure 4.3: Abstract scalar multiplication example. This example shows the interactions between a user, the EC interface and the PKA engine. The markings are pseudocode.

an old driver for the CC-312 that is also contained in the repository of TF-M [83]. It contains an implementation of Ed25519 and thus the necessary information how to represent the curve as data structure and how to handle the PKA operations and initialization. The c25519 library, used in the software-based implementation, was also used as a reference to have an implementation in C code instead of PKA operations. Next to that, different guides [27, 53, 90–92], books [46, 63, 80] and papers [16, 28, 48] served as valuable sources.

Following the flow visible in Figure 4.3, the initialization is the first thing to adapt. This step writes all values to the PKA SRAM that are later needed throughout operations and stores the register IDs in the curve object. The actual values for a concrete curve are hardcoded as `cc3xx_ec_curve_data_t` structs. During initialization the right structure is selected via the curve identifier, given by the user. Only three parameters had to be added to the structure to be able to work with it. One was the curve parameter d, directly visible from the curve equation ($ax^2 + y^2 = 1 + dx^2y^2$). The other two were related to point decompression and adapted from the mentioned old CC-312 driver. All values for edwards25519 were added to an already existing but empty struct for curve edwards25519 in the curve data compilation unit. Further the initialization step was modified to make use of the new elements in the `cc3xx_ec_curve_data_t` struct.

Addition and Doubling of Points

The Explicit-Formulas Database (EFD) [14, 15] served as valuable source for addition and doubling formulas. The sources mentioned above point to the EFD as source for the same purpose as well.

"add-2008-hwcd-3" and a mix of "mdbl-2008-hwcd" and "dbl-2008-hwcd" obtained from the EFD were chosen for addition and doubling. The addition formula is also in use in c25519 and was used without further reasoning. The same applies to the doubling formula "dbl-2008-hwcd". This was mixed with "mdbl-2008-hwcd" which assumes that the z coordinate of the point to double is one. The mix was implemented because the second approach needs fewer operations during execution. However, the mix comes with the additional overhead of ensuring that the assumptions are fulfilled and it was not tested whether that is a performance gain or loss. What should also be mentioned, is that the formula is designed for twisted Edwards curves with curve parameter $a = -1$. This is valid for edwards25519 but must be reconsidered when further curves shall be added. Implementing the formulas meant using the CC-310's PKA engine to execute the single addition and multiplication steps. Interaction with it is handled via the cc3xx driver as well. All necessary opcodes were available. The EFD offers the used formulas in the form of three-operand codes which are close to the implementation as PKA operations.

As mentioned, those formulas make use of extended coordinates (x, y, z, t). The Weierstrass curve implementation in the cc3xx driver uses yet another type of coordinates already: projective coordinates (x, y, z). It served as orientation how to add and use different coordinates. Essentially, a point in extended coordinates needs a new structure with PKA register IDs for each coordinate, functions to allocate SRAM space and to transform to and from affine coordinates. Additionally, a structure to store the coordinates of extended points on stack was added for the 4-bit window scalar multiplication approach, explained in the next section.

Scalar Multiplication

There are multiple possible approaches to scalar multiplication. The simplest possible implementation, executing as many additions as defined by the scalar (e.g. $3P = P + P + P$), scales very badly and is not a useful solution. Two different versions were implemented and their performance compared (see Section 5.2). Both were adapted from the explanations in [91] and the code in [11] and [83]. The first version is called double-and-add-always (daa), a variant of the double-and-add approach with constant execution time to achieve side-channel resistance, which is also used in c25519. The second version is called 4-bit window approach in the following and is implemented for faster execution time.

Double-and-add-always

```
daa(result, point, scalar):
    result, tmp = 0
    bit_list[max_scalar_length] = [0,...,0]
    bit_list = bits_in_scalar_msb_to_lsb(scalar)
    for bit in bit_list:
      res = double_point(res) //double
      if bit == 1:
        res = add_points(res,point) //add
      else:
        tmp = add_points(res,point) //always
    return res
```

Listing 4.1: Pseudo Code for double-and-add-always

The daa concept is shown as pseudocode in Listing 4.1. The single bits of the scalar are iterated from most to least significant bit. In each iteration, the result, which is initially zero, is doubled unconditionally. If the bit in the current iteration is set to one, the point to be multiplied is added to the result after it is doubled. If it is not set, a calculation is done anyways to maintain constant execution time.

Adding only when the current bit equals one would cause a higher execution time for ones than for zeroes. Measuring the different execution times would yield the bit representation of the scalar. As the scalar may be a secret, this must be avoided. Accordingly, an addition is also done in any case. The second factor for constant execution time is the length of the scalar. The bit representation of the scalar must be prefixed with as many 0s as necessary to have the maximum length of possible scalar values. Otherwise, the amount of iterations and thus the execution time would differ depending on the position of the most significant set bit in the scalar.

4-bit Window

The second approach makes use of a 4-bit window to achieve faster calculations. It was implemented to lower the execution time of a scalar multiplication. This is achieved by lowering the amount of addition operations of the daa approach. Instead of one addition per iteration, 4-bits are evaluated together and an addition is done only every four iterations.

```
4b_window(result, point, scalar):
  res = 0
  bit_list[max_scalar_length] = [0,...,0]
  table = calculate_table(point)
  bit_position = max_scalar_length-1
  bit_list = bits_in_scalar_lsb_to_msb(scalar)
  for bit in bit_list:
      res = double_point(res)
      if bit_position % 4 == 0:
        //interpret bits as integer e.g. 0101 = 5, 1111 = 15
        w = bit_list[bit_position+3 : bit_position]
        res = add_points(res,table[w])
      bit_position = bit_position-1
  return res

calculate_table(point):
  table[4 * 4] = [0,...,0]
  table[1] = point
  for i in 2...15:
    table[i] = add_points(table[i-1],point)
    //table[2] = 2*point, table[3]=3*point,table[15]=15*point
  return table
```

Listing 4.2: Pseudocode for 4-bit window

The approach makes use of a table that is computed in the beginning of the multiplication. Listing 4.2 shows the concept as a whole in pseudocode, including the table generation. It contains multiples of the point P to be multiplied, from $0 \cdot P$ to $15 \cdot P$, necessary for the actual calculation. The loop is the same as for the daa approach, from the most significant bit to the least significant bit of the scalar, prefixed with zeroes to the maximum length. It also starts with the unconditional doubling of the result point, but additions happen only every fourth iteration. Then, the four bits, starting from the current location in the bit string are evaluated as integer value between 0 and 15. The matching multiple of P is read from the table and added to the result. Compared to the daa approach, the amount of additions necessary for one scalar multiplication is decreased by 75%. The amount of doublings does not change. This comes at the expense of space and additions needed to hold and fill the table. 14 additions are needed to generate the table entries 2 to 15. In the current implementation they are stored as point representations on the stack and written to the PKA SRAM when

needed for addition. Storing the table in the SRAM would save the operations needed for reading and writing and might bring a speedup in the implementation. However, the table contains 16 points with 4 coordinates of 32 Byte each. Thus, a table is 2048 Byte in size which is half the SRAM space. Exploring this tradeoff is future work. The additions to generate the table can be saved in case the point to be multiplied is known in advance. If so, the table can then be hardcoded or persisted after initial generation. The only point that is known in advance here is the generator point of the curve. Its table was added to the code. Section 5.2 contains performance measurements for that optimization as well.

It should be mentioned that the 4-bit window approach is the *n-bit* window approach with $n = 4$, meaning that the amount of bits that make the window can be modified. Using a smaller/larger window implies less/more space necessary for the table and less/more operations to generate it and in turn means less/more saved additions per scalar multiplication. Here, the implementation is hardcoded to a 4-bit window. This was simply the first implemented approach without much reasoning. It remains future work to test the performance of a larger or smaller window.

Combined Scalar Multiplication and Addition

The combined multiplication of points and scalars with subsequent addition (i.e. $R = aP + bQ$) is a performance optimization for this selected kind of calculation.

```
4b—combined(result, P, Q, a, b):
  res = 0
  bit_list_a[max_scalar_length] = [0,...,0]
  bit_list_b[max_scalar_length] = [0,...,0]
  table_P = calculate_table(P)
  table_Q = calculate_table(Q)
  bit_position = max_scalar_length-1
  bit_list = bits_in_scalar_lsb_to_msb(scalar)
  for bit in bit_list:
      res = double_point(res)
      if bit_position % 4 == 0:
        //interpret bits as integer e.g. 0101 = 5, 1111 = 15
        w = bit_list_a[bit_position+3 : bit_position]
        res = add_points(res,table_a[w])
        w = bit_list_b[bit_position+3 : bit_position]
        res = add_points(res,table_b[w])
      bit_position = bit_position-1
  return res
```

Listing 4.3: Pseudocode for $aP + bQ$

As visible in Listing 4.3, the approach combines the addition with the multiplication. It is represented with the 4-bit window approach, but it can just as well be implemented with the daa approach. The amount of additions is the same as would be needed for two sequential

scalar additions. Doublings of the result, however, are only done half as many times as would be needed sequentially. This comes at the expense of memory, however, as now three points must be held in memory at a time: the two points to multiply P, Q and the resulting point as well. The same is true for the scalar value. If the underlying approach is daa, this expense comes in the form of extra register IDs and the extra point residing in the PKA SRAM. With the 4-bit window approach, both tables must be accessible at the same time. In the current implementation this means that both tables allocate space on the stack.

Point Compression and Decompression

To make use of the elliptic curve arithmetic with points given in compressed form, the point compression and decompression as defined in RFC 8032 (5.1.2, 5.1.3) [50] had to be implemented as well. The implementation in the CC-312 driver [83] as well as the explanations of the single steps defined in the RFC of [92] were employed during the implementation. It restrains the possible compression and decompression to points for edwards25519. The implementation will need to be modified to support the larger curve edwards448 (see Section 2 of RFC 8032 and RFC 7748 [56]). This functionality is necessary to use the cc3xx driver in the same way as the c25519 library for the software-based implementation. It should also be mentioned that this is functionality additional to that of the interface defined in the driver and must thus be used directly and not via the interface.

Testing

All arithmetic operations were tested for correct results. Test vectors were created by use of existing libraries for elliptic curve cryptography as introduced in Section 4.2.4. Point compression and decompression was tested as well. Test points were generated manually and obtained from the test cases for the CBS implementation. The tests were integrated in the existing test structure of the cc3xx driver.

4.3.3 Integration in Wallet Application

Integration of the hardware- and software-based versions was done as described in the design section (see Section 4.2). The CBS implementation was added to the `taler_crypto_ll` module and tests for the CBS scheme were added as unit tests in the existing structure. Per default, the hardware-accelerated approach uses the 4-bit window approach with pregenerated table for the generator point for general scalar multiplication as well as the combined multiplication with subsequent addition. The application already had a build-time decision mechanism for the hardware-accelerated or software-based application of the RSA implementation. For the CBS implementation the same was used to ensure inclusion of the necessary modules for each approach. This implies that the hardware-accelerated part of the new implementation is bound to the nRF52840DK as only useable board. It should further be mentioned, that the implementation partially makes use of the Edwards curve implementation directly and does not adhere to the elliptic curve interface defined in the cc3xx driver. Completely adhering to the interface remains future work.

The CBS implementation was not integrated in the taler-riot demo application and can not be used for actual withdrawals from an exchange. Implementing this needs changes at least in the `taler_crypto`, `taler_data` and `taler_webapi` modules and remains future work. However, the implementation can be used to test the blinding primitives blind, unblind and verify and compare them to the respective RSA primitives. For the evaluation an extra branch was created that allows the use of additional environment parameters to influence the build process. Details can be found in the next chapter.

5 Evaluation

The evaluation will look at different aspects of the implementation. The setup used to take measurements will be introduced first. Next, the single improvements for the scalar multiplication described in the Section 4.3.2 will be compared. Afterwards, the three blind signature primitives blind, unblind and verify will be compared between the two approaches CBS and RSA. This comparison includes the runtimes of the single primitives, the memory footprint and the security levels of both approaches.

5.1 Setup

The measurement application [89] is build on top of the benchmark application that was already used for [44]. Just as the taler-riot application it is written for the RIOT operating system and located in the subdirectory `app/bench_crypto_ll` of the repository. All measurements are done by flashing the application onto an nRF52840DK board that is connected to a computer via USB. Outputs are printed to the command line of the computer via a serial connection.

The benchmark application contains functions to benchmark the withdrawal process with the RSA approach. This is used as is without further changes. The hardware-acceleration backend is the cc3xx driver of TF-M, the software backend is the RELIC library [67]. For CBS the implementation described in Chapter 4 is used. It also uses the cc3xx driver, with the added functionality for twisted Edwards curves that was introduced in Section 4.3.2 [86]. For the software-based implementation the library c25519 [11] is used. The elliptic curve that is used is the twisted Edwards curve version of Curve25519 (edwards25519) [16]. The operations under test are the primitives of the blind signature schemes executed on the wallet side: blind, unblind and verify. Measurements include execution times of the primitives, the maximum stack allocations and the memory segment (text, data, bss) sizes.

The stack allocation is measured by use of the `thread_measure_stack_free` function of RIOT [74]. This function delivers the difference between the total available stack size and the maximum amount of stack space used so far. For example, let the stack have a size of 1024 Byte and three operations are called in order. Operation one allocates 100 Byte of memory on the stack and returns, Operation two allocates 200 Byte of memory and returns and lastly,

Operation three allocates 50 Byte of memory and returns. Assume that no stack space was used before operation one was executed. Hence, a call to `thread_measure_stack_free` would return 1024 B before any operation would have taken place. After operation one, the return value would be 924 B (1024-100), after operation two 824 (1024-200). When called after operation three, however, it would also return 824 B because the operation only checks the highest amount of memory used so far. In other words, invoked at the end of a program execution, it returns the peak stack usage during the programs runtime. It is used here to get exactly that information if not noted otherwise.

Measuring the sizes of the single segments is done by use of the tool *cosy* [70] which is integrated in the RIOT build system and can be called as a build target. It delivers the composition of text, data and bss segments as well as more detailed drill-down options to get sizes of single objects in the memory segments. Here, detailed information is only presented when necessary.

Lastly, measuring the execution time is done in loops of ten iterations for each of the operations. A RIOT `ztimer_stopwatch_t` stopwatch is used to measure the execution time of the whole loop execution. At the end, the execution time is divided by the amount of iterations to get the average execution time.

The collected data presented below can be found in the GitHub repository [89] that also contains the application in the subdirectory `app/bench_crypto_ll/evaluation_thesis`.

5.2 Scalar Multiplication Approaches

In Chapter 4 multiple versions of the scalar multiplication operation were discussed. To evaluate the impact and performance of the single improvements, the CBS blind, unblind and verify primitives are executed with different configurations and the results are compared. Table 5.1 summarizes the six different settings.

	double-and-add-always	4-bit window	
aX+bY		pregenerated	not pregenerated
serial	daa-serial	4b-serial-pg	4b-serial
combined	daa-combined	4b-combined-pg	4b-combined

Table 5.1: Table summarizing the differentiated settings. The names in the single cells are used in the figures further below and in the appendix.

The foundational setting is how the multiplication is executed: using the 4-bit window (4b) approach or the double-and-add-always (daa) approach. Executing the multiplication of two different points with two different scalars ($aX + bY$), saves operations but effectively uses the same multiplication approach. Lastly, for the 4-bit window approach the table for the generator point can be either pregenerated and hardcoded or generated when needed. This setting does not affect the double-and-add-always method. It also does not affect multiplications with points other than the generator point. The application used for evaluation (see Section 5.1) must be recompiled to enable or disable the respective settings. Further information about that can be found in [89].

Before discussing the single settings, it should be made clear what the impact on the single blinding primitives is. Equations 5.1 – 5.4 reiterate the single operations done when each primitive is executed (see also Figure 2.9). 5.1 and 5.4 are expected to be generally impacted by the change of the multiplication approach from double-and-add-always to 4-bit window as they contain scalar multiplications. In case of the 4-bit window approach both can be expected to be effected by the use of the pregenerated table as both contain multiplications with the generator point. Making use of the combined approach for multiplications and addition can only impact 5.1 and thus the blinding process, as this is the only equation where the pattern $aX + bY$ exists. 5.2 and 5.3 are expected to be unaffected by the settings as the underlying addition method is never changed.

blind:

$$R' = R + aG + bD_p \tag{5.1}$$

$$c = c' + b \tag{5.2}$$

unblind:

$$s' = s + a \tag{5.3}$$

verify:

$$sG = R + cD_p \tag{5.4}$$

Figure 5.1 shows all measured runtimes and Figure 5.2 shows the memory footprint of all settings. The names in the figures match those used in Table 5.1. Discussion of the results will be done in order of the blinding execution speed. Note that the runtimes and memory footprint measurements for hardware- and software-based RSA blind signatures and the software-based CBS scheme are just included for reference and completeness. They are discussed in detail in Section 5.3. Also, since the unblind operation is not affected by the changes it will also not be discussed.

Blinding contains the operations 5.1 and 5.2 from Equation (5.4) as well as the full-domain-hash operation to calculate c' (see Figure 2.9). This last operation was not reimplemented. Instead, the existing RSA implementation was simply reused. Hence, this operation is completely uninfluenced by any changes in the scalar multiplication approaches. It does, however, consume some stack itself. In order to examine the stack usage of the single steps during the blinding operation, the steps to calculate R', c' and c were executed on single threads for each of the hardware-accelerated settings. Figure 5.3 gives more details about the stack usage of the single blinding step operations.

Double-and-add-Always Approaches

The configuration HW-CS-daa-serial takes the longest time to complete blinding and verification. It has a slightly smaller text segment compared to HW-CS-daa-combined and uses the same maximum stack space. The effective stack used for the calculation of R differs just by a few bytes from that of HW-CS-daa-combined. This is understandable since the combined approach effectively just takes another point into the calculation. The data for

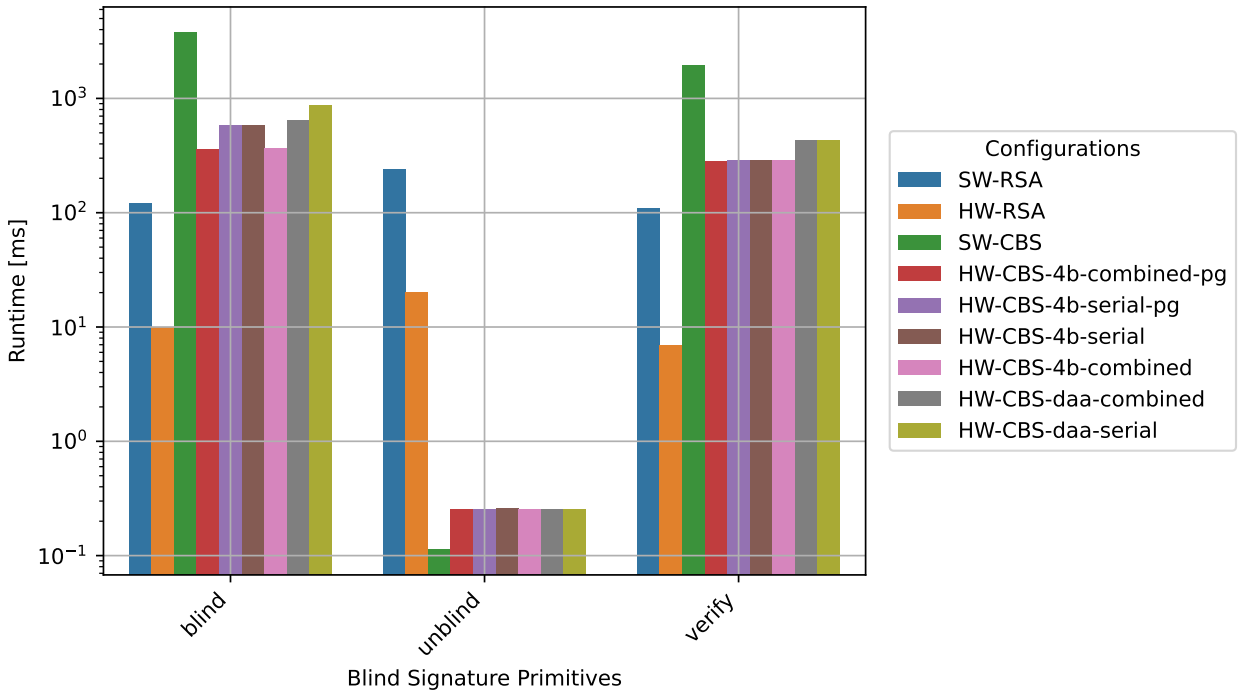

Figure 5.1: Runtimes of primitives in all configurations.

that point is in the PKA SRAM and only an additional pointer is held on stack. Thus, the difference is small. As to be expected, the HW-CS-daa-combined blinding runtime is faster while the verification runtime equals that of the serial setting. This is to be expected since the number of doublings needed during blinding is halved.

4-bit Window Approaches

Changing from the double-and-add-always approach to the 4-bit window approach shows a gain in speed for blinding and verification. This is to be expected, since the amount of additions per multiplication decreases. Looking at Figure 5.3, it becomes visible that the stack used for the calculation of R' grows significantly with the 4-bit approach. This is expected since a table of 2048 Byte is created for that purpose. In case of the combined settings two tables are held, and thus those settings have the highest stack requirement. If the generator table is pregenerated, it is visible as a 2kB increase in the text segment. All other settings do not influence the static memory in a major way. In terms of execution speed, the table pregeneration does not seem to impact neither the blinding nor the verification step significantly. Hence, this improvement offers no significant benefit. For the verification, the major decrease in execution time is visible when using any of the 4-bit window approaches. Looking at the blinding operation it becomes visible that the settings using the combined multiplication and add configuration are faster than the serial ones. This matches the

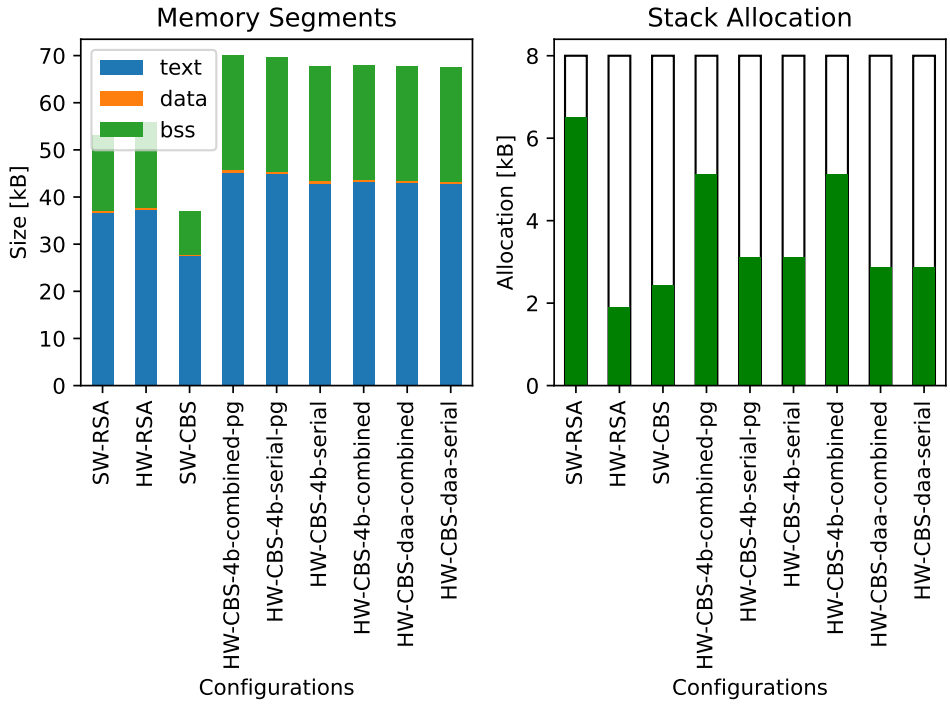

Figure 5.2: Memory footprint for all configurations. The transparent bars in the stack usage diagram symbolize the maximum available stack size of 8kB. The stack allocation has the same color as the bss segment since the stack is part of bss in RIOT.

expectations, since the calculation of R' needs that exact operation.

The measurements show that the use of the 4-bit window approach offers a visible increase in calculation speed of a scalar multiplication opposed to the double-and-add-always approach. Further, it is visible that the combined approach of two scalar multiplications followed by addition of the results speeds up operations. Lastly, pregenerating the 4-bit window table for the generator point does not lower the execution time significantly. Given those findings the default setting for the CBS approach is the 4-bit window approach with combined multiplications and additions as well as the pregenerated generator table.

The execution times can be further enhanced by exploring more involved scalar multiplication approaches. Also, the implementation as it is now may make better use of the PKA SRAM by storing the 4-bit window table in there, if possible. Other underlying addition and doubling operations could be evaluated as well.

5.3 Comparing RSA and CBS

The two blind signature schemes shall be compared to show which benefits each of them offers. The argument to investigate the CBS approach is to have a potential alternative to the RSA blind signatures which are already implemented. The different approaches are

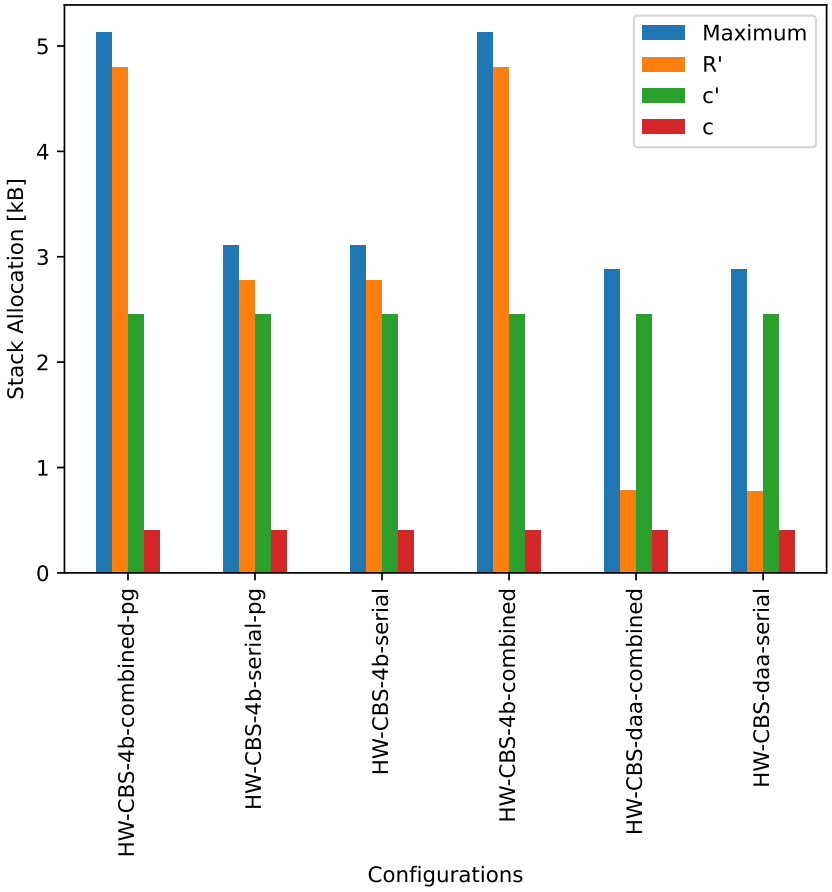

Figure 5.3: Stack allocations of the single steps in the blind operation. R', c', c mean the maximum stack allocation for the individual calculation of each of those parameters. "Maximum" shows the maximum stack allocation for the complete blind operation. This is higher because it measures the complete blind operation call stack.

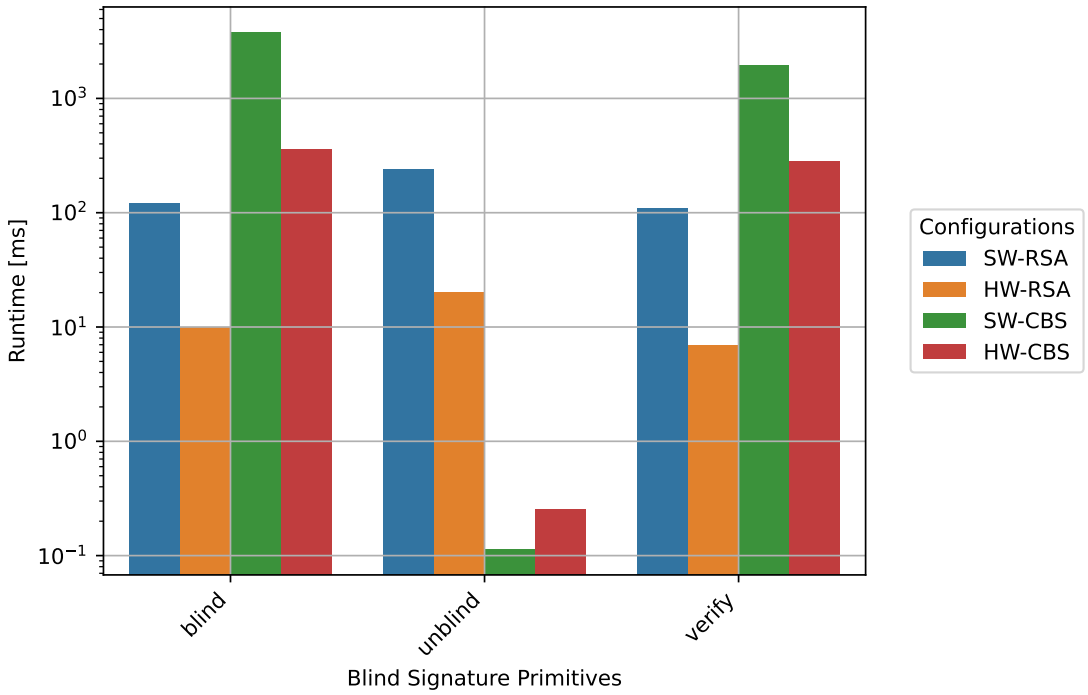

Figure 5.4: Execution times of the single blind signature primitives for CBS and RSA with the software-based (SW) and hardware-accelerated (HW) approaches.

named HW-RSA/SW-RSA for the hardware-accelerated/software-based RSA approach and analogously HW-CBS/SW-CBS for the CBS approach. The HW-CBS configuration is that of the 4-bit window with pregenerated table and combined multiplications with subsequent addition. Note also that the exact RSA blind signature scheme applied here is RSA-FDH with 2048-bit keys. This is abbreviated to just RSA as long as it is not mentioned otherwise in the text.

5.3.1 Time Measurements

Figure 5.4 shows the times needed for the primitives blind, unblind and verify for both schemes. The single operations will be discussed one after another.

blind

As visible, RSA is faster than CBS by one or two orders of magnitude, in the hardware-accelerated and the software-based setting, respectively. For both schemes the hardware-accelerated version is faster than the software-based. To put this finding in perspective, it can be compared with the findings of [29]. Their performance comparison (Section 6.3) highlights the comparisons of CBS with RSA-1024 and RSA-3072 (where 1024 and 3072 are the bit lengths of the RSA keys) and finds that blinding is quicker with RSA-1024. The measurement for this

finding was done on a notebook computer with more powerful hardware than used here. That is also the case for the other systems listed in their appendix. In there, 10 different datasets measured with different hardware setups can be found. What remains to be true for all their evaluated systems, however, is that RSA-1024 is faster than the CBS approach. In some cases (Tables B.3, B.4) CBS is even slower than RSA-2048 and in one case even slower than RSA-4096 (Table B.2) Their slowest measurement (on a system with a 4-core risc64 processor) for blinding (Table B.8) takes 72ms for 10 blindings. Together with the 2.514 ms for the derivation of the blinding secrets, the average time for one blinding operation is ∼7.45 ms. Here, the fastest time for a hardware-accelerated blinding process is around 360ms, so about a factor of ~ 48 slower. The RSA-2048 approach listed in Table B.8 takes 3.7ms for a single blind. Here, the hardware-accelerated RSA approach takes about 9ms which makes a difference factor ~ 2.4. Potential reasons for those different results are given after the verification operation is evaluated as they are similar for both cases.

unblind

Looking at the unblind operation, the findings are in line with those found in [29]. The CBS approach is faster than the RSA approach both in the hardware- and software-based implementation. Both CBS implementations have an execution time below 1ms, while the RSA implementations are above 10ms for the hardware-accelerated implementation and above 100ms for the software-based implementation. As mentioned in 6.3 of [29], unblinding is the expensive operation of the RSA blind signature scheme as it involves an inversion. In CBS, on the other hand, unblinding is a cheap modular addition. Notably, the software-based approach of CBS is the fastest of all four. The hardware-accelerated version enables and disables the crypto-peripheral each time it is executed in the loop. Measuring this time as part of an unblind operation seems reasonable as the CC-310 would also only be enabled when needed, if the operation would be a part of a complete application. The software-based implementation does not have that initialization phase. If one would want to investigate this further, the time needed for the initialization would need to be measured. This time can potentially be decreased by modifying the function to enable the PKA. In the current implementation that is done via the elliptic curve interface for all CBS related operations. As the unblind primitive just needs modular arithmetic, it would suffice to enable the PKA as such. Improving the implementation remains future work.

verify

For verification, the situation is similar to that of the blinding primitive. The RSA approach is faster than CBS in hardware as well as in software. The time needed for the operation is less than that needed for blinding in all settings. For CBS that is understandable, as the amount of operations needed for the verification primitive is less than that needed for the blinding primitive. For RSA, the verification also involves fewer operations and is thus understandably faster as well. Compared to the unblind primitve, it is faster for RSA and notably slower for CBS. This can be explained when comparing the involved operations. Those findings are again not in line with those of [29]. Only one of their measurements (Table B.2) has the same

result of CBS being slower than RSA-2048. In all other case the verification is faster for CBS than for RSA-2048. Looking again at the slowest of their measurements (Table B.8), CBS is slower than RSA-1024 but almost double as fast as RSA-2048 (CBS 2.9ms vs. RSA-2048 5.4ms). This is ∼98 times faster than the fasters CBS verification here (∼285ms).

Potential reasons for the different results between this work and [29] are twofold. The first and obvious point is that the hardware used for their comparisons is significantly more powerful than the microcontroller used here. The second difference is the used software. Here, the solution for hardware-accelerated elliptic curve cryptography is partially self developed throughout the thesis. The solution for software-based cryptography on the other hand is well known. On the homepage [11], the author mentions the small size and thus appropriateness for microcontrollers but not speed. In [29] the libraries libsodium [57] and libgcrypt[65] are used. Both libraries are well known and tested. There is a high probability that the software for the hardware-accelerated implementation can be improved significantly when more time is invested. For the software-based solution one could employ a library that is more focused on speed then on size, in order to find other values for comparison. Monocypher [61] would be one example.

5.3.2 Memory Footprint

Figure 5.5 shows the memory segment sizes as well as the stack usage of all four versions of CBS and RSA.

Memory Segments

HW-CBS has the largest sizes over all segments. A meaningful comparison can be made with the HW-RSA as both share the same CC-310 driver. The CBS implementation contains not just the objects for the PKA engine, as RSA-HW does, but also those for the elliptic curve cryptography. 2kB of the text segment are alone the pregenerated table for the generator point, used in the 4-bit window approach for scalar multiplication. This explains a larger text segment for the CBS-HW approach. Section 5.2 handles the implications of the different hardware-accelerated scalar multiplication approaches on memory in more detail. The two software-based approaches differ slightly in terms of the text segment but SW-RSA has a larger BSS segment. The size of the two used crypto libraries differs by about 5 kB, with 10.4 kB for relic and 4.5 kB for c25519. As already mentioned in the comparison of the blinding operation, c25519 is designed to be small. No further research was done on how RELIC implements its RSA operations. Hence, no meaningful argument can be given about whether its size could be lowered by some setting. Overall, the two RSA approaches differ mainly in terms of their bss segment size. The remaining sizes are quite similar. For CBS, the different approaches are sized very differently in all segments. The CBS-SW implementation has the smallest binary size, CBS-HW the largest.

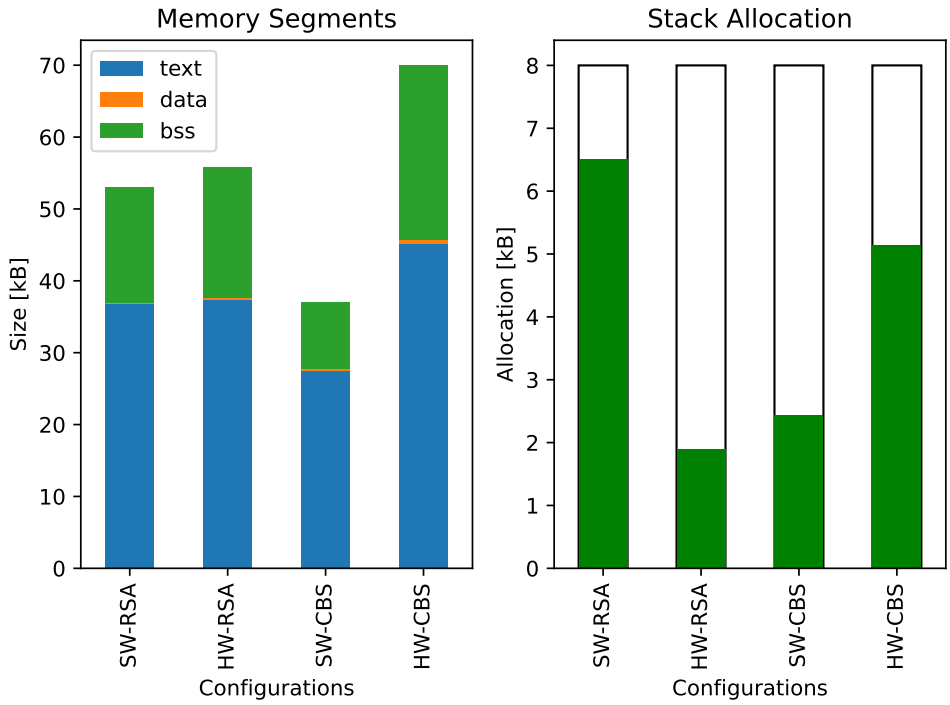

Figure 5.5: Memory segment sizes and stack usage of the hardware-accelerated (HW) and software-based (SW) implementations of CBS and RSA. The stack allocation has the same color as the bss segment since the stack is part of bss in RIOT.

Signature Scheme	Size per Coin	Size per Signature
RSA-FDH-2048	276 Byte	256 Byte
CBS	160 Byte	64 Byte

Table 5.2: Storage sizes of coins and signatures with the RSA and CBS schemes.

Stack Usage

The benchmark application does not use any form of `malloc`, hence only the stack allocation is examined. For the hardware-accelerated implementations, the PKA-SRAM is used to store and use the operators for the single calculations. However, the allocation of the SRAM was not measured. It would not be possible to use it for other data and also a measurement operation would need to be implemented as there is none available. The scalar multiplication using the 4-bit window approach with combined multiplication and addition for HW-CS explains the higher stack usage compared to that of HW-RSA. The combined multiplication and addition approach creates two tables (each 2 kB) in memory. HW-RSA on the other hand only holds pointers to large numbers stored in the PKA SRAM (see Section 4.3.2). Those are necessary for CS-HW as well as it also uses the PKA. The two software-based implementations differ significantly in terms of stack usage. This is plausible as the RSA-SW approach needs to store the 2048-bit RSA keys in memory while the CS-SW approach just needs to store 256-bit elliptic curve keys.

Storage and Transmission

The chosen blind signature approach has a direct effect on the storage space needed for a single coin as well as the data that must be transmitted to receive a blind signature for it. This is relevant in terms of the amount of coins that can be stored in a limited amount of storage and also in terms of the bandwidth needed to create coins. The calculations in this section are purely theoretical and not measured. Generally, this topic is largely handled in parallel to [29] as their design was simply adopted.

Table 5.2 shows an overview of the sizes of coins and signatures. A coin consists of a Ed25519 Keypair c_s, C_p, the unblinded signature σ_c and the denomination key of the exchange D_p. The size of the coin key pair is 512 bit (64 Byte) both for RSA and CBS. This can be halved if only the private key is kept. As long as the curve is fixed, the generator is also fixed and the public key can be easily recalculated instead of stored. What differs between the two schemes is the size of a signature. In the case examined here, a RSA-FDH 2048-bit signature has a size of 2048 bit (256 Byte), while a CBS signature is only 512 bit (64 Byte) large. The denomination key D_p has a size of 256-bit for CBS and 2048-bit for RSA. Hence, a coin created with the CBS scheme has a size of 116 Byte ($\sim$42%) less than a coin created with the RSA-FDH-2048 blind signature scheme. In Chapter 6.3 of [29], calculations for different RSA sizes can be found as well. Note, however, that the calculation there makes different assumptions about the parts that make up a coin, so the numbers differ slightly. The major finding, that CBS needs less space than RSA, is the same, however.

The necessary communication for the two schemes differs by one round trip time. That means that a user needs to send two messages to the server and wait for two answers before a withdrawal can be completed, while for RSA a single message and response suffice. For RSA the user sends the blinded public key of his coin (2048 bit) and directly receives a signature from the exchange server (2048 bit). For CBS, the user first has to initialize the blinding process by sending the withdrawal nonce (256 bit). He then receives the R values ($2 \cdot 256$ bit) needed for blinding. Once blinding is done, he again contacts the exchange, sending the generated c values ($2 \cdot 256$ bit) and receives the signature (256 bit) together with the decision (1 bit) about which of the runs is finished by the exchange. During those two roundtrips, the amount of sent data is less than what is sent in the single roundtrip of the RSA case. Table 5.3 gives a summary of the transmitted amount of data and the roundtrips.

Signature Scheme	Roundtrips	Transmitted Data
RSA-FDH-2048	1	4096 bit
CBS	2	1281 bit

Table 5.3: Transmitted data and roundtrips for the CBS and RSA-FDH scheme.

5.3.3 Security Level

Both blind signature schemes must be secure in order to be used in a live payment system like GNU Taler. As mentioned in Section 2.3.1, a minimum key length of 3072 bits is advised for newly developed systems in [49]. In 7.9 of [63] from 2024 it is mentioned that key lengths between 2048 and 4096 bit are recommended. Considering the second advice, the RSA key length used here is at the lower end of the range. Table 2 in Section 5.6.1.1 of [10] gives an overview of the security strenght of different asymmetric-key algorithms. It lists RSA-2048 with a security strength of 112 bits. edwards25519 is a 256-bit curve and thus achieves a security level of 128 bits. However, this value does not say anything about the CBS scheme implemented and discussed here. As discussed in Section 3.1, the security level of the CBS scheme with the specific curve used here, is considered to be 70-bit in the worst case. This lower security level is based on the fact that Clause Blind Schnorr signature do not just rely on the ECDLP but also on the mROS problem. That means, that the CBS scheme delivers a lower security level than the existing RSA approach. As mentioned in [39], one could increase the amount of parallel runs in one blinding process to achieve a higher security level, but it is also mentioned that this would increase the communication complexity very much. Another option, already mentioned in [29], is to use an elliptic curve with higher security level like Curve448 [45].

This opens up the second aspect of this comparison. Neither of the implemented blind signature schemes is configurable in terms of its security level. The implementation for CBS is hardcoded for curve edwards25519 and RSA is fixed to 2048-bit keys. As this wallet version for low-end IoT systems is a special implementation it should be mentioned that the main wallet implementation for phones and desktop computers offers flexibility in the RSA key size. Since

the curve is also fixed in the exchange implementation, the main wallet would, if CBS would be implemented there, also have no other choice for curves. As already mentioned in [29], this fixed curve is known and making it flexible is considered as possible future work. The same can be said for the IoT version of the wallet. Lastly, an aspect regarding the implementation of CBS that is evaluated here, is the low-level hardware-accelerated implementation of the elliptic curve operations. It should be kept in mind that it was implemented during the implementation phase of this thesis. While side-channel attack resistance was considered and correctness of the arithmetic operations was tested, the probability that there is some flaw in the implementation should be taken into account.

6 Conclusion

This chapter contains a summary over the findings of this thesis and an outlook on options for future work.

6.1 Summary

This thesis investigated the use of Clause Blind Schnorr (CBS) signatures on constrained systems. It has given an overview over the signature scheme and introduced its use in the context of the payment system GNU Taler. An implementation of CBS was created within a wallet application for the payment system which was developed for the RIOT operating system on constrained systems. [29] added the scheme to the payment system and the design developed there was adopted. The implementation contains a pure software-based version that uses the c25519 [11] library as well as a hardware-accelerated version. Hardware-acceleration is limited to an nRF52840 Development Kit and the ARM Cryptocell-310 peripheral available on that board. cc3xx, a part of the Trusted Firmware-M project, was used as driver for the CC-310 and had to be enhanced to support the twisted Edwards version of Curve25519 (edwards25519) and operations to perform arithmetic on that curve.

Different options in regard to scalar multiplications were pursued for the hardware-accelerated elliptic curve arithmetic. Evaluation of these options showed that the implemented 4-bit window approach offers a gain in performance for the blinding and verification primitives of the CBS scheme over the double-and-add-always approach. Precalculations for known points in the window scheme did not show significant gains in execution time. Blinding could further be enhanced by combining multiplications with subsequent addition.

The implementation of the CBS scheme was evaluated on the nRF52840DK board in regard to execution time, storage and transmission requirements as well as security-level. Execution times of the single blind signature primitives (blind, unblind, verify) were measured and compared to those of the already implemented RSA-FDH blind signature scheme with 2048-bit key size. The fastest times for hardware-accelerated CBS blinding ($\sim$361ms) and verification($\sim$284ms) were found to be higher than the times for the RSA-FDH based blinding ($\sim$10ms) and verification ($\sim$7ms) by one or two orders of magnitude respectively. Unblinding ($\sim$0.1ms) was fastest with the software-based CBS approach and with that two orders

of magnitude faster than the fastest measured RSA-FDH unblinding procedure (∼20ms). This was the only operation that was found to be faster in the software-based than in the hardware-accelerated implementation.

The measurements were further compared to the findings of [29]. In general, their measurements showed that, compared to 2048-bit RSA-FDH blind signatures, most of their tested systems were faster with the CBS approach for all blind signature primitives. Comparison of their slowest results (measured on a 4-core risc64 CPU) to the findings of this thesis showed that blinding (7.45ms) is ∼48 times faster and verification (2.9ms) ∼98 times faster than on the nRF52840DK board. For unblinding the common finding was that it was faster for CBS than for RSA-FDH.

In terms of storage and transmission measurements the findings of [29] were confirmed. Blind signatures based on the CBS scheme are smaller and need less bandwidth. However, for creation they need a second roundtrip which is not needed for RSA-FDH. This translates to a larger amount of coins that could be stored in a Taler wallet with a fixed amount of storage.

Regarding the security level of both approaches, it was noted that CBS offers only 70-bit security in the worst case, while for RSA-FDH a security level of 128-bit is assumed. In the examined system neither of the approaches had a configurable security level. The overall Taler system allows different RSA key sizes but the CBS scheme is fixed to the edwards25519 curve that was used here as well. Changing to a curve with a higher security level like Curve448 would increase the security level of the scheme.

To summarize, the thesis showed that the CBS scheme can be implemented on constrained systems. Making use of cryptographic accelerators was distinctly more involved than using software-based solutions but offered benefits in performance.

6.2 Future Work

Multiple options for future enhancements exist. The cc3xx driver can be further optimized, the integration of the CBS implementation in the taler-riot application can be further pursued and the GNU Taler system as such can be made more flexible in terms of curve choice for the CBS scheme.

The cc3xx driver as the basis for the hardware-acceleration can be enhanced by implementation of faster scalar multiplication approaches, investigations about faster addition and doubling formulas and better use of the PKA SRAM. Further, the existing implementation can be completed to be fully compatible to the existing elliptic curve interface. The implementation of Curve448 would be just an addition of parameters if twisted Edwards curves are implemented without limitations. Also, with the added support for edwards25519, the Ed25519 signature system could be incorporated into the driver.

Full integration of the CBS scheme in the taler-riot application remains an open task. If that is to be completed, the withdrawal protocol unit remains to be adjusted to the new scheme as well as other involved modules. The solution as it is right now also needs more adjustment to fully use the elliptic curve framework of cc3xx. To remain independent of the hardware-accelerator, different cryptographic software libraries can be tested to get better performance with them.

Lastly, with regard to the security level of the blind signature approach, the GNU Taler system could be modified to support CBS with different curves. Especially curves with higher security levels like Curve448. This would impact not only the wallet implementation but also the exchange software. Other than that, the research regarding blind signatures on standardized elliptic curves appears to be very active and could be monitored to find more suitable blind signature solutions for GNU Taler.

List of Abbreviations

CBS Clause Blind Schnorr

CC-310 ARM Cryptocell-310

CC-312 ARM Cryptocell-312

daa double-and-add-always

EC elliptic curve

EdDSA Edwards-Curve Digital Signature Algorithm

edwards25519 twisted Edwards version of Curve25519

EFD Explicit-Formulas Database

FDH Full Domain Hash

HKDF HMAC-based key derivation function

IoT Internet of Things

nRF52840DK nRF52840 Development Kit

OS operating system

psa-crypto PSA Certified Crypto API

Taler GNU Taler

taler-riot Taler on RIOT

TF-M Trusted Firmware-M

List of Figures

2.1 Concept of digital signatures 9
2.2 Blind Signatures 10
2.3 RSA Full Domain Hash Signature scheme 12
2.4 RSA-FDH Blind Signature Scheme 14
2.5 Schnorr Signature Scheme on Elliptic Curves 16
2.6 Blind Schnorr Signature Scheme 18
2.7 Clause Blind Schnorr Signature Scheme 19
2.8 Taler Overview 24
2.9 The Clause Blind Schnorr Withdrawal Process 27

4.1 Taler-RIOT Architecture Overview 32
4.2 Simplified Withdrawal with Participants 33
4.3 Abstract Scalar Multiplication Example 38

5.1 Runtimes of primitives in all configurations. 48
5.2 Memory footprints for all configurations 49
5.3 Stack allocations per configuration for blinding step 50
5.4 Execution Times of CBS and RSA primitives 51
5.5 Memory Footprint for RSA and CBS 54

List of Tables

2.1 Additive and multiplicative groups 4
2.2 EdDSA Parameters 20

4.1 Necessary Withdrawal Implementations 34

5.1 Summary of scalar multiplication configurations 46
5.2 Storage sizes of coins and signatures with the RSA and CBS schemes. 55
5.3 Transmitted data and roundtrips for the CBS and RSA-FDH scheme. 56

Bibliography

[1] ARM. *ARM-software/psa-api*. Nov. 26, 2025. URL: https://github.com/ARM-software/psa-api (visited on 11/26/2025).

[2] ARM. *PSA Certified Crypto API*. PSA Certified APIs. URL: https://arm-software.github.io/psa-api/crypto/ (visited on 11/26/2025).

[3] Nordic Semiconductor ASA. *CRYPTOCELL — Arm TrustZone CryptoCell 310*. CRYPTOCELL — Arm TrustZone CryptoCell 310. URL: https://docs.nordicsemi.com/bundle/ps_nrf52840/page/cryptocell.html (visited on 11/28/2025).

[4] Nordic Semiconductor ASA. *CRYPTOCELL — Arm TrustZone CryptoCell 312*. CRYPTOCELL — Arm TrustZone CryptoCell 312. URL: https://docs.nordicsemi.com/bundle/ps_nrf5340/page/cryptocell.html (visited on 11/28/2025).

[5] Nordic Semiconductor ASA. *nRF5 SDK v17.1.0: CryptoCell EC Edwards APIs*. URL: https://docs.nordicsemi.com/bundle/sdk_nrf5_v17.1.0/page/group_crys_ec_edw.html (visited on 11/26/2025).

[6] Nordic Semiconductor ASA. *nRF5 SDK v17.1.0: Cryptography API*. URL: https://docs.nordicsemi.com/bundle/sdk_nrf5_v17.1.0/page/group_crypto_api.html (visited on 11/26/2025).

[7] Nordic Semiconductor ASA. *nRF52840 DK*. nRF52840 DK. URL: https://www.nordicsemi.com/Products/Development-hardware/nRF52840-DK (visited on 11/28/2025).

[8] Nordic Semiconductor ASA. *nRF52840 Product Specification*. nRF52840 Product Specification. URL: https://docs.nordicsemi.com/bundle/ps_nrf52840/page/keyfeatures_html5.html (visited on 12/01/2025).

[9] Emmanuel Baccelli et al. "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT". In: *IEEE Internet of Things Journal* 5.6 (Dec. 2018), pp. 4428–4440. URL: https://ieeexplore.ieee.org/document/8315125 (visited on 12/02/2025).

[10] Elaine Barker. *Recommendation for Key Management Part 1: General*. NIST SP 800-57pt1r4. National Institute of Standards and Technology, Jan. 2016, NIST SP 800–57pt1r4. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf (visited on 05/26/2025).

[11] Daniel Beer. *Curve25519 and Ed25519 for low-memory systems*. URL: https://www.dlbeer.co.nz/oss/c25519.html (visited on 11/17/2025).

[12] Mihir Bellare and Phillip Rogaway. "Random oracles are practical: a paradigm for designing efficient protocols". In: *Proceedings of the 1st ACM conference on Computer and communications security*. CCS '93. New York, NY, USA: Association for Computing Machinery, Dec. 1, 1993, pp. 62–73. URL: https://dl.acm.org/doi/10.1145/168588.168596 (visited on 12/01/2025).

[13] Fabrice Benhamouda et al. "On the (in)Security of ROS". In: *Journal of Cryptology* 35.4 (Sept. 15, 2022), p. 25. URL: https://doi.org/10.1007/s00145-022-09436-0 (visited on 10/27/2025).

[14] Daniel J Bernstein and Tanja Lange. *Explicit-Formulas Database*. Explicit-Formulas Database. URL: https://hyperelliptic.org/EFD/ (visited on 11/27/2025).

[15] Daniel J. Bernstein and Tanja Lange. "Analysis and optimization of elliptic-curve single-scalar multiplication". In: *Contemporary Mathematics*. Ed. by Gary L. Mullen, Daniel Panario, and Igor E. Shparlinski. Vol. 461. Providence, Rhode Island: American Mathematical Society, 2008, pp. 1–19. URL: http://www.ams.org/conm/461/ (visited on 11/28/2025).

[16] Daniel J. Bernstein et al. "High-speed high-security signatures". In: *Journal of Cryptographic Engineering* 2.2 (Sept. 2012), pp. 77–89. URL: http://link.springer.com/10.1007/s13389-012-0027-1 (visited on 11/30/2025).

[17] Daniel J. Bernstein et al. "Twisted Edwards Curves". In: *Progress in Cryptology – AFRICACRYPT 2008*. Ed. by Serge Vaudenay. Vol. 5023. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 389–405. URL: http://link.springer.com/10.1007/978-3-540-68164-9_26 (visited on 10/01/2025).

[18] Mojtaba Bisheh-Niasar, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. "Cryptographic Accelerators for Digital Signature Based on Ed25519". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 29.7 (July 2021), pp. 1297–1305. URL: https://ieeexplore.ieee.org/abstract/document/9437467 (visited on 10/29/2025).

[19] Carsten Bormann, Mehmet Ersue, and Ari Keränen. *Terminology for Constrained-Node Networks*. Request for Comments RFC 7228. Internet Engineering Task Force, May 2014. URL: https://datatracker.ietf.org/doc/rfc7228 (visited on 12/01/2025).

[20] Carsten Bormann et al. *Terminology for Constrained-Node Networks*. Internet Draft draft-ietf-iotops-7228bis-03. Internet Engineering Task Force, Nov. 4, 2025. URL: https://datatracker.ietf.org/doc/draft-ietf-iotops-7228bis (visited on 12/01/2025).

[21] cfrg. *Related work · Issue #193 · cfrg/draft-irtf-cfrg-blind-signatures*. GitHub. URL: https://github.com/cfrg/draft-irtf-cfrg-blind-signatures/issues/193 (visited on 12/02/2025).

[22] David Chaum. "Blind Signatures for Untraceable Payments". In: *Advances in Cryptology*. Ed. by David Chaum, Ronald L. Rivest, and Alan T. Sherman. Boston, MA: Springer US, 1983, pp. 199–203.

[23] David Chaum and Torben Pryds Pedersen. "Wallet Databases with Observers". In: *Advances in Cryptology — CRYPTO' 92*. Ed. by Ernest F. Brickell. Berlin, Heidelberg: Springer, 1993, pp. 89–105.

[24] Lily Chen et al. *Recommendations for Discrete Logarithm-based Cryptography: Elliptic Curve Domain Parameters*. NIST SP 800-186. Gaithersburg, MD: National Institute of Standards and Technology, Feb. 3, 2023, NIST SP 800–186. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-186.pdf (visited on 12/02/2025).

[25] Contiki-NG. *Contiki-NG*. Contiki-NG, the OS for Next Generation IoT Devices. URL: https://www.contiki-ng.org/ (visited on 12/02/2025).

[26] Douglas Crockford. *Base 32*. Base 32. URL: https://www.crockford.com/base32.html (visited on 11/27/2025).

[27] cry.college. *Cry.College - Modern Cryptography Lectures*. URL: https://cry.college/ (visited on 11/27/2025).

[28] "Curve25519: New Diffie-Hellman Speed Records". In: Daniel J. Bernstein. *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 207–228. URL: http://link.springer.com/10.1007/11745853_14 (visited on 07/17/2025).

[29] Gian Demarmels, Lucien Heuzeveldt, and Expert Elektronikingenieur HTL Daniel Voisard. "Adding Schnorr's Blind Signature in Taler". PhD thesis. 2022. URL: https://www.taler.net/papers/cs-thesis.pdf (visited on 05/30/2025).

[30] Frank Denis, Frederic Jacobs, and Christopher A. Wood. *RSA Blind Signatures*. Request for Comments RFC 9474. Internet Engineering Task Force, Oct. 2023. URL: https://datatracker.ietf.org/doc/draft-irtf-cfrg-rsa-blind-signatures/02/ (visited on 12/02/2025).

[31] Frank Denis, Frederic Jacobs, and Christopher A. Wood. *RSA Blind Signatures*. Request for Comments RFC 9474. Internet Engineering Task Force, Oct. 2023. URL: https://datatracker.ietf.org/doc/rfc9474 (visited on 12/02/2025).

[32] Frank Denis, Frederic Jacobs, and Christopher A. Wood. *RSA Blind Signatures*. Request for Comments RFC 9474. Internet Engineering Task Force, Oct. 2023. URL: https://datatracker.ietf.org/doc/draft-irtf-cfrg-rsa-blind-signatures/07 (visited on 12/02/2025).

[33] W. Diffie and M. Hellman. "New directions in cryptography". In: *IEEE Transactions on Information Theory* 22.6 (Nov. 1976), pp. 644–654. URL: https://ieeexplore.ieee.org/document/1055638/ (visited on 11/04/2025).

[34] Florian Dold. "The GNU Taler system: practical and provably secure electronic payments". PhD thesis. Université de Rennes, 2019.

[35] GNUnet e.V. and Taler Systems SA. *1.2. Conventions for Taler RESTful APIs — GNU Taler*. 1.2. Conventions for Taler RESTful APIs - 1.2.5.16. Keys. URL: https://docs.taler.net/core/api-common.html#keys (visited on 11/27/2025).

[36] GNUnet e.V. and Taler Systems SA. *GNU Taler - Taxable Anonymous Libre Electronic Resources*. URL: https://www.taler.net/en/index.html (visited on 11/26/2025).

[37] GNUnet e.V. and Taler Systems SA. *Taler Overview Diagram*. URL: https://www.taler.net/images/TalerDiagram.svg (visited on 11/25/2025).

[38] Georg Fuchsbauer, Antoine Plouviez, and Yannick Seurin. "Blind Schnorr Signatures and Signed ElGamal Encryption in the Algebraic Group Model". In: *Advances in Cryptology – EUROCRYPT 2020*. Ed. by Anne Canteaut and Yuval Ishai. Vol. 12106. Cham: Springer International Publishing, 2020, pp. 63–95. URL: https://link.springer.com/10.1007/978-3-030-45724-2_3 (visited on 05/26/2025).

[39] Georg Fuchsbauer and Mathias Wolf. "Concurrently Secure Blind Schnorr Signatures". In: *Advances in Cryptology – EUROCRYPT 2024*. Ed. by Marc Joye and Gregor Leander. Cham: Springer Nature Switzerland, 2024, pp. 124–160.

[40] Oscar Garcia-Morchon, Sandeep Kumar, and Mohit Sethi. *Internet of Things (IoT) Security: State of the Art and Challenges*. Request for Comments RFC 8576. Internet Engineering Task Force, Apr. 2019. URL: https://datatracker.ietf.org/doc/rfc8576 (visited on 12/01/2025).

[41] GNUNET. *test_crypto_cs.c « util « lib « src - gnunet.git - Main GNUnet Logic*. GNUNET CS Test. URL: https://git.gnunet.org/gnunet.git/tree/src/lib/util/test_crypto_cs.c#n491 (visited on 11/19/2025).

[42] GNUnet. *GNUnet*. URL: https://www.gnunet.org/en/ (visited on 11/19/2025).

[43] Nils Gura et al. "Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs". In: *Cryptographic Hardware and Embedded Systems - CHES 2004*. Ed. by Marc Joye and Jean-Jacques Quisquater. Berlin, Heidelberg: Springer, 2004, pp. 119–132.

[44] Mikolai Gütschow and Matthias Wählisch. "Privacy-Preserving Payments on Constrained End-User Devices". In: *Proceedings of the ACM SIGCOMM 2025 Posters and Demos*. ACM SIGCOMM Posters and Demos '25. New York, NY, USA: Association for Computing Machinery, Sept. 10, 2025, pp. 152–154. URL: https://dl.acm.org/doi/10.1145/3744969.3748416 (visited on 10/30/2025).

[45] Mike Hamburg. *Ed448-Goldilocks, a new elliptic curve*. 2015. URL: https://eprint.iacr.org/2015/625 (visited on 11/24/2025).

[46] Darrel R. Hankerson, Alfred J. Menezes, and Scott A. Vanstone. *Guide to Elliptic Curve Cryptography*. 1st ed. 2004. Springer Professional Computing. New York, NY: Imprint: Springer, 2004. 1 p.

[47] Franklin Harding and Jiayu Xu. *Unforgeability of Blind Schnorr in the Limited Concurrency Setting*. 2024. URL: https://eprint.iacr.org/2024/1100 (visited on 10/24/2025).

[48] Huseyin Hisil et al. "Twisted Edwards Curves Revisited". In: *Advances in Cryptology - ASIACRYPT 2008*. Ed. by Josef Pieprzyk. Vol. 5350. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 326–343. URL: http://link.springer.com/10.1007/978-3-540-89255-7_20 (visited on 10/02/2025).

[49] German Federal Office for Information Security. *BSI – Technical Guideline Cryptographic Mechanisms: Recommendations and Key Lengths*. Jan. 31, 2025. URL: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf?__blob=publicationFile (visited on 08/07/2025).

[50] Simon Josefsson and Ilari Liusvaara. *Edwards-Curve Digital Signature Algorithm (EdDSA)*. Request for Comments RFC 8032. Internet Engineering Task Force, Jan. 2017. URL: https://datatracker.ietf.org/doc/rfc8032 (visited on 05/26/2025).

[51] Antoine Joux, Julian Loss, and Giacomo Santato. *Dimensional eROSion: Improving the ROS Attack with Decomposition in Higher Bases*. 2025. URL: https://eprint.iacr.org/2025/306 (visited on 10/27/2025).

[52] Peter Kietzmann et al. *A Performance Study of Crypto-Hardware in the Low-end IoT*. 2021. URL: https://eprint.iacr.org/2021/058 (visited on 12/02/2025).

[53] Martin Kleppmann. *Implementing Curve25519/X25519: A Tutorial on Elliptic Curve Cryptography*. URL: https://martin.kleppmann.com/papers/curve25519.pdf (visited on 07/23/2025).

[54] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-Hashing for Message Authentication*. Request for Comments RFC 2104. Internet Engineering Task Force, Feb. 1997. URL: https://datatracker.ietf.org/doc/rfc2104 (visited on 11/17/2025).

[55] Hugo Krawczyk and Pasi Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. Request for Comments RFC 5869. Internet Engineering Task Force, May 2010. URL: https://datatracker.ietf.org/doc/rfc5869 (visited on 11/17/2025).

[56] Adam Langley, Mike Hamburg, and Sean Turner. *Elliptic Curves for Security*. Request for Comments RFC 7748. Internet Engineering Task Force, Jan. 2016. URL: https://datatracker.ietf.org/doc/rfc7748 (visited on 11/28/2025).

[57] libsodium. *Introduction | Libsodium documentation*. Oct. 23, 2025. URL: https://libsodium.gitbook.io/doc (visited on 11/24/2025).

[58] Ilari Liusvaara. *CFRG Elliptic Curve Diffie-Hellman (ECDH) and Signatures in JSON Object Signing and Encryption (JOSE)*. Request for Comments RFC 8037. Internet Engineering Task Force, Jan. 2017. URL: https://datatracker.ietf.org/doc/rfc8037 (visited on 12/02/2025).

[59] Arm Ltd. *Arm CryptoCell-300 Machine Learning Processor*. CryptoCell-300 Family. URL: https://www.arm.com/products/silicon-ip-security/crypto-cell-300 (visited on 11/28/2025).

[60] Ken MacKay. *micro-ecc*. micro-ecc. URL: http://kmackay.ca/micro-ecc/ (visited on 12/02/2025).

[61] Monocypher. *Monocypher*. URL: https://monocypher.org/ (visited on 11/24/2025).

[62] National Institute of Standards and Technology (US). *Digital Signature Standard (DSS)*. NIST FIPS 186-5. Washington, D.C.: National Institute of Standards and Technology (U.S.), Feb. 3, 2023, NIST FIPS 186–5. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5.pdf (visited on 11/22/2025).

[63] Christof Paar, Jan Pelzl, and Tim Güneysu. *Understanding Cryptography: From Established Symmetric and Asymmetric Ciphers to Post-Quantum Algorithms*. Berlin, Heidelberg: Springer, 2024. URL: https://link.springer.com/10.1007/978-3-662-69007-9 (visited on 05/27/2025).

[64] Thomas Pornin. *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*. Request for Comments RFC 6979. Internet Engineering Task Force, Aug. 2013. URL: https://datatracker.ietf.org/doc/rfc6979 (visited on 12/02/2025).

[65] The GnuPG Project. *libgcrypt*. libgcrypt. Feb. 10, 2025. URL: https://www.gnupg.org/software/libgcrypt/index.html (visited on 11/24/2025).

[66] Zephyr® Project. *Zephyr Project*. Zephyr Project. URL: https://www.zephyrproject.org/ (visited on 12/02/2025).

[67] RELIC. *GitHub - relic-toolkit/relic: Code*. URL: https://github.com/relic-toolkit/relic (visited on 11/17/2025).

[68] RIOT. *ARM CryptoCell 310 Driver*. ARM CryptoCell 310 Driver. URL: https://doc.riot-os.org/group__pkg__driver__cryptocell__310.html (visited on 11/26/2025).

[69] RIOT. *C25519 cryptographic library*. URL: https://doxygen.riot-os.org/group__pkg__c25519.html (visited on 11/17/2025).

[70] RIOT. *GitHub - RIOT-OS/cosy: Python tool analyzing memory usage and distribution in .elf files*. URL: https://github.com/RIOT-OS/cosy (visited on 11/17/2025).

[71] RIOT. *PSA Cryptographic API*. PSA Cryptographic API. URL: https://doc.riot-os.org/group__sys__psa__crypto.html (visited on 11/26/2025).

[72] RIOT. *Relic toolkit for RIOT*. Relic toolkit for RIOT. Nov. 26, 2025. URL: https://doc.riot-os.org/group__pkg__relic.html (visited on 11/26/2025).

[73] RIOT. *RIOT - The friendly Operating System for the Internet of Things*. URL: https://www.riot-os.org/ (visited on 05/30/2025).

[74] RIOT. *Threading*. RIOT Threading. URL: https://doc.riot-os.org/group__core__thread.html (visited on 11/24/2025).

[75] R. L. Rivest, A. Shamir, and L. Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Commun. ACM* 21.2 (Feb. 1, 1978), pp. 120–126. URL: https://dl.acm.org/doi/10.1145/359340.359342 (visited on 11/07/2025).

[76] Bruce Schneier. *Applied Cryptography, Second Edition*. 1st ed. John Wiley & Sons, Ltd, 2015. URL: https://onlinelibrary.wiley.com/doi/10.1002/9781119183471 (visited on 06/03/2025).

[77] C. P. Schnorr. "Efficient signature generation by smart cards". In: *Journal of Cryptology* 4.3 (Jan. 1, 1991), pp. 161–174. URL: https://doi.org/10.1007/BF00196725 (visited on 05/27/2025).

[78] Claus Peter Schnorr. "Security of Blind Discrete Log Signatures against Interactive Attacks". In: *Information and Communications Security*. Ed. by Sihan Qing, Tatsuaki Okamoto, and Jianying Zhou. Berlin, Heidelberg: Springer, 2001, pp. 1–12.

[79] Sefik Ilkin Serengil. *serengil/LightECC*. Oct. 29, 2025. URL: https://github.com/serengil/LightECC (visited on 11/27/2025).

[80] Nigel P. Smart. *Cryptography Made Simple*. Information Security and Cryptography. Cham: Springer International Publishing, 2016. URL: http://link.springer.com/10.1007/978-3-319-21936-3 (visited on 11/03/2025).

[81] GNU TALER. *taler-crypto.test.ts « src « taler-util « packages - taler-typescript-core.git - Wallet core logic and WebUIs for various components*. taler-typescript-core cs tests. URL: https://git.taler.net/taler-typescript-core.git/tree/packages/taler-util/src/taler-crypto.test.ts#n230 (visited on 11/19/2025).

[82] Stefano Tessaro and Chenzhi Zhu. "Short Pairing-Free Blind Signatures with Exponential Security". In: *Advances in Cryptology – EUROCRYPT 2022*. Ed. by Orr Dunkelman and Stefan Dziembowski. Cham: Springer International Publishing, 2022, pp. 782–811.

[83] TrustedFirmware-M. *trusted-firmware-m/lib/ext/cryptocell-312-runtime/codesafe/src/crypto_api/pki/ec_edw/pka_ec_edw.c at main · TrustedFirmware-M/trusted-firmware-m*. GitHub. URL: https://github.com/TrustedFirmware-M/trusted-firmware-m/blob/main/lib/ext/cryptocell-312-runtime/codesafe/src/crypto_api/pki/ec_edw/pka_ec_edw.c (visited on 11/27/2025).

[84] TrustedFirmware-M. *trusted-firmware-m/platform/ext/target/arm/drivers/cc3xx at main · TrustedFirmware-M/trusted-firmware-m*. GitHub. URL: https://github.com/TrustedFirmware-M/trusted-firmware-m/tree/main/platform/ext/target/arm/drivers/cc3xx (visited on 11/17/2025).

[85] TrustedFirmware-M and Stolze. *BarschusRochus/trusted-firmware-m at edwards25519*. edwards25519 for TF-M. URL: https://github.com/BarschusRochus/trusted-firmware-m/tree/edwards25519 (visited on 11/24/2025).

[86] TrustedFirmware-M and Stolze. *BarschusRochus/trusted-firmware-m at evaluation_thesis*. evaluation branch of edwards25519 for TF-M. URL: https://github.com/BarschusRochus/trusted-firmware-m/tree/evaluation_thesis (visited on 11/24/2025).

[87] TUD. *netd-tud/taler-riot*. GitHub. URL: https://github.com/netd-tud/taler-riot (visited on 11/17/2025).

[88] TUD. *netd-tud/taler-riot at clause-schnorr*. taler-riot clause-schnorr branch. URL: https://github.com/netd-tud/taler-riot/tree/clause-schnorr (visited on 11/26/2025).

[89] TUD. *netd-tud/taler-riot at cs_evaluation*. taler-riot thesis evaluation branch. URL: https://github.com/netd-tud/taler-riot/tree/cs_evaluation (visited on 11/24/2025).

[90] user163. *Part1-Ed25519-from-the-scratch/300_point_compression.py at main · user163/Part1-Ed25519-from-the-scratch*. GitHub. URL: https://github.com/user163/Part1-Ed25519-from-the-scratch/blob/main/300_point_compression.py (visited on 11/27/2025).

[91] Loup Vaillant. *Fast Multiplication with Slow Additions*. URL: https://loup-vaillant.fr/tutorials/fast-scalarmult (visited on 11/27/2025).

[92] Filippo Valsorda. *Re-Deriving the edwards25519 Decoding Formulas*. Dec. 18, 2020. URL: https://words.filippo.io/edwards25519-formulas/ (visited on 11/27/2025).

[93] David Wagner. "A Generalized Birthday Problem". In: *Advances in Cryptology — CRYPTO 2002*. Ed. by Moti Yung. Berlin, Heidelberg: Springer, 2002, pp. 288–304.